SESSION

GRAPH AND NETWORK BASED ALGORITHMS

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

SELFISH STABILIZATION OF SHORTEST PATH TREE FOR TWO- COLORED GRAPHS

A. Anurag Dasgupta[1], and B. Anindya Bhattacharya[2]
[1]Mathematics and Computer Science, Valdosta State University, Valdosta, GA, USA
[2]Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, USA

Abstract - *In modern day stabilizing distributed systems, each process/node or each administrative domain may have selfish motives to optimize its payoff. While maximizing/minimizing own payoffs, the nodes or the domains do not require to give up their stabilization property. Optimizing individual pay offs without sacrificing the stabilization property is a relatively new trend and this characteristic of the system is termed as selfish stabilization*

The focus of this paper is to investigate the problem of finding a stable shortest path tree for two-colored graphs, where the colors represent different types of processes or domains. In a shortest path tree, for every node, its path along the tree has the minimum possible distance of any path to the root. In this paper we study the impact of selfishness on stabilization, provide examples to demonstrate the effects of different types of schedulers, and explore how the stabilization time is affected by parameter changes.

Keywords: Graph theory, stabilization, distributed systems, shortest path tree, algorithms, fault tolerance.

1 Introduction

Stabilization is an important model of fault-tolerance for distributed computation. The appeal of a stabilizing system lies in its robustness and ability to recover from any transient fault. A stabilizing distributed system has a subset of desirable states to which the system converges. These are called the set of legal states. A state not belonging to the set of legal states is called an illegal state. A system is stabilizing if and only if it satisfies two properties: a) starting from any state, it is guaranteed that the system will eventually reach a legal state (convergence), and b) given that the system is in a legal state, it is guaranteed to stay in a legal state, provided that no fault happens (closure) [1]. The above two properties guarantee that a stabilizing system will eventually recover from any transient faults that take the system to some arbitrary configuration and this recovery procedure does not require any manual intervention. For the above reasons, stabilizing systems do not need initialization and they can be spontaneously deployed. Because a stabilizing algorithm does not require correct initialization and can recover from any transient failures of arbitrary types occurring at any time, stabilization is an interesting and active research field and it is used in a large number of applications, including sensor networks, peer-to-peer networks, mobile computing, topology update, clock synchronization, and many others.

Selfish stabilization combines the concept of game theory and stabilization together. There are some strong similarities between selfish stabilization and game theory, but there are significant differences too. The players in games are analogous to processes in a stabilizing system, and the equilibrium in games is comparable to the stable configuration of stabilizing systems, in as much as both satisfy the convergence and closure properties. However, games usually start from predefined initial configurations, and mostly ignore faulty moves or transient state corruptions, which are not necessarily true for stabilizing systems [2].

In traditional stabilizing distributed systems [3], we assume that all processes run some predefined programs or algorithms. These algorithms are mandated by an external agency and most often the agency is the owner or the administrator of the entire distributed system. The model is widely recognized by the stabilization community. This works fine when processes cooperate with one another and share a purely global goal. But in modern times in the Internet, it is possible for the processes to have some private goals besides the common global goal. It is quite realistic and fairly common these days to have a distributed system spanning over multiple administrative domains and therefore processes having individual goals are not a rare occurrence. On Internet-scale distributed systems, each process or each domain may have selfish motives to optimize its own payoff besides the global goal. So the spirit of competition in such cases does not conflict with the general spirit of cooperation. Optimizing individual payoffs without sacrificing the stabilization property of the system is termed as *selfish stabilization* [4].

The focus of this paper is to finding a selfish-stabilizing shortest path tree algorithm for two-colored graphs, where the colors represent different types of processes or domains. In a shortest path tree, for every node, its path along the tree has the minimum possible distance of any path to the root. In the subsequent sections, we study the impact of selfishness on stabilization, provide examples to demonstrate the effects of different types of schedulers, and explore how the stabilization time is affected by changes to a given graph's parameter

changes. We also present examples to show how competition blends with cooperation in a stabilizing environment and provide some experimental results.

2 Background

2.1 Model and Notation

Assume a graph $G = (V, E)$. Let $V = \{1, 2, ..., n\}$ denote the set of nodes or processes and E be the set of edges connecting pairs of nodes. Let there are p different subsets or colors of nodes. In our case, $p = 2$, but in general p could be any value greater than 1. For each subset, we define a separate cost function to map the set of edges to the set of positive integers. Following our selfish stabilization algorithm, starting from any random initial configuration, the different subsets or colors of nodes will cooperate with one another to form a rooted *shortest path tree* and simultaneously compete against each other to minimize their distance with the root node.

We will assume the shared memory model for the communication among the nodes. According to this model, each process can read the states of its 1-distance neighbors and update its own state if required. In each individual step, a process checks a guarded action $g \rightarrow A$: where g is a Boolean variable. The value of g is a function of the process's own state and the states of its immediate neighbors. If g is true, the process executes action A to perform an update of its own state. If g is false, no action is taken. The global state or configuration of the system consists of the local states of all the processes. Unless stated otherwise, a serial scheduler/daemon schedules the action by randomly choosing a process with an enabled guard to execute its action.

Let us convert G into a multi-weighted graph by defining a cost function w of $E \rightarrow N^p$, where N is the set of positive integers. For every $i \in [1. . . p]$, the function $w_i : E \rightarrow N$ denotes the cost of using edge e (the distance value). Starting from any arbitrary initial configuration, the p diff erent colors of nodes cooperate with one another to form a rooted spanning tree, and at the same time compete against each other to minimize their distance value to the root.

All nodes in the graph have a common global goal in this problem: starting from an arbitrary initial configuration, each node collaborate with one another to form a rooted *shortest path tree*. But in addition to the common goal, the subsets or colors have their private goals. The private goal of each node is to optimize (in this case, it is a minimization problem) its distance value without violating the spanning tree constraints.

Fig. 1 shows an example of a two-colored graph *(a)* in which three spanning trees could be obtained at some point of a computation (none of these necessarily denotes the terminal configuration). The root is denoted by r and we chose grey color to indicate the root. For example, the cost of tree *(b)* is (10, 9), while the cost of tree *(c)* is (9, 9) and the cost of tree *(d)* is (11, 8). So, different trees yield different costs for different colors [4].

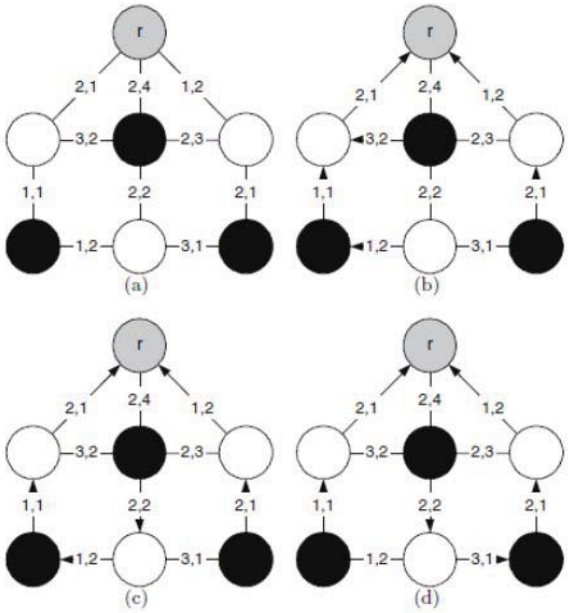


Fig. 1: Different spanning trees of the graph in part (a) (note that not all trees are terminal configurations)

2.2 Related Work

Our work is directly related to the paper by Cohen et al. [5] in which the authors described a selfish stabilization algorithm for the *minimum spanning tree* problem. The algorithm for the *minimum spanning tree* and the *shortest path tree* is essentially the same. In another paper, Dasgupta et al. [6] described a probabilistic fault-containment algorithm that stabilizes a system from minor failures with a stabilization time independent of the network size. In [7], the author described a selfish stabilization algorithm for the *maximum flow tree* problem. Cobb et al. [8] proposed a stabilizing solution to the stable path problem. Mavronicolas [9] used a game theoretic presentation to model security in wireless sensor networks where the network security is viewed as a game between the attackers and the defenders. The last one is only tangentially related to our work. It involves the spirit of competition and co-operation simultaneously as in our case, but stabilization is not an issue.

3 Algorithm

In accordance with the shared memory model, each node i can read the states of $N(i)$, the set of its neighbors (excluding i

itself). Each node *i* is also aware of the cost of each of its adjacent edges $e = (i, j) : j \in N(i)$. The cost of an edge *e* is a distance vector $w(e) = (w_1(e), w_2(e), w_3(e), \ldots, w_p(e))$ where $w_k(e)$ denotes the cost of the edge *e* for a node of color *k* ($1 \geq k \geq p$). Also, *i* maintains two variables: $\pi(i)$ and $d(i)$. The variable $\pi(i)$ denotes the parent node of *i* in the *shortest path tree*. By definition, the root does not have any parent. So, $\pi(r)$ is non-existent. Every other node picks a neighboring node as its parent following the stabilization algorithm. The variable $d(i)$ denotes the vector $d(i) = (d_1(i), d_2(i), d_3(i), \ldots, d_p(i))$ where $d_k(i)$ denotes the distance for a node of the k^{th} color from node *i* to the root.

The stabilization algorithm for *k*-colored graph is described below [5] -

Conditions

$\text{LevelOK}(i) \equiv d(i) = d(\pi(i)) + \omega(i, \pi(i))$
$\text{ParentOK}(i)_{i \in Vk} \equiv d_k(\pi(i)) + \omega_k(i, \pi(i)) = \min\{d_k(j) + \omega_k(i, j) : j \in N(i)\}$

Actions

$\text{FixLevel}(i) \equiv d(i) := d(\pi(i)) + \omega(i, \pi(i))$
$\text{FixParent}(i)_{i \in Vk} \equiv$ select $\pi(i)$: $d_k(\pi(i)) + \omega_k(i, \pi(i)) = \min\{d_k(j) + \omega_k(i, j) : j \in N(i)\}$

The proposed algorithm has a two guarded actions. The root r is exempt from any action. The other nodes update their labels to make them consistent with their parent's labels. This is in order to locally minimize the cost of the metric for the node color. The label adjustment action is taken prior to the parent adjustment action.

The actions for node $i \neq r$ are described in the following algorithm:

Program for process i

{ Fix level }
$\neg$ LevelOK(i) $\rightarrow$ FixLevel(i);

{ Fix parent }
LevelOK(i) $\wedge \neg$ ParentOK(i) $\rightarrow$ FixParent(i);

4 Observations

We make a couple of crucial observations when the algorithm is applied with different types of schedulers/daemons**.** The observations are listed below with suitable examples [10].

Observation1. Stabilization may not be feasible if at the same time, more than one process make moves i.e., if a distributed synchronous scheduler is used, it can play the role of an adversary and the configurations can repeat infinitely (Fig. 2).

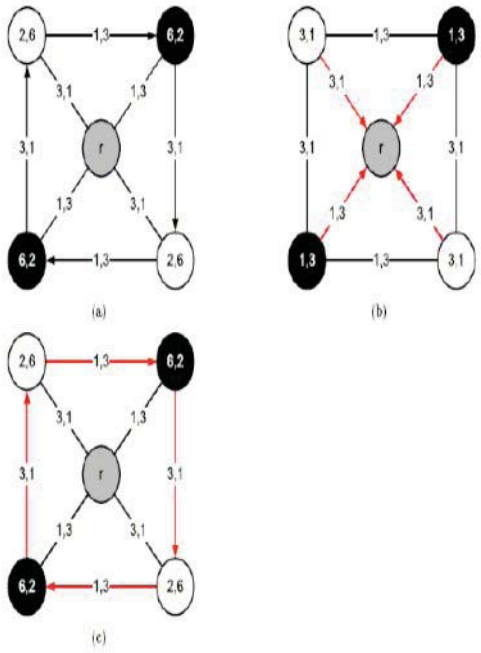


Fig. 2: Example execution with a distributed synchronous scheduler. Configuration (a) and (c) are the same, so the system can alternate between the two configurations via (b) and may never stabilize.

Observation2. More than one equilibrium are possible with the same setting for specific graphs (Fig. 3).

Consider the graph in Fig. 3. Both configurations are stable but they yield different shortest path trees, one is the best choice from the black nodes' perspective, the other being the best from white nodes' point of view.

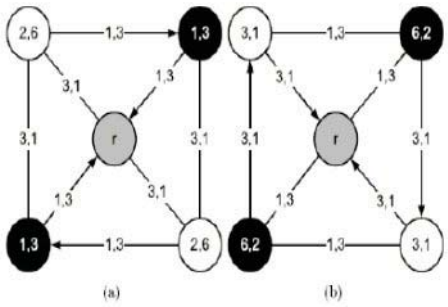


Fig. 3: Examples of multiple equilibria in a graph.

*color0 n*odes%	*color1n*odes%	Stabilization Time (ns)
10%	90%	0.30102
20%	80%	0.310594
30%	70%	0.282257
40%	60%	0.283535
50%	50%	0.286158
60%	40%	0.286584
70%	30%	0.282772
80%	20%	0.285366
90%	10%	0.347891

Table 1: Stabilization time variation for 500 nodes with respect to percentage of color change (Initially, there were 10% *color0* nodes and 90% *color1* nodes. We gradually increased the *color0* percentage up to 90%).

5 Experimental Results

The algorithm was implemented using a central scheduler on graphs of two colors, i.e., for $p = 2$. We experimented how stabilization time gets affected by changes to a given graph's color compositions and edge arrangements. In case of multiple equilibria, any one of the solutions would lead to a stable configuration and hence, it is an acceptable solution.

The first set of experiments was done by varying a given graph's color composition. We did the experiments with two colors, namely *color0* and *color1*. This can be thought of as using white nodes and black nodes as in our previous examples. We started out with 10% *color0* nodes and 90% *color1* nodes on a graph of 500 nodes. We gradually increased the *color0* percentage up to 90%, the *color1* percentage decreased accordingly. The stabilization times were measured in nanoseconds. The experiment results are listed in Table 1 and Fig. 4.

From the results, we observed that the stabilization time reaches the maximum value near the 90% mark of *color0* nodes. The stabilization time on average is the same for the range of 30%-80% range of *color0* nodes. This is intuitive as stabilization is expected to take more time when the system is tilted towards one type of color (10% of *color0* or 10% of *color1*). The trend in stabilization time variation with respect to color percentage variation is not linear towards the extreme ends, although it is somewhat linear when the graph consists of considerable percentages of both colors.

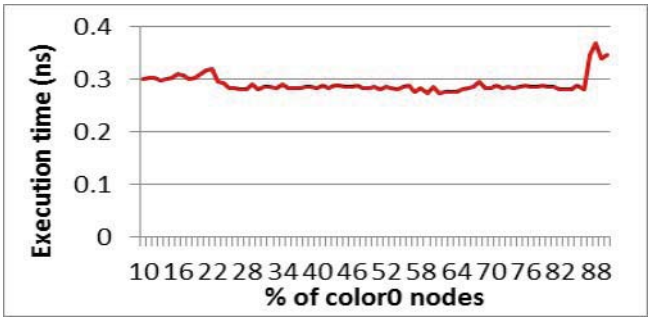


Fig. 4: Stabilization time variation for 500 nodes with respect to changes for percentage *color0* nodes and *color1* nodes.

The second set of experiments was done by varying the edge arrangements on a graph of 500 nodes. The total number of nodes is kept intact, but unlike the first experiment, this time there were equal number of *color0* and *color1* nodes. Complete graphs are considered to have 100% connectivity. We initially started with 40% connectivity. Then the number of edges for both colors was evenly increased to 90% connectivity. The experiment results are listed in Table 2 and Fig. 5.

The stabilization time steadily increases as we increment the connectivity percentages. This is expected because increasing connectivity means the degree of a node is also increased. As the degree increases, a node has to go through a list of all its neighbors before it can determine its parent node. In other words, the stabilization time computation becomes more time consuming.

Number of nodes	Stabilization Time (ns)
100	0.021436
500	0.324317
1000	1.216871
1500	2.872554
2000	7.256624

Table 2: Stabilization time variation with respect to connectivity variation (The number of edges are increased to change connectivity level from 40% to 90%).

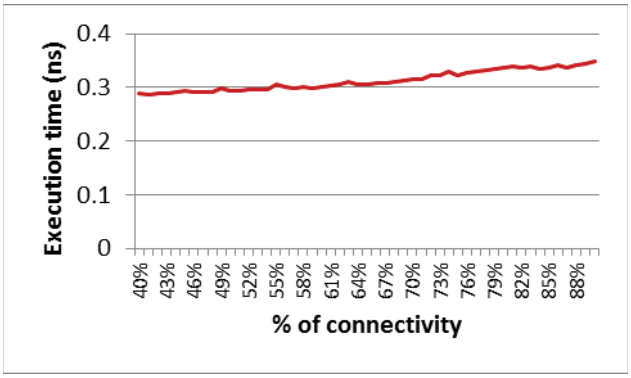


Fig. 5: The graph for stabilization time variation with respect to connectivity variation (The number of edges are increased to change connectivity level from 40% to 90%).

In the third set of experiments, we varied the total number of nodes, but kept equal number of *color0* and *color1* nodes in the graph. The experiment results are provided in Table 3 and Fig. 6.

The stabilization time steadily increases up to a certain point as we increment the total number of nodes. This makes sense as increasing the number of node means experimenting with a larger graph. Then after a certain point we observe a sudden growth in stabilization time. The 'knee' in the graph is consistent with the exponential nature of many graph algorithms. As we increased the total number of nodes, beyond a threshold value, when the algorithm has to go through a larger set of neighbors before it can determine a parent node, there is a sudden increase in the stabilization time.

% of connectivity	Stabilization Time (ns)
40%	0.290089
50%	0.293069
60%	0.301307
70%	0.314228
80%	0.335802
90%	0.34899

Table 3: Stabilization time variation with respect to total number of nodes (The experiment was done by increasing the total number of nodes but keeping equal number of *color0* and *color1* nodes in the graph).

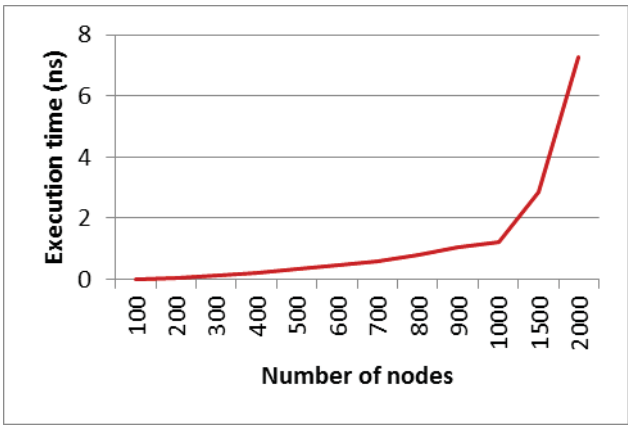


Fig. 6: Stabilization time variation with respect to total number of nodes (The stabilization time steadily increases as the total number of nodes was increased).

6 Conclusions

In the future, we would like to conduct experiments for higher values of p as there is no restriction to limit the number of colors of the graph to 2; implementing more colors would provide more insight on how changing a graph's properties affect the algorithm's run time. It is also to be seen how different topologies can affect the stabilization time. It will be interesting to see if any graph-theoretic structures can provide overall improved stabilization time.

7 References

[1] A. Arora, M.G. Gouda, Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering, 19*(11), 1993, 1015-1027.

[2] J. Y. Halpem, Computer Science and Game Theory: A Brief Survey, *CoRRabs/cs,* 2007.

[3] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control. *Commun. ACM, 17*(11), 1974, 643-644.

[4] A. Dasgupta, S. Ghosh, & S. Tixeuil, Selfish Stabilization. SSS 2006: 231-243. Dallas, TX.

[5] J. Cohen, A. Dasgupta, S. Ghosh, & S. Tixeuil, An Exercise in Selfish stabilization. *ACM TAAS, 3*(4): 2008, 1-12.

[6] A. Dasgupta, S. Ghosh, & X. Xiao, Fault-Containment in Weakly-Stabilizing Systems, *Theor. Comput. Sci., 412*(33), 2011, 4297-4311.

[7] A. Dasgupta, Selfish stabilization of maximum flow tree for two colored graphs, *The Pennsylvania Association of Computer and Information Science Educators,* California, PA, 2014.

[8] J. A. Cobb, M. G. Gouda, & R Musunari, A stabilizing solution to the stable path problem, *Self-Stabilizing Systems*, San Francisco, CA, 2003, 169-183.

[9] M. Mavronicolas, V. G. Papadopoulou, A. Philippou, & P. G. Spirakis, A graph-theoretic network security game, *WINE*, Hong Kong, SAR of China, 2005, 969-978.

[10] A. Dasgupta, Extensions and refinements of stabilization. *PhD thesis*, Department of Computer Science, The University of Iowa, Iowa City, IA, 2009.

An Algorithm for Counting the Number of Edge Covers for Graphs with Intersecting Cycles

Aparna Pamidi[1], Yijie Han[2]
[12]School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO-64110

Abstract- Counting the number of edge cover on graphs is well known as the edge cover problem. This problem is #P-complete [5]. There are algorithms that are designed to address the edge cover problem for acyclic graphs or graphs with nonintersecting cycles. In this paper we propose an algorithm that helps to compute number of edge covers for graphs with intersecting cycles as well.

1. Introduction

Graphs can be utilized to model numerous sorts of relations and procedures in physical, biological, social and information systems [2]. For example, the link structure of a website can be represented by a directed graph, in which the vertices represent the web pages and directed edges represent links from one web page to other. The improvement of algorithms to handle the graphs can even improve the performance of the system and therefore is of major interest in computer science.

An edge cover is a subset of graph edges $e \in E$ such that the union of edge endpoints corresponds to the entire vertex set V of the graph G. The problem of counting the number of edge covers of a graph is denoted as #Edge_Covers. In [1] #Edge_Covers was studied for path graphs, trees, cycles and trees with nonintersecting cycles. In this paper we present algorithms for #Edge_Covers for graphs with intersecting cycles.

2. Counting Edge Covers Based on the Structure of the Graph[3]

Let G= (V, E) be a graph. If two edges of a graph G have a common vertex $v \in V(G)$, then the edges are incident, likewise if the vertex v is on edge $e \in E(G)$ then the vertex is said to be the incident vertex of edge e.

A cyclic graph G is a graph that has at least one cycle in the graph. A graph is a directed cyclic graph if the edge set of the graph contains the ordered vertex pairs. In the case of undirected cyclic the edge set does not have any ordered vertex pairs.

Let G = (V, E) be a graph then $S = (V_1 , E_1)$ is a subgraph of G if $V_1 \subseteq V$ and E_1 contain edges $\{v, w\} \in E$ such that $v, w \in V_1$.

The algorithms given in this section was presented in [1]. We briefly go through them in order to motivate our algorithms in Section 3. In Section 3 we present algorithms for counting the number of edge covers for graphs with multiple (intersecting) cycles. In Section 2 only the situations of trees with added non-intersecting cycles are considered.

Case 1: Path Graph

A path graph is a graph that can be drawn so that all of its vertices and edges lie on a single straight line. Before counting the edge covers lets first consider few terms that are necessary.

Fixed edge: Fixed edge is an edge E that appears in all the edge covers of a graph G = (V, E).

Based on the status of the edge (i.e. edge visited or not) we find the two states on the vertex

1. Vertex free-that is the vertex is not covered by any edge in the edge cover

2. Vertex covered- that is the vertex is covered by at least one edge in the edge cover

Each edge in an edge cover is associated with a pair of integers (α, β) where α indicates the number of edge covers where this particular edge occurs to cover its preceding vertex and β indicates the number of edge covers where this particular does appear in order to cover its preceding vertex.

Counting the Edge covers for a linear structure (Path graph): For example consider a linear graph G with 7-vertices with 6 edges.

ISBN: 1-60132-434-0, CSREA Press ©

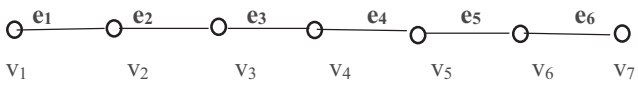


Figure 1: path graph

The edge covers for a linear graph follows a Fibonacci series pattern. Let F(n) be the number of edge covers for a linear graph of n edges, then we have

Lemma 1: F(1)=1, F(2)=1, and F(n)= F(n-1)+F(n-2).

Proof: For a linear graph the existence and non-existence of a particular edge is proportional to the existence or non-existence of the previous edge. Let us consider the linear graph with n edges- the edges e_1 and e_n should be selected in all cases of edge covers because we have to cover vertex v_1 and vertex v_{n+1}. If edge e_2 is also selected then the case becomes from edge e_2 to e_n as shown in Figure 2a and thus there are F(n-1) edge covers. If edge e_2 is not selected then edge e_3 must be selected as we have to cover v_3. In this case the situation is from edge e_3 to e_n as shown in Figure 2b and thus there F(n-2) edge covers. F(1)=F(2)=1 an be easily verified.

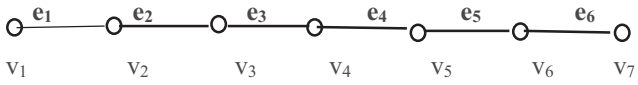


Figure 2a

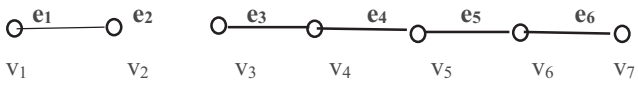


Figure 2b

F(n) is a Fibonacci number and solution for it is known as $F(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n$, where c_1 and c_2 are determined by F(1) and F(2).

Case 2: Tree Graph

A graph in which any two vertices are connected by exactly one path is known as a tree graph. In a tree graph we can distinguish three different types of edges.

i) Root Edge: An edge with one incident vertex is a root node

ii) Leaf Edge: An edge with one incident vertex is a leaf node

iii) Child Edge: All edges other than the above two types of edges are child edges i.e. the internal edges of the tree graph.

The leaf edge in the graph has (1,1) as an integer pair because the leaf vertex are only incident to leaf edge so in order to cover the leaf vertices in the edge cover these edge should be considered in all the edge covers.

Counting Edge covers for a Tree structure:

Counting the edge covers for a tree structure is a bottom-up [7] approach which follows the below form. For an edge (p, c) from parent p to child c, if c has k children then the (α, β) value for (p, c) can be written as

$$\alpha = \Sigma_{i=1}^{2^k} \prod_{i=1}^{k} r_i \qquad \beta = \Sigma_{i=1}^{2^k} \prod_{i=1}^{k} r_i - \prod_{i=1}^{k} \beta_i$$

where $r_i = \begin{cases} \alpha i \text{ if is the edge has to be considered} \\ \beta i \text{ if the edge is not to be considered} \end{cases}$

While calculating the ordered pair for an edge we should use its descendant edge order pairs only.

Consider the below figure as an example for the tree graph

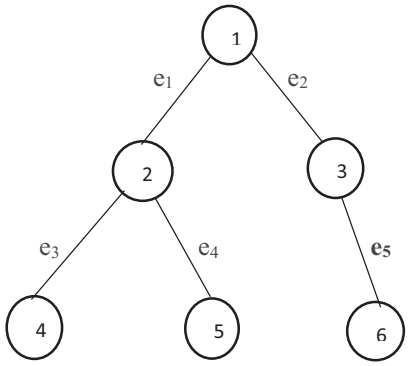


Figure 2: Tree Graph

The edges e_4, e_5 and e_6 are the leaf edges so (1, 0) are integer pair associated with these edges. For the ordered pair (α_1, β_1) of e1 we consider the (α, β) of its descendant edges in the graph (i.e. e_3 and e_4) and calculate.

$$\alpha_1 = \alpha_3 * \alpha_4 + \alpha_3 * \beta_4 + \alpha_4 * \beta_3 + \beta_3 * \beta_4$$

$$= 1*1+1*0+1*0+0*0 \quad = 1$$

$$\beta_1 = \alpha_3 * \alpha_4 + \alpha_3 * \beta_4 + \alpha_4 * \beta_3 + \beta_3 * \beta_4 - \beta_3 * \beta_4$$

$$= 1*1+1*0+1*0+0*0-0*0 = 1$$

$$(\alpha_1, \beta_1) = (1, 1)$$

For the edge e_2

$$\alpha_2 = \alpha_5 + \beta_5$$

$$=1+0$$

$$=1$$

$$\beta_2 = \alpha_5 + \beta_5 - \beta_5$$

$$=1+0-0$$

$$=1$$

$$(\alpha_2, \beta_2) = (1,1)$$

The number of edge covers for the tree is $\beta_\varepsilon = \alpha_1 * \alpha_2 + \alpha_1 * \beta_2 + \alpha_2 * \beta_1 = 1*1+1*1+1*1 = 2$

Case 3: Circular Graph (Ring)

A graph C in the ring form and with all the vertices have the same number of incident edges.

Counting Edge covers for a Cyclic Structure:

Let C(n) be the number of edge covers for a cycle of n edges. Then

Lemma 2: C(n) = F(n-1)+F(n-1)+F(n-2)……+F(2)+F(1)+1.

Proof: Let us consider a circular graph with n edges. When dealing with any edge e_i in the graph and considered the case if it is not selected then we end up with the graph similar as a linear graph. So, the n-edge graph follows the following paradigm

Algorithm Cycle:

If (the edge e_1 is not selected then we have F(n-1) edge covers)

else if (e_1 is selected and e_2 is not selected then we have F(n-1) edge covers)

else if (e_1 and e_2 are selected and e_3 is not selected then we have F(n-2) edge covers)

else if (e_1,e_2,e_3 are selected and e_4 is not selected then we have F(n-3) edge covers)

…

…

else if (e_1 through e_{n-2} are selected and e_{n-1} is not select then we have F(2) edge covers)

else if (e_1 through e_{n-1} are selected and e_n is not selected then we have F(1) edge covers)

else if (all the edges in the graph are selected then we have only one possible edge cover to represent it)

Thus the formula is derived.

For instance let us consider a circular graph G with 6-vertices. Counting the number of edge covers from

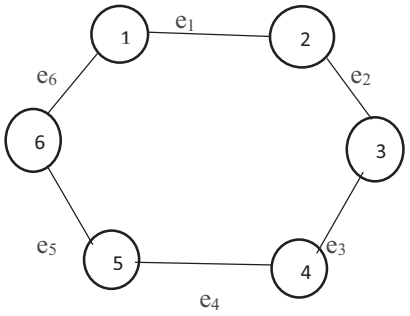
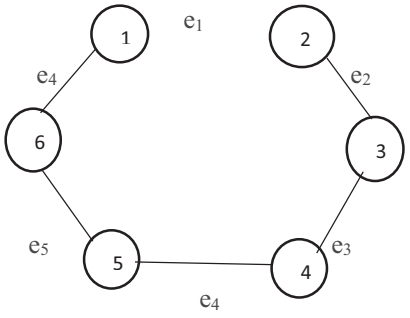


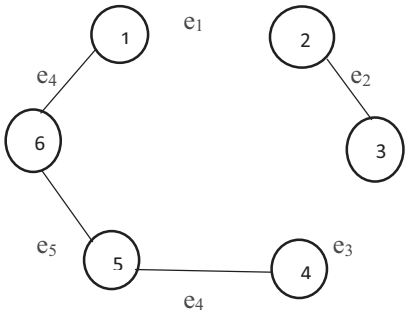
Figure 3: Circular Graph

without considering the edge1 and assuming the edge e_6 and e_2 exists in order to cover the both vertices 1 and 2 and the existence of other edges is optional. With considering all the possible cases we will retain with the 5 edge covers which is denoted by F(5).

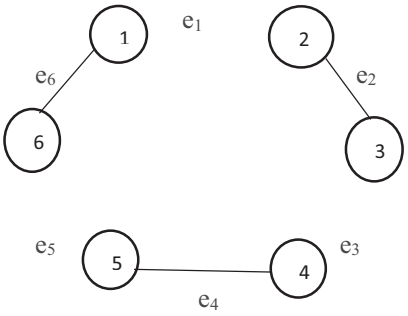
Edge Cover1



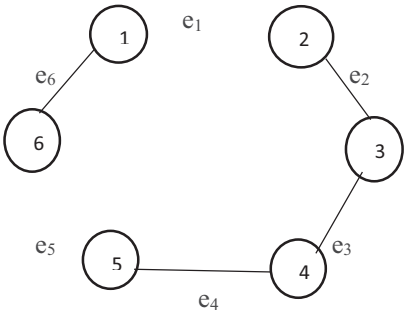
Edge Cover2



Edge Cover3



Edge Cover4



Edge Cover 5

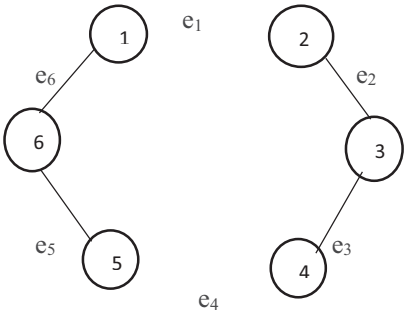


Figure 3a: edge covers for the edge e_1

The case is similar when dealing with edge e_2 but as the edge e_1 is visited before we consider the e1 as fixed edge in the all the cases when dealing with edges e_2 to e_6 and e_3 is fixed because vertex 3 has to be covered, This results F(5) (i.e. 5 different edge covers). When edge e_3 is considered we will make e_1 and e_2 as fixed edges and also e_4 because the vertex 4 has to be covered this results F(4). This process continues for all other edges in the graph so finally we will end with the form C(n).

In our case C(6) = F(5)+F(5)+F(4)+F(3)+F(2)+F(1)+1

= 5+5+3+2+1+1+1

=18 (the total number of edge covers)

3. Counting the Number of Edge Covers for Graphs with Intersecting Cycles

Graphs with intersecting cycles is built from trees, then trees with one cycle, trees with two cycles, trees with three cycles, and so on. Below is the procedure to solve it.

Tree graph with one cycle: Let Tree(T, S_1, S_2) be the procedure returning the number of edge covers for tree T with S_1 being the set of edges of T that are not selected and S_2 being the set of edges of T that are selected.

Let Tree-with-One-Cycle(TC, S_1, S_2) be the procedure returning the number of edge covers for the tree with one cycle TC assuming that edges in set S_1 are not selected and edges in S_2 are selected.

Tree-with-One-Cycle(TC, ϕ, ϕ)

{

Let e_1, e_2, e_3,…, e_c be the edges in the cycle in TC.

num_edge_cover=0;

for(i=1; i<=c; i++)

{

num_edge_cover +=Tree(TC, {e_i}, {e_1, e_2, …, $e_{(i-1)}$});

}

num_edge_cover+=Tree-Modify1(TC, {e_1, e_2, …, e_c})

}

Tree-Modify1(TC, {e_1, e_2, …, e_c}) is the procedure returning the number of edge covers for the tree obtained by replacing the cycle {e_1, e_2, e_3, …, e_c} with one vertex v. All vertices in TC that was incident to any of e_1, e_2, …, e_c are now neighbors of v. The difference for computing Tree and Tree-Modify1 is that in the procedure for Tree there must be an edge incident to v while in Tree-Modify1 we may not select all edges incident to v as all edges in the cycle are assumed to have been selected.

Tree graph with two or more cycles: A Tree graph with two or more cycles is an iterative process of the tree graph with one cycle procedure.

Let Tree-with-Two-Cycles(TC, S_1, S_2) be the procedure returning the number of edge covers for the tree with two (intersecting) cycles TC assuming that edges in set S_1 are not selected and edges in S_2 are selected.

Tree-with-Two-Cycles(TC, ϕ, ϕ)

{

Let e_1, e_2, e_3,…, e_c be the edges in the two cycles in TC.

num_edge_cover=0;

for(i=1; i<=c; i++)

{

num_edge_cover +=Tree-with-One-Cycle(TC, {e_i}, {e_1, e_2, …, e_{i-1}});

}

num_edge_cover+=Tree-Modify2(TC, {e_1, e_2, …, e_c})

}

Tree-Modify2 is the procedure returning the number of edge covers for TC after we collapse the two cycles into two vertices if these two cycles do not intersect, or collapse the two cycles into one vertex is these two cycles intersect. After collapsing cycles to vertices we obtain a tree. The difference for computing Tree and Tree-Modify2 is that in the procedure for Tree there must be an edge incident to every vertex while in Tree-Modify2 we may not select all edges incident to the collapsed vertices as all edges in the cycle are assumed to have been selected.

Example1: Consider the tree graph with one cycle in it. For the finding the number of edge covers for a graph with one cycle we will follow the procedure Tree-with-One-Cycle(TC, ϕ, ϕ) which is presented above.

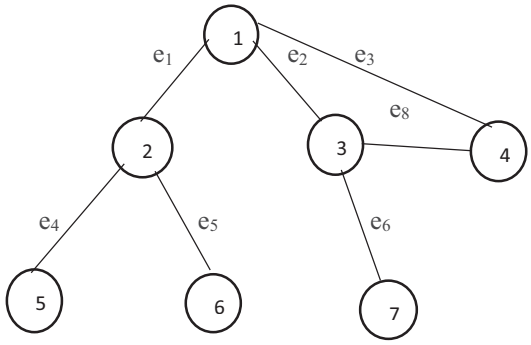


Figure 4a: tree graph with one cycle

For the above we will first remove the edge e_3 as shown in Figure 4a(i) then we are left with a simple tree graph and the procedure for discovering the edge covers for a simple tree graph is presented in Case 2. After finding the number of edge covers we will now remove the edge e_2 and have edge e_3 selected as shown in Figure 4a(ii) then we are again left with a simple tree graph follow the same procedure presented in Case 2 above.

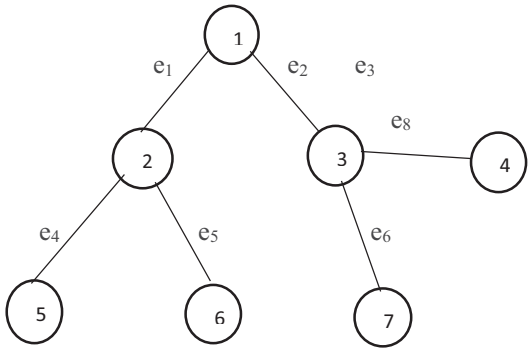


Figure 4a(i):removing the edge e_3

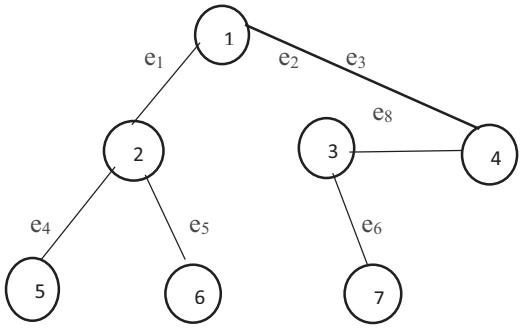


Figure 4a(ii): removing the edge e_2

Now consider the edge e_8 is removed and the edges e_2, e_3 are selected as shown in Figure 4a(iii) then apply the simple tree graph procedure to obtain the number of edge covers

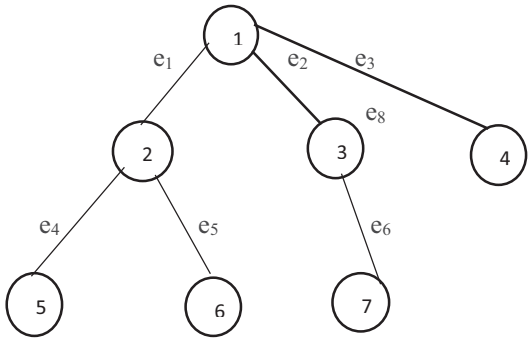


Figure 4a(iii): Removing the edge e8

Now all the edges in the cycle are selected

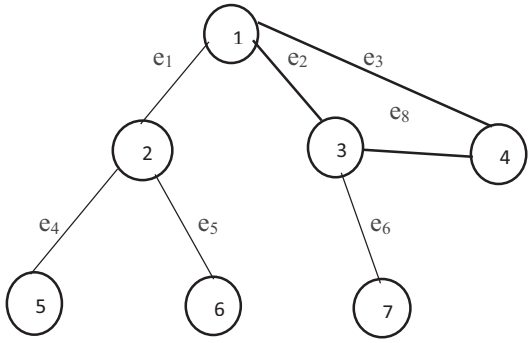


Figure 4a(iv): All the edges in the cycle are selected

After the solving the edge covers for the cycle we will collapse the entire cycle into a single vertex $v_1$as shown in the Figure 4a(v) below.

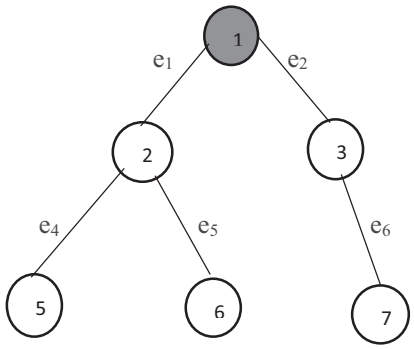


Figure 4a(v): Collapse the cycle into one vertex v_1

Example2: Consider the below figure as an example for the tree graph with two intersecting cycles

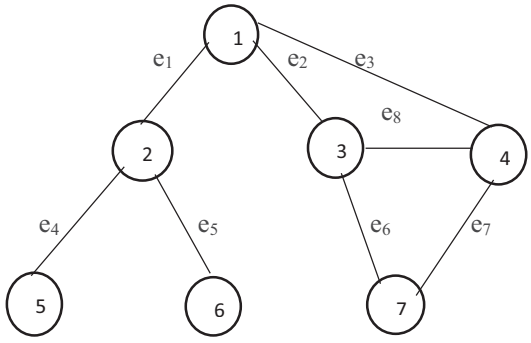


Figure 4b: Tree graph with two cycles

Form the above example we assume the edge e_3 is deleted then we ended up with a graph with one cycle shown as below Figure 4a(i). For a graph with one cycle we will follow the procedure Tree-with-One-Cycle(TC, {e3}, ϕ) and then we assume that the edge e_2 is not selected and e_3 is selected as shown in Figure 4b(ii) then we will again end up with a graph with one cycle and follow the procedure presented in the case of single cycle.

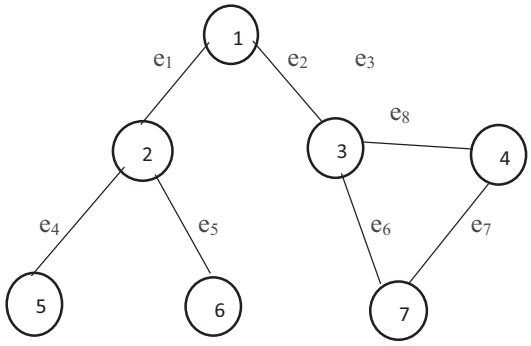


Figure 4b(i): tree graph with one cycle

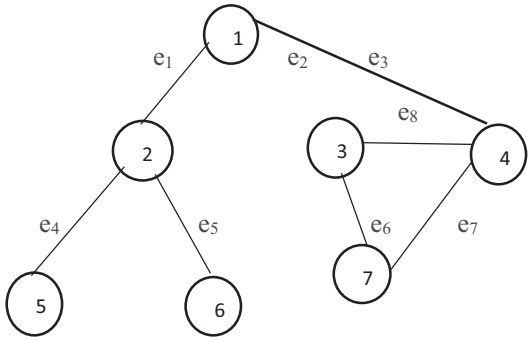


Figure 4b(ii): tree graph with one cycle

Similar is the cases with e3 and e2 selected and e8 unselected, with e3, e2, e8 selected and e6 unselected, with e3, e2, e8, e6 selected and e7 unselected. And finally with e3, e2, e8, e, e7 all selected and then we will collapse the two cycles into a single vertex v_1 in the graph as shown below Figure 4b(iii) which will result in the simple tree graph. But when we count the number of edge covers in this tree we may unselect e1 as all edges in the cycles are already selected.

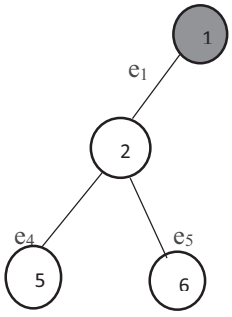


Figure 4b(iii): Simple tree after collapse of the two cycles

The example which we considered in the case of two cycles graph is a graph with two intersecting non overlapping cycles. The graph is with two overlapping cycles is actually considered as a graph with three intersecting cycles as shown in below Figure 4c. in which there are two overlapping cycles: {e1, e2, e6, e9, e5} and {e2, e8, e9, e7, e3}.

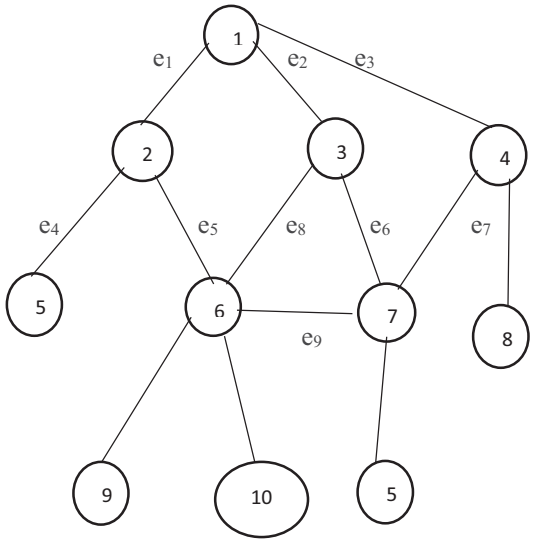


Figure 4c: Graph with three intersecting cycles

Our approach can solve the edge cover problem for graphs of any number of intersecting cycles.
The computation of number of edge covers for a graph G(V,E) with a simple cycle has a time complexity of O(n) where n being the number of edges. Likewise for a graph with two intersecting non-overlapping cycles - The first cycle can be computed in O(n) and after computation the whole cycle collapse into a single vertex then the second cycle can be solved in an another O(n) time. Therefore the total graph be computed in $O(n^2)$. The time complexity of the graph increases with respect to the number of non-overlapping intersecting cycles.

4. Conclusion and Future work

With the help of the edge covers algorithm for the simple graph structure we can even the find the edge covers for the graph with complex by applying the same procedure repetitively. This paper has shown the examples of tree graphs with two cycles in it. When the tree graph with more than two cycles similar is the case. Though finding the edge covers for the complex structure seems to be difficult but for the same structure when we solve it in a sub problems it quite easy.

5. References

[1] J.Raymundo Marcial-Romero, Guillermo D Ita, J. A. Hernandez and R.M. Valdovinos. An algorithm for counting the number of edge covers on acyclic graphs. Proc. 2015 Int. Conf. on Foundations of Computer Science (FCS'2015), 34-39(2015).

[2] https://en.wikipedia.org/wiki/Graph_theory

[3] De Ita G., Marcial-Romero J. Raymundo, Montes-Venegas Hector., ´ Counting the number of edge covers on common network topologies, Electronic Notes in Discrete Mathematics, Vol. 36, pp. 247-254, 2010.

[4] Bubley R., Dyer M., Greenhill C., Jerrum M., On approximately counting colourings of small degree graphs, SIAM Jour. on Computing, 29, (1999), pp. 387-400.

[5] Dyer M., Greenhill C., Some #P-completeness Proofs for Colourings and Independent Sets, Research Report Series, University of Leeds, 1997.

[6] Bubley R., Randomized Algorithms: Approximation, Generation, and Counting, Distinguished dissertations Springer, 2001.

[7] Tarjan R., Depth-First Search and Linear Graph Algorithms, SIAM Journal on Computing, Vol. 1, pp.146-160, 1972.

[8] Vadhan Salil P., The Complexity of Counting in Sparse, Regular, and Planar Graphs, SIAM Journal on Computing, Vol. 31, No.2, pp. 398-427, 2001.

[9] Garey M., Johnson D., Computers and Intractability a Guide to the Theory of NP-Completeness, W.H. Freeman and Co., 1979

[10] Roth D., On the hardness of approximate reasoning, Artificial Intelligence 82, (1996), pp. 273-302.

[11] Greenhill Catherine, The complexity of counting colourings and independent sets in sparse graphs and hypergraphs", Computational Complexity, 9(1): 52-72, 2000.

[12] Levit V.E., Mandrescu E., The independence polynomial of a graph - a survey, To appear, Holon Academic Inst. of Technology, 2005.

[13] Darwiche Adnan, On the Tractability of Counting Theory Models and its Application to Belief Revision and Truth Maintence, Jour. of Applied Non-classical Logics,11 (1-2),(2001), 11-34

Set Intersection and Document Search Algorithms using Tries and Graphs

Nikhita Sharma[1], Yijie Han[2]
[12]School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, Missouri, 64110

Abstract – *Designing a fast keyword search algorithm is one of the many challenges search engines face today. A search engine usually receives a conjunctive query as input and has to deliver a set of relevant results as output to users. The main factors of consideration while proposing such an algorithm is efficiency and pragmatism. In this paper, we discuss effective search algorithms for keywords search by set intersection. The main idea is to use a 'least frequent first search' approach, thereby reducing the number of set intersection computations. We also discuss better ways to perform set intersection showing trie and graph structure approaches.*

Keywords: *Search engine; set intersection; trie; graph; perfect hash;*

1. Introduction

The notion of algorithm has existed for centuries and the perception of writing an algorithm to solve a computational problem has been changing rapidly ever since. With new requirements come new challenges, and designing an effective search engine algorithm is one of the many popular and heuristic computational problems today.

There have been many approaches proposed to design text matching techniques like inverted lists [4], signature trees [5] [6], treaps [7], etc. In this paper, we will propose better approaches and substantially different techniques to achieve better performance keeping in mind the real time scenarios. We consider the problem of retrieval of documents containing multiple keywords as a conjunctive query of the form q1 ^ q2 ^ q3 ^…..^qn. If a document D is retrieved by the system, then it implies that D contains a positive result for each query $q_i \in Q$ where $1 \leq i \leq n$ and Q is set of such queries Q = {q1, q2, q3…..,qn}. Each query in this set returns a set of documents containing that query word. We need to perform an intersection on all of these sets and come up with an efficient technique to do so, so that both time and resources are minimized.

We suggest a 'least frequent first search' method where in, we reduce the time and resources consumption by starting our keyword search from the keyword with the least frequency among all documents. We implement this idea using a trie structure [8] and graph structure. The trie structure approach was first presented in [1]. However, in [1] the most frequent first search method was used. We shall show that by using the least frequent first search method, the performance is be more efficient.

2. New Approaches

Consider the problem of finding a list of documents containing a set of words. Since we have a set of documents which contain a specific word, we have to perform set intersection on multiple sets. For example, if word w1 is in set of documents S1={d1,d3,d4,d7} and word w2 is in set of documents S2={ d2,d3,d5,d6,d7}, then a set intersection $S1 \cap S2$ is required to find documents which contain both words w1 and w2, which is {d3,d7}. In order to speed up this computation, we pre-compute some data structures.

A trie is a special tree structure, where the node positions represent their associated keys unlike in other search trees where nodes store their associated keys. In [1]

a trie structure was built and a word search algorithm searching from higher to lower frequent words was presented. In Section 3, we show that a search algorithm searching from lower to higher frequent words is much more efficient.

A graph is a set of vertices and edges where vertices represent objects and edges between them represent any result or relationship obtained by mathematical or logical computations. In Section 4, we propose a completely new approach to keywords search using undirected graph design. We design an algorithm much more efficient than that discussed in Section 3 and it achieves higher performance. We retain our main idea of 'least frequent first search' in this algorithm. We also discuss the data structures to implement the graph approach in real time and show the detailed computation results.

3. The Trie Approach

In this section, we discuss various steps involved in constructing a trie structure by applying the 'least frequent first search' idea and compute results to evaluate and show that this approach gives much better performance. The trie approach was first presented in [1]. However, the most frequent first search method was used there. Here we follow the example of the trie constructed in [1] but use a least frequent first search approach.

- **Constructing Trie and Assigning Intervals**

We construct a trie structure with keywords as vertices such that keywords in the same document fall in the same path, with least frequent words at top level of the trie. Consider eleven documents D={1,2,3,4,5,6,7,8,9,10,11} and keywords in the documents as shown in Fig. 1a . The inverted list for these keywords is shown in Fig. 1b.

The root vertex of the trie is taken as a virtual vertex with empty word ϵ. We now add the least frequent keyword to the tree i.e. b and then add the next least frequent keyword, c, and so on to the tree in such a way that keywords in same document are in the same path. Whenever we do not have a compatible path for a keyword, we try adding it to another vertex in the tree at which a compatibility can be established. Only when there is no possibility of adding it to the existing vertices, we add a new subtree to the root.

DocID	Words sorted with frequency
1	a,f,d
2	a,d
3	a,d,e
4	b,a,f
5	c,d,e
6	c,f,d,a
7	a,f,d,e
8	b,f,d,e
9	c,e
10	a,f,e
11	c,f,e

Fig. 1a

b: {4,8}
c: {5,6,9,11}
a: {1,2,3,4,7,10}
f: {1,4,6,7,8,10,11}
d: {1,2,3,5,6,7,8}
e: {3,5,6,7,8,9,10,11}

Fig. 1b: Inverted lists for keywords.

We follow the digital encoding discussed in detail in [1] using concepts in [3] [9], which uses lowest rank of the sub-tree and rank of the vertex in post order traversal as starting and ending interval bounds respectively. The resultant interval sequences are shown in Fig. 1c. The complete trie and interval sequences marked to the respective vertices is shown in Fig. 1d.

b: [9,14]
c: [1,8]
a: [9,10] [15,21]
f: [4,7] [9,9] [11,13] [15,18]
d: [2,3] [4,5] [11,12] [15,16] [19,20]
e: [1,1] [2,2] [4,4] [6,6] [11,11] [15,15] [17,17] [19,19]

Fig. 1c: Interval Sequences.

We observe that there is no path with a higher frequency keyword above a lower frequency keyword i.e. the 'least frequency first search' approach is implemented. Also, vertices on the same path in the trie are in the same document.

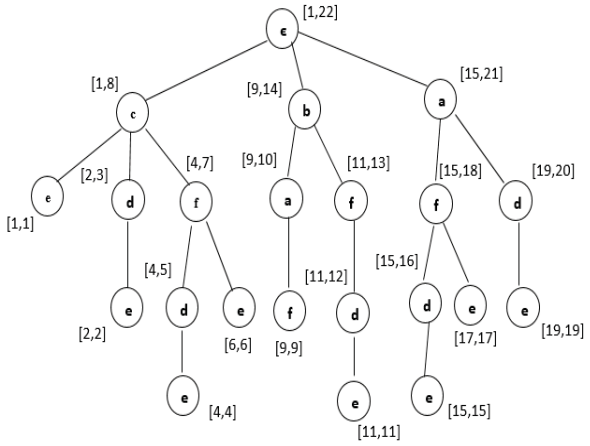


Fig. 1d: Resultant Trie structure.

- **Assigning Document Sets to Intervals**

We will also need document identifier information at each vertex to be able to find the documents corresponding to each interval or vertex. We use information from Fig. 1b and allocate document IDs to vertices such that at each vertex, the document identifiers allocated contains all words taken from root to itself. Fig. 1e shows such a trie structure.

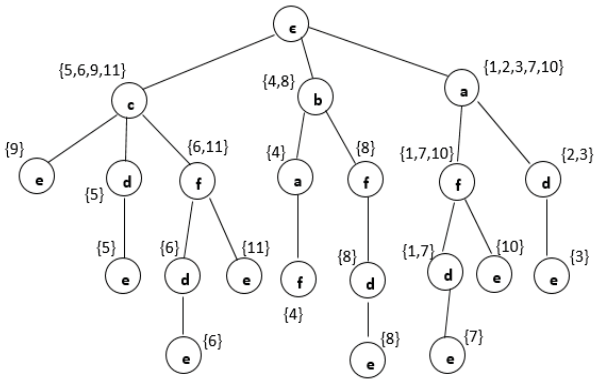


Fig. 1e: Trie with document identifiers.

- **Search Query Evaluation**

We now check interval containment to conclude if two words are in the same document. Choosing least frequent word first, reduces the number of interval comparisons to find a document containing all words in query.

We take the example of finding documents containing the words 'a' and 'b'. We choose these nodes keeping in mind that 'b' is of least frequency and 'a' has an average frequency among all other nodes. We first put words in increasing order of their frequency, i.e. b, a. Interval sequences for these are I_b = [9, 14], I_a = [9, 10] [15, 21]. Using the process proposed to check for interval containment in [1] or by observation, we have that interval [9, 10] is contained in interval [9, 14]. Here, we check if an interval of higher frequency word is contained in an interval of lower frequency word as lower frequency intervals are above higher frequency intervals in our trie. Thus by matching the resultant interval [9, 10], to document Identifiers from Fig. 1d and Fig. 1e, we conclude that document {4} contains both the words b and a.

As discussed in [1], for the same query search, intervals for b and a respectively will be I'_b = [5,5], [12,12], I'_a = [1,1] [3,3], [5,6] [8,8], [11,11] [16,16]. As compared to our approach of 'least frequency first search', we observe the below improvements:

a. Number of intervals for 'b' and 'a' are lesser, reducing the time to perform interval containment.
b. We search from lower frequency to higher frequency keyword intervals. From intervals in Fig. 1c we observe that each time we check two interval sequences, we check lesser number of intervals with another lesser number of intervals as compared in [1].
c. It is more likely that we get no interval containment when checking two less frequent words than when checking two more frequent words. Thus, by following this new approach, if there is no document

satisfying the conjunctive query, we can conclude it earlier and eliminate useless checking steps.

4. Graph Approach

We propose another solution to this problem using graph structure instead of a trie structure. Our graph will have words as vertices and a link between two vertices would represent the documents which contain both words. Similarly we could represent multiple words in a document with a clique which is a subset of vertices of a graph such that every two distinct vertices are adjacent. We implement this method using matrix along with linked lists and perfect hash tables. We retain the approach of 'least frequent first search' approach in this method.

4.1 Building Graph and Matrix

Suppose a graph G' represents a subset of a complete graph formed from the keywords and document IDs used in previous sections in Fig. 1a, such that it shows the graph structure for only c, a, f and d keywords. The document sets with any two keywords is found by using the inverted lists of keywords in Fig b. The Figure 1 represents the structure of such graph G'.

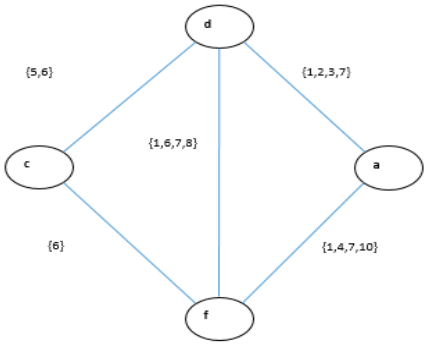


Fig. 2a: G' (c,a,f,d)

As we see in the graph G', there is no edge joining c and a as there are no documents which contain both the keywords c, a. We can represent this type of graph structure using a 4×4 matrix as shown in Fig. 2b.

	c	a	f	d
c	P_c	P_{ca}	P_{cf}	P_{cd}
a	P_{ca}	P_a	P_{af}	P_{ad}
f	P_{cf}	P_{af}	P_f	P_{fd}
d	P_{cd}	P_{ad}	P_{fd}	P_d

Fig. 2b: Matrix M'(c,a,f,d)

Here M'[c,c] = Pc can represent the set of documents containing c and element M'[c,a] = P_{ca} represents the set of documents containing c and a, and so on. In this scenario, we have $P_{ca} = \emptyset$ if there are no documents which contain both c and a.

These sets can be implemented as linked lists in real time. In such a scenario, any element M'[x,y] = p_{xy} in the above matrix will be considered as a pointer to the start of a linked list, where x and y value range from c to f. Fig. 2c shows the linked list for M'[a,f] = P_{af} and represents the set of documents containing both a and f which is [1-> 4-> 7-> 10-> NULL].

We will also build perfect hash tables for each of the document sets in Matrix M, by using a perfect hash function. A perfect hash function is a hash function which, for each unique value as key, will map to a distinct integer such that there are zero collisions. The build time and look up time for a perfect hash table for n elements is $O(n(lglgn)^2)$ and constant time of O(1) respectively [2]. Example hash table for combinations of words H_{bc} , H_{ca} and H_{af} are represented in Fig. 2d where H represents a perfect hash function.

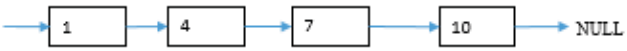


Fig. 2c: Linked List for p_{af}

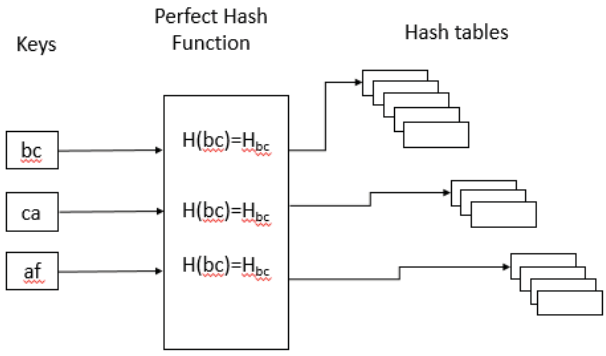


Fig. 2d: Hash Tables H_{bc} , H_{ca} and H_{af}

4.2 Computing Intersection

From the example in Fig. 1a, we construct a full graph by following process explained in Section 4.1. We will have a graph G as shown in Fig. 3. We will represent this graph as a 6×6 Matrix M as shown in Fig. 4. By looking up for the pointers we can jump to the start of linked lists containing set of resultant documents

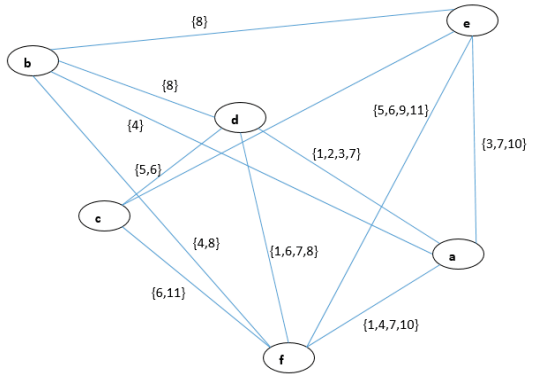


Fig. 3: Graph G

	b	C	a	F	d	E
b	p_b	p_{bc}	p_{ba}	p_{bf}	p_{bd}	p_{be}
c	p_{bc}	p_c	p_{ca}	p_{cf}	p_{cd}	p_{ce}
a	p_{ba}	p_{ca}	p_a	p_{af}	p_{ad}	p_{ae}
f	p_{bf}	p_{cf}	p_{af}	p_f	p_{fd}	p_{fe}
d	p_{bd}	p_{cd}	p_{ad}	p_{fd}	p_d	p_{de}
e	p_{be}	p_{ce}	p_{ae}	p_{fe}	p_{de}	p_e

Fig. 4: Matrix M(b,c,a,f,d,e)

In Fig. 4 , p_b = [4-> 8->NULL]; p_c = [5-> 6-> 9-> 11-> NULL]; p_{bc} = [NULL]; p_{ca} = [NULL]; p_{af} = [1-> 4-> 7-> 10-> NULL]; p_{fd} = [1-> 6-> 7-> 8-> NULL]; p_{de} = [3-> 5-> 6-> 7-> 8-> NULL]; ...so on computed from inverted lists in Fig. 1b.

We will now show how to perform set intersection on these sets (Step 1-7). We will consider the same scenario as discussed in Sections 3 and 4. If we have an inverted list as in Fig. 1b and we want to find documents containing a, f, d and e.

1. Start with the least frequent word and the next least frequent word in the query. Since, 'a' and 'f' have the least frequency, we choose 'a' and 'f' first.

2. Look up in the matrix M for the element representing pointer to document set containing both 'a' and 'f', which is M[a,f]. We get P_{af}, pointer to the linked list of Documents $LinkList_{af}$ = [1-> 4-> 7-> 10-> NULL] as represented in Fig. 2d.

3. Now, we choose the least frequent word and next least frequent word after 'f' in the query, which is 'a' and 'd' respectively. We look up for the perfect hash structure for 'ad' i.e. H_{ad}= [1, 2, 3, 7].

4. Next, we perform a search using perfect hash structure $H_{ad,}$ for each document in $LinkList_{af}$, by hashing each documentID from [1-> 4-> 7-> 10-> NULL] in H_{ad}= [1, 2, 3, 7].and return the matching document IDs to result set R_d = {7}.

5. We now have result set from previous steps representing documents containing all the words 'a', 'd' and 'f'. We repeat the above process for next least frequent word in the list which is e, but this time we will hash each document ID in the result from previous step (R_d) to hash table $H_{ae.}$

In H_{ae}= [3, 7, 10], we search for elements in result set R_d = {7} from previous computation. We finally return matching document IDs to result set R_e = {7}.

6. The final output is the last result sets we get after we cover all keywords. In this case, it is R_e = {7}, which is the final set of documents containing all words a, f, d and e.

7. At any point in the above process, if we encounter an empty set as a result set, we terminate the process and return an empty set as the result set, which concludes that there are no documents containing all the keywords given in conjunctive query. In this example tough, we do not encounter an empty result set.

4.3 Algorithm for Graph Approach and Evaluation

Consider a conjunctive query containing n words to be input which is sorted by their frequencies, in the increasing order. We denote such a query as W[2..n]. n must be at least two here as our conjunctive query must have a minimum of two keywords for the algorithm to compute intersection.

We resultant output is a set of documents containing all n words. As discussed in Section 4.2, we build a Matrix M with each pointer (p_{xy}) pointing to start of a linked list $LinkList_{xy}$. To access each document ID in the linked list in constant time, we built a perfectly hashed Hash table H_{xy} for all linked lists represented in matrix M.

ALGORITHM docSearchG(W [2..n], R)

```
begin
1.  R ← Φ;                    /* initialize result set to empty set */
2.  if n = 2                  /* two keywords query */
3.     then do /*add all docs in LinkList12 to result set */
4.     LinkList12 ← M [W[1],W[2] ] ;
5.     while (LinkList12 ! = NULL)
6.          R ← R ∪ (LinkList12 →data);
7.          LinkList12 ← (LinkList12 → next);
8.     end while
9.  else                      /* more than two keywords query */
10.    l ← W[1] ;             /* set least frequent word to l */
11.    i=3
12.    k ← W[i];
13.    while (LinkList12 ! = NULL)
14.       if searchHash (LinkList12 → data, Hlk) is true
15.          then do /* if key found, add to result set */
16.            R ← R ∪ (LinkList12 →data);
17.       end if
18.       LinkList12 ← (LinkList12 → next);
19.    end while
20.    while (R != Φ and i ≤ n)
21.       Rp = R ;  /* assign previous result set to Rp */
22.       R = Φ;                /* initialize result set R*/
23.       k ← W[i++];   /* move to next word in W */
24.       for each r in Rp
25.          if searchHash ( r, Hlk) is true
26.             then do /*if key found, add to result set */
27.                R ← R ∪ r;
28.          end if
29.       end for
30.    end while
31. end if
32. return R
end
```

From line 1-8, we check if the input has only two keywords. If so, we make result R as all the documents in $LinkList_{12}$. As we discussed before, $LinkList_{12}$ will contains documents containing both the words W[1] and W[2]. From line 9-31, we show computation if there are more than two keywords in the input. The *searchHash(r,*H_{lk}) at line 25 of the algorithm is a function which takes a document ID as first parameter and searches this documentID in the Hash table provided in the second parameter. If found, it returns true. In the second parameter (H_{lk}), the value of *l* is fixed to the least frequent word W[1] and the value of k varies from 3..n during each iteration, till all keywords are processed. This above algorithm is designed my mirroring the seven steps discussed in Section 4.2.

The output of the above algorithm is set R which contains all document identifiers, each containing the n words in them. If no document exists, which contains all the n words, this algorithm returns an empty set. The above algorithm can be evaluated for performance as discussed below.

The advantages of using a perfect hash for set intersection over linear or binary search algorithms are

significant. In case of binary search, we need to sort the input, which takes O(nlgn) time. Additional average time of O(lgn) is required for performing search on these sorted values. In our case, building a perfect hash requires $O(n(lglgn)^2)$ and search requires O(1).

Since we search from less frequent to more frequent words in the conjunctive query, we search relatively less number of documents on relatively smaller hash tables. Also, results from previous step are used to perform search in the next step. This hugely reduces the probability of checking same document IDs in multiple hash tables. Also, there are more chances of getting an empty set at an earlier step in the process, if we have no documents satisfying the query while using 'least frequent first search' approach.

5. Conclusion

In this paper, two approaches are discussed to solve the problem of set intersection for a conjunctive query. We mainly emphasize on the 'least frequent first search' method of searching keywords from a conjunctive query. This paper also explains how searching from lower frequency to higher frequency keywords considerably reduces the time required for such a search using set intersection.

In Section 3, we improved the method discussed in [1] by implementing it using 'least frequency first search'. In Section 4, we propose a complete new approach of representing keywords and documents intersections using graphs, linked lists and perfect hash functions.

Future scope of topics discussed in this paper would be look at possibilities of improving the search performance in Section 4, by using better algorithmic approaches or data structure which help in better searching.

These methods will be specifically useful in designing search engines more efficiently and offer better performance to programmers and end users.

6. References

[1]. Y. Chen, W. Shen. On the Intersection of Inverted Lists. *Proc. 2015 Int. Conf. Foundations of Computer Science FCS'2015*, in WORLDCOMP'2015. 51-57(2015).

[2]. M. Ruzic. Constructing efficient dictionaries in close to sorting time. *ICALP 2008*, Part 1, LNCS 5125, 84-95 (2008)

[3] Y. Chen and Y.B. Chen: Decomposing DAGs into spanning trees: A new way to compress transitive closures, in *Proc. 27th Int. Conf. on Data Engineering (ICDE 2011), IEEE*, April 2011, pp. 1007-1018.

[4]. J. Zobel and A. Moffat: Inverted Files for Text Search Engines, *ACM Computing Surveys*, 38(2):1-56, July 2006.

[5]. Y. Chen, Y.B. Chen: On the Signature Tree Construction and Analysis, *IEEE TKDE*, Sept. 2006, Vol.18, No. 9, pp 1207 – 1224.

[6]. Y. Chen: Building Signature Trees into OODBs, *Journal of Information Science and Engineering,* 20, 275-304 (2004).

[7]. G.E. Blelloch and M. Reid-Miller. Fast Set Operations using Treaps. In ACM SPAA, pp. 16-26, 1998.

[8]. D.E. Knuth, The Art of Computer Programming, Vol. 3, *Massachusetts, Addison-Wesley Publish Com.*, 1975.

[9]. Y. Chen and Y.B. Chen: An Efficient Algorithm for Answering Graph Reachability Queries, in *Proc. 24th Int. Conf. on Data Engineering (ICDE 2008), IEEE*, April 2008, pp. 892-901.

SESSION

QUANTUM COMPUTING + NANOTECHNOLOGY AND RELATED ISSUES + MODELING AND SIMULATION

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

Understanding Quantum Computing: A Case Study Using Shor's Algorithm

Casey J. Riggs, Charlton D. Lewis, Logan O. Mailloux, Michael Grimaila
Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio 45433, United States
{Casey.Riggs, Charlton.Lewis, Logan.Mailloux, Michael.Grimaila}@afit.edu

***Abstract*—Quantum computing is an exciting technology which utilizes the unique properties of quantum mechanics to increase the speed of classical computational operations in certain cases. However, understanding quantum computing requires knowledge of both computer science and quantum mechanics in order to develop and employ quantum algorithms. Thus, this paper provides an understandable introduction to quantum computing, and more specifically, quantum algorithms for computer scientists and practitioners. First, a number of foundational topics such as quantum measurement, RSA security, and Simon's algorithm are discussed. Next, a detailed case study of Shor's algorithm is presented as an example of how quantum algorithms can be utilized to solve computationally difficult problems.**

***Keywords*—Quantum Computing; Quantum Algorithms; Shor's Algorithm; Simon's Algorithm; RSA Encryption**

I. INTRODUCTION

Currently RSA encryption is widely employed to protect digital information including e-mails, bank transactions, and even things as simple as text messages. The security of RSA is typically measured in the amount of time it would take to break the scheme and decrypt the data. Because the decryption process is relatively quick once the scheme is broken, the inherent strength of RSA relies on the tedious nature of finding prime factors to large numbers.

Shor's algorithm grants the ability to find these prime numbers much faster than current methods. It is because the current encryption scheme is relied on so heavily by both the private and government sectors—to include the military, which drives a new field of study dubbed "post-quantum cryptography". This field is concentrated on what to do after the physical implementation of sufficiently large quantum computers and the realization of Shor's algorithm.

In 1994 Peter Shor developed a quantum algorithm (i.e., a mathematical or quantum mechanical algorithm to be executed on quantum computer) to factor large numbers with prime factors extremely quickly [11]. This discovery threatens the security of RSA encryption directly. Although a large part of the algorithm is run on a classical computer, the key component that allows Shor's algorithm to be so effective relies on quantum computing technology. Although quantum computing is still in nascent stages, researchers at MIT and the University of Innsbruck in Austria have published findings for a scalable architecture to execute Shor's algorithm [1]. Although there are challenges associated with scaling this architecture to solve larger problems, this breakthrough is instrumental in the downfall of the RSA encryption scheme [13], [21], [22], [23].

Shor's algorithm incorporates several quantum phenomena which are fundamental to quantum mechanics. It is vital to understand these quantum properties and effects before studying Shor's quantum algorithm. Additionally, Simon's quantum algorithm is also useful to understand before approaching Shor's work because it is a much more simplified period finding algorithm. A brief introduction to quantum phenomena and an abbreviated RSA encryption overview will give us the background needed to approach both Simon's then Shor's algorithm in detail.

II. QUANTUM PHENOMENA

Quantum computing offers the ability to solve relational problems rather than execute set processes. Extracting this relational information is at the heart of quantum computing. In this section, we introduce several areas of quantum mechanics necessary for understanding quantum algorithms.

A. Quantum Bits

A classical bit is restricted to existing in one of two states (either a 0 or a 1), while a quantum bit or "qubit" is a quantum-mechanical system that exists in a superposition of states (a continuum between 0 and 1). These qubits differ significantly from classical bits and because of the qubit's unique properties (i.e., the ability to put qubits into a superposition of states and entangle them with each other) means that qubits can interact naturally, and in these interactions is where large amounts of relational information is stored [17].

ISBN: 1-60132-434-0, CSREA Press ©

With regard to the Bloch Sphere in Figure 1, classical bits can exist as a unit vector in the z-direction, straight up or down. These two states can also be described in a 2-dimensional vector space as two orthonormal vectors $|0\rangle$ and $|1\rangle$. Qubits on the other hand, are able to exist in a linear combination (superposition) of these two states [16]. This is best illustrated as the state of a qubit which can exist as any unit vector in the Bloch Sphere $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, subject to the constraint $|\alpha^2| + |\beta^2| = 1$. The key difference is that the classical bit is restricted to existing solely in the direction of the unit vectors $|0\rangle$ and $|1\rangle$, while the qubit can exist in any combination of $|0\rangle$ and $|1\rangle$. This means, the qubit can exist in an infinite number of states.

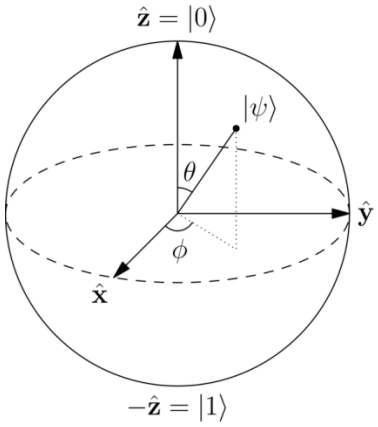


Figure 1. Bloch Sphere [9].

Many options are being considered for physical implementation of qubits including photons, trapped ions, electrons, superconducting materials, and atomic nuclei [2], [14], [15], [18], [19], [20].

B. Hadamard Gate

The Hadamard gate is often one of the first operations in a quantum circuit model, as the ability to leverage the superposition principle of the qubit is what gives a quantum computer its power. The Hadamard gate, when used to operate on a qubit, maps a single qubit into a superposition of $|0\rangle$ and $|1\rangle$ basis vectors with equal weight $|\psi\rangle = 1/\sqrt{2}\,(|0\rangle + 1/\sqrt{2}\,|1\rangle)$ where $\left|\left(\frac{1}{\sqrt{2}}\right)^2\right| + \left|\left(\frac{1}{\sqrt{2}}\right)^2\right| = 1$. This is best described as a horizontal unit vector (perpendicular to the z-vector, $\theta = 90°$) in the Bloch Sphere—a superposition of both states $|0\rangle$ and $|1\rangle$. For example, if there are 100 qubits in the model, and each is acted upon by a Hadamard gate, there now exists a superposition of all 2^{100} possible solutions within the model. However, it is not possible to measure all these solutions. In a quantum system it is only possible to measure each qubit once, and thus, obtain a single solution.

C. Measuring Qubits

In a classical computer, bits can be measured and then remain in the same state afterwards; in a quantum computer, measuring the qubits forces the qubits to collapse into a particular state of the measurement basis (e.g., either $|0\rangle$ or $|1\rangle$) [16]. Any superposition, which is where relational data is held, disappears once the qubit has been measured. This phenomenon is called the "collapse" of the qubit. It is important to note that no further data from the quantum system can be taken from the qubit after the measurement is performed, it is an irreversible process.

D. Qubit Decoherence

While purposefully measuring a qubit causes it to collapse, outside factors such as environmental noise (e.g., errant electro-magnetic waves) may also cause the quantum system to collapse before a proper measurement can be taken [8]. Quantum computing requires precisely controlled conditions in order for qubits to maintain superposition and become entangled (that the state of one qubit is dependent on the state of one or more other qubits) [17]. For example, the qubits maintained in D-wave's adiabatic quantum computer must be kept at near absolute zero in order to effectively function in superposition [12]. Whether it be isolation from electro-magnetic waves, extreme temperatures, or other unknown factors, decoherence can cause major problems with the integrity of the data stored in the qubits. Solutions to this problem include isolation from environmental factors (e.g., controlled environments and shielding), as well as quantum error correction techniques to mitigate the effects of decoherence.

E. Quantum Error Correction

In a classical computer, in order to reliably store information for long periods of time, bits can be copied, re-copied, and stored redundantly. However, in a quantum computer, it is not possible to perfectly clone an unknown quantum state [6]. This is because the measurement inherently affects the qubit you wish to copy. However, it is possible to create a series of entangled qubits and use that series as a representation of a single qubit of information, this is called a "logical qubit" [3]. If one or a few of those entangled qubits erroneously change state due to decoherence it can be corrected by assessing its conformity with the other qubits within the logical qubit.

III. RSA ENCRYPTION

Modern computer systems use public-key cryptography such as RSA which relies on the difficulty of factoring the product of two large prime numbers. For most computer systems the time it would take to factor these large numbers becomes unreasonable, and therefore public key cryptography is able to provide strong security [10].

A. Key Creation

The architecture of the RSA schema is comprised of three parts: a private key d, a public key c, and a publicly available very large number N. The process of creating these keys starts with picking two very large prime numbers; typically called p and q. Next, these numbers are multiplied to create a very large number N:

$$N = pq \tag{1}$$

After the creation of N, Euler's totient of N is created, which is the total number of integers less than N which are relatively prime to N (i.e., all the integers in the totient and N have a greatest common divisor of 1). Because Euler's totient is multiplicative we know:

$$\varphi(N) = \varphi(p)\varphi(q) \tag{2}$$

Also, because we chose p and q as prime numbers, we know $\varphi(p) = p - 1$ and $\varphi(q) = q - 1$. This allows us to create the totient of N:

$$\varphi(N) = (p-1)(q-1) \tag{3}$$

We now choose the public key c which is relatively prime to the totient of N, meaning the greatest common divisor of the totient of N and c is 1. The fastest way to know if a chosen number and the totient of N are relatively prime is by using the Euclidian algorithm to calculate the greatest common divisor and check if it really is 1:

$$\gcd\big(c, \varphi(N)\big) = 1 \tag{4}$$

Next, in order to calculate the private key, d, we need to calculate the modular inverse of our public key c. This is done by using the extended Euclidian algorithm. This process solves the following equation for d [2]:

$$cd = 1 mod(\varphi(N)) \tag{5}$$

After the creation of the private key d, the cryptosystem is complete and the encryption/decryption process can begin. At this point it is important to understand that only the large number N and the public key c are publicly available. The private key d is only known by the individual to whom it belongs and the totient $\varphi(N)$ is discarded.

B. Encrypting/Decrypting with RSA

Once the private-public key pairs are created and appropriate distribution techniques are established, the encryption process is relatively straightforward. To encrypt the message a Bob wants to send to Alice, it is first encrypted using both Alice's public key, c, and the large number N which are available to Bob because they are public knowledge. The encrypted message is denoted by the letter b:

$$b = a^c mod(N) \tag{6}$$

When Alice receives the encrypted message b she is able to decrypt the message using her private key:

$$a = b^d mod(N) \tag{7}$$

A simple overview of the public key encryption scheme is provided in Figure 2.

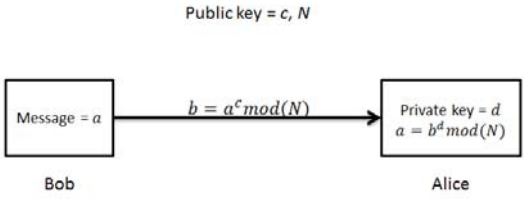


Figure 2. Illustration of public key cryptography.

IV. Understanding Quantum Algorithms

Before moving on to a complex quantum algorithm such as Shor's algorithm, understanding another—Simon's algorithm makes the approach significantly easier. As Shor's algorithm is a specific implementation of Simon's algorithm, an overview of Simon's period finding algorithm is useful. The quantum Fourier transform will be introduced later because it is used in Shor's algorithm to speed up the period finding process.

A. Simon's Algorithm

In 1997, Daniel Simon introduced a quantum algorithm to reduce the number of measurements required to solve an unknown period problem [5]. In a classical computer, finding an unknown period a takes order $O(2^{n/2})$ measurements, while Simon's technique only requires $O(n)$ measurements where n is the number of bits needed to represent the period in base 2 [3]. The classical method is akin to a guess and check until the unknown period is found and as the size of the period a grows, the number of measurements grows exponentially along with it. Using Simon's algorithm, as the size of the period a grows, the number of measurements only grows linearly with n.

Simon's algorithm works through a series of quantum operations and measurements. First, the input and output registers must be initialized, which is by convention done in the state $|0\rangle$. Next, each qubit in the input register is operated on by a Hadamard transformation, putting the qubits into a state of equal superposition of all possible combinations. The state of the system is described as [4]:

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle\,|0\rangle \tag{8}$$

where $|x\rangle$ represents the input register after the Hadamard transformation such that $|x\rangle$ is in a superposition state and $|0\rangle$ represents the output

register still in its initialization state. Next, the unitary transform $\hat{U}_f$ is applied to the superposition state of the input register $|x\rangle$ and stored in the output register, the new state of the system becomes [3]:

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle |f(x)\rangle \tag{9}$$

After the unitary transform $\hat{U}_f$ operates, the output register holds the results of the function $|f(x)\rangle$, while the input register $|x\rangle$ is still in a state of superposition. Now suppose a measurement of the output register $|f(x)\rangle$ is taken, and thus, collapses both the output and input registers. The output register collapses to a random evaluation of x called $f(x_0)$. The input register can now only exist in one of two states: $|x_0\rangle$ or $|x_0 \oplus a\rangle$ according to the generalized Born rule [7]. This is because the function (the unitary transform $\hat{U}_f$) is defined as having the same result for two specific inputs (i.e., the function is periodic under bitwise modulo-2 addition) and $f(x) = f(x \oplus a)$. The resulting state of the input register is [3]:

$$\frac{1}{\sqrt{2}}(|x_0\rangle + |x_0 \oplus a\rangle) \tag{10}$$

The input register, although it now contains valuable information (i.e., we can solve for a given both states), is not as useful as it seems because the register can only be measured one time. Successive trials would yield more random values for $|x_0\rangle$ and $|x_0 \oplus a\rangle$ satisfying different measured outputs, which would not help solve for the unknown period a efficiently.

The next step in this process is to again apply the Hadamard transformation to the input register $|x_0\rangle + |x_0 \oplus a\rangle$, and the state of the quantum system becomes [3]:

$$\frac{1}{\sqrt{2^{n+1}}} \sum_{y=0}^{2^{n-1}} |(-1)^{x_0 \cdot y} + (-1)^{(x \oplus a) \cdot y}\rangle |y\rangle \tag{11}$$

where $|y\rangle$ represents the output register. More simply, the input register can be interpreted as the expansion coefficient of the output register ($|(-1)^{x_0 \cdot y} + (-1)^{(x \oplus a) \cdot y}\rangle$ becomes a_y) and Eq. (11) simplifies to [4]:

$$\sum_{y=0}^{2^{n-1}} a_y |y\rangle \tag{12}$$

From Eqs. (11) and (12), we know that the coefficient of the output register $|y\rangle$ will be 0 if $a \cdot y = 1$. Because the probability of a measurement is represented by the absolute value squared of the expansion coefficient, $|a_y|^2$, this means the probability of measuring a solution in which $a \cdot y = 1$ is 0. Thus, the output register $|y\rangle$ is limited only to solutions in which $a \cdot y = 0$.

For this reason, any measurement of Eq. (12) must yield a random y in which $a \cdot y = 0$, where each y value obtained reduces the possible choices for the period a by half. This allows the unknown period a to be found in only $O(n)$ invocations of Simon's algorithm by the creation of a system of equations for a which is comprised of n equations.

B. Quantum Fourier Transform

The quantum Fourier transform (QFT) is an important part of Shor's algorithm because when introduced, it emphasizes a relationship between the states of an input register, the period of the function, and the total size of the register. The QFT (denoted as U_{FT}) like all other valid quantum operations is a linear, unitary operator. The QFT maps n qubits to n qubits (the output size of the QFT is the same as the input size in terms of number of qubits), and the effect of the QFT on a register is [3]:

$$U_{FT}|x\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^{n-1}} e^{2\pi i x y / 2^n} |y\rangle_n \tag{13}$$

The QFT operates on the input register $|x\rangle$ to create a set of states in the output register $|y\rangle$ with the probabilities of measurement of $\left|e^{2\pi i x y / 2^n}\right|^2$ for each state. The QFT, like the other operators, can also operate on a superposition of states which is invaluable for Shor's algorithm.

V. SHOR'S ALGORITHM

Introduced in 1994, Shor's algorithm is a quantum algorithm designed to quickly solve prime factors of a given number which is of great concern in modern cryptography—specifically the RSA public key cryptography [11]. The method Shor created to solve these prime factors utilizes a number of classical computing processes and only leverages quantum computing to solve one aspect of the problem—finding the period. This piece of Shor's algorithm is a specific realization of Simon's algorithm.

As shown in Table 1, Shor's factoring process can be summarized in five steps, of which only the fourth step is quantum in nature—the very same step is the most computationally intensive part of the process [4].

Table 1. A summary of the factoring process [4].

1. If N is even, return a factor of 2. Otherwise, continue to the next step.
2. Check whether $N = a^b$ for integers a and b such that $a \geq 1$ and $b \geq 2$. If $N = a^b$ then return the factor a.
3. Randomly choose an integer $x \in \{2,3,\cdots, N-1\}$ and compute $\gcd(x, N)$. If $\gcd(x, N) > 1$ then return the factor $\gcd(x, N)$. If $\gcd(x, N) = 1$ (i.e., if x is a coprime of N) then continue to the next step.
4. Find out the order r [period] of $x^r mod N$. If r is even and $x^{r/2} mod N \neq -1$ then continue to the next step. Otherwise, restart from Step 3 with a different x.
5. Compute $gcd(x^{r/2} \pm 1, N)$ and check whether one of them is (or both of them are) nontrivial factor (factors) of N. If so, then return the factor (factors). Otherwise, restart from Step 3 with a different x.

The remainder of the paper focuses specifically on understanding Shor's quantum algorithm contribution as described in step 4, where the order r is the period of the function which needs to be found. Just as in Simon's algorithm, we will consider both an input and an output register throughout each step.

A. Understanding Shor's Quantum Algorithm

The output register must be able to hold N, in binary form. This means, for example, if $N = 64$ the output register must contain at least 6 qubits because 64 is represented within 6 binary digits ($2^6 = 64$). The size of the output register (the number of qubits required), will be denoted as l.

The input register generally needs to have twice as many qubits as the output register ($2l$). This configuration is desirable so that the input register can contain at least N different states that produce the same output — this gives us more "workspace" with which to capture the period of the function. The size of the input register is denoted as t.

Entering step 4 of the process, we know that the number to be factored is N and we have already chosen an x which is coprime to N. First, the input and output registers must be initialized to a known value (typically $|0\rangle$):

$$|\psi\rangle_0 = |0\rangle_t |0\rangle_l \tag{14}$$

The quantum system or total wave function of the system is written as $|\psi\rangle$ at step 0 with the input and output registers (t and l, respectively) initialized to $|0\rangle$. Next, the input register is put through t Hadamard gates, placing the input register $|0\rangle_t$ into a state of superposition represented as $|k\rangle$ [4]:

$$|\psi\rangle_1 = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |k\rangle |0\rangle \tag{15}$$

Next, the superposition state $|k\rangle$ is operated on by a modular exponent and the result is stored in the output register [4]:

$$|\psi\rangle_2 = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |k\rangle |x^k mod(N)\rangle \tag{16}$$

Notice the similarity with Simon's problem with this method. Next, a measurement of the output register, yields a random value of $x^k mod(N)$ called z_0. This measurement forces the input register into a state of superposition of all the possible inputs that would yield the measured value z_0, satisfying the generalized Born rule [7]. The total number of valid input states is represented as M. The function is periodic so we know that the valid inputs for a particular solution are $f(d + mr) = z_0$, where the value of d is the smallest possible input for this function that yields z_0 and any multiple m of the period r added to the smallest value d will yield the same z_0.

Focusing on the input register, which now contains the values of interest, and temporarily disregarding the output register, the total wave function at step 3, without the output register is now [4]:

$$|\psi\rangle_3 = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} |d + mr\rangle \tag{17}$$

Similar to Simon's problem, valuable information is stored in the input register and if it was possible to make a copy of it, the period r could be found in a small number of measurements. However, only one measurement yielding a random number can be taken and successive measurements would yield more random numbers for different measured outputs.

Since, the number of qubits in the input register is double the output register, the number of solutions that can simultaneously exist in the input register satisfying $|d + mr\rangle$ is large. Thus, the next step is to apply a quantum Fourier transform to the input register yielding [4]:

$$|\psi\rangle_4 = \frac{1}{\sqrt{M}} \sum_{m=0}^{M-1} \frac{1}{\sqrt{2^t}} \sum_{y=0}^{2^t-1} e^{2i\pi(d+mr)y/2^t} |y\rangle \tag{18}$$

where the input register is now represented as $|y\rangle$ and useful information can now be measured.

B. Finding the Period

Simplifying Eq. (18) and using the substitution $\xi = e^{2i\pi\frac{yr}{2^t}}$ gives a wave function of the input register [4]:

$$|\psi\rangle_4 = \frac{1}{\sqrt{2^t M}} \sum_{y=0}^{2^t-1} e^{2i\pi dy/2^t} \left(\sum_{m=0}^{M-1} \xi^m |y\rangle \right) \tag{19}$$

From this wave function, the probability of measuring any particular $|y\rangle$ is given by [4]:

$$\frac{1}{2^t M}\left|\sum_{m=0}^{M-1}\xi^m\right|^2 \quad (20)$$

This means the inputs will constructively interfere when $\frac{yr}{2^t}$ is close to an integer and destructively interfere when $\frac{yr}{2^t}$ is otherwise. This raises the probability of measuring a particular input y that, if C is an integer, satisfies $\frac{yr}{2^t} \approx C$. Moreover, if this value of y is close to an integer, we know that $\xi \approx 1$, and therefore the probability of measurement is [4]:

$$\frac{1}{2^t M}\left|\sum_{m=0}^{M-1}\xi^m\right|^2 = \frac{M^2}{2^t M} = \frac{M}{2^t} \approx \frac{1}{r} \quad (21)$$

Thus, the probability of measuring a specific value in the input register $|y\rangle$ that satisfies $\frac{yr}{2^t} \approx C$ is approximately $\frac{1}{r}$, which is much higher than the values in the input register which destructively interfere.

The final quantum step of Shor's algorithm is to measure the input register $|y\rangle$. The result of this measurement is assumed to follow the high likelihood that $\frac{yr}{2^t} \approx C$. Assuming this is true, we can rearrange the equation to understand the relationship better [4]:

$$\frac{y}{2^t} \approx \frac{C}{r} \quad (22)$$

The quantum part of Shor's algorithm is now complete and the rest can be handled by a classical computer. The quantum aspects of Shor's algorithm result in a high likelihood of a solution which satisfies a relationship between the period r, the solution space 2^t, an integer C, and the measured result y. Since the result y and solution space 2^t are known, we can solve for the left half of Eq. (22) and find an equivalent integer fraction to solve the right hand side. More specifically, the continued fraction method is used to solve for the period r.

Since we know that C is likely an integer, thus $2C$, $3C$,…,etc. are also likely integers. This means that when we find the equivalent fraction for the right hand side we must also consider that $\frac{2C}{r} = \frac{C}{r/2}$ and $\frac{3C}{r} = \frac{C}{r/3}$ and so on, are valid solutions as well. Using the number of steps to convergence in the continued fraction, an initial value for the period r is generated. The initial period r must be double checked by substituting the value r back into the original equation we are trying to solve:

$$x^r mod(N) = 1 \quad (23)$$

If the statement is incorrect, then small multiples of r can be tried, since C, $2C$, $3C$,…,etc. are all integers. This process is used to find the smallest period r that satisfies Eq. (23).

Lastly, the value r must also be even and satisfy the condition $x^{r/2} mod N \neq -1$. If r does not satisfy these conditions, the quantum algorithm must be re-accomplished with a new value for our initial coprime number x. Once this step has been accomplished successfully and one or both prime factors of N has been found—the factoring process would be complete. If only one prime factor is found, simple division of N by the known value would yield the other prime factor. Knowing the prime factors to N would effectively break the RSA encryption because once the prime factors are known the private key can be computed easily.

C. Breaking RSA

To break the RSA encryption an alternate step may also be used. An overview of this attack on RSA public-key encryption is provided in Figure 3.

After finding the period r, a pseudo-private key d' can be created satisfying [3]:

$$cd' = 1 mod(r) \quad (24)$$

Using this value for d', the original content of the encrypted message b can be easily decrypted [3]:

$$a = b^{d'} mod(N) \quad (25)$$

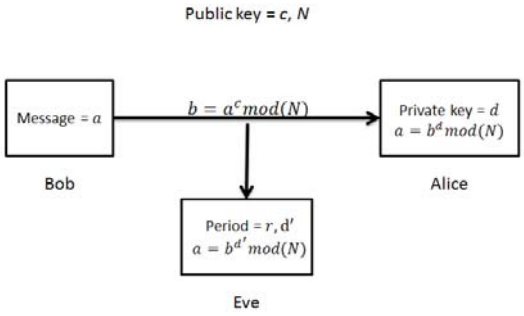


Figure 3. An overview of the alternate method to break public key encryption using the period.

VI. CONCLUSIONS

Peter Shor made a very important contribution to the field of quantum algorithms with his realization of quantum period finding—its relation to the RSA encryption scheme has drawn international acclaim and notoriety from renowned security specialists. However, there have been many other discoveries as to the types of computations quantum computers can perform. Currently, three classes of algorithms: (i) algebraic and number theoretic; (ii) oracular; and (iii) approximation and simulation are highlighted in the "quantum zoo," the most complete compendium of quantum algorithms available [24]. Unfortunately, each of these algorithms needs to be further studied

and expanded upon as they wait to be applied on a quantum computer.

Further study of this area needs to run parallel with the kinds of difficult problems we are facing using classical computers to determine how we can leverage the strengths of quantum computing. In this work, we have built a foundation for understanding quantum algorithms by first understanding the quantum phenomena necessary for quantum computing and then demonstrated the importance of applying quantum algorithms by using Shor's algorithm. This work provides a starting point for those interested in quantum computing and quantum algorithms.

DISCLAIMER

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the U.S. Government.

REFERENCES

[1] T. Monz, D. Nigg, E. A. Martinez, M. F. Brandl, P. Schindler, R. Rines, S. X. Wang, I. L. Chuang, and R. Blatt. "Realization of a scalable Shor algorithm." arXiv preprint arXiv:1507.08852 (2015).

[2] M. A. Nielsen., and I. L. Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[3] N. D. Mermin. *Quantum computer science: an introduction*. Cambridge University Press, 2007.

[4] A. Pathak. *Elements of quantum computation and quantum communication*. Taylor & Francis, 2013.

[5] D. R. Simon. "On the power of quantum computation." *SIAM journal on computing* 26, no. 5 (1997): 1474-1483.

[6] W. K. Wootters, and W. H. Zurek. "A single quantum cannot be cloned." *Nature* 299, no. 5886 (1982): 802-803.

[7] Born, Max. "Quantenmechanik der stoßvorgänge." *Zeitschrift für Physik* 38, no. 11-12 (1926): 803-827.

[8] M. Schlosshauer. "Decoherence, the measurement problem, and interpretations of quantum mechanics." *Reviews of Modern Physics* 76, no. 4 (2005): 1267.

[9] [Bloch Sphere]. Retrieved February 25, 2016 from https://upload.wikimedia.org/wikipedia/commons/thumb/f/f4/Bloch_Sphere.svg/2000px-Bloch_Sphere.svg.png

[10] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems." Communications of the ACM 21, no. 2 (1978): 120-126.

[11] P. W. Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." SIAM review 41, no. 2 (1999): 303-332.

[12] R. Harris, J. Johansson, A. J. Berkley, M. W. Johnson, T. Lanting, S. Han, P. Bunyk et al. "Experimental demonstration of a robust and scalable flux qubit." Physical Review B 81, no. 13 (2010): 134510.

[13] R. Van Meter, and C. Horsman. "A blueprint for building a quantum computer." Communications of the ACM 56, no. 10 (2013): 84-93.

[14] L. M. K. Vandersypen, M. Steffen, G. Breyta, C.S. Yannoni, M. H. Sherwood, and I. L. Chuang. "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance." Nature 414, no. 6866 (2001): 883-887.

[15] D.Kielpinski, C. Monroe, and D. J. Wineland. "Architecture for a large-scale ion-trap quantum computer." Nature 417, no. 6890 (2002): 709-711.

[16] E. Schrödinger. "Discussion of probability relations between separated systems." In Mathematical Proceedings of the Cambridge Philosophical Society, vol. 31, no. 04, pp. 555-563. Cambridge University Press, 1935.

[17] A. Einstein, B. Podolsky, and N. Rosen. "Can quantum-mechanical description of physical reality be considered complete?." Physical review 47, no. 10 (1935): 777.

[18] C. Lu, D. E. Browne, T. Yang, and J. Pan. "Demonstration of a compiled version of shor's quantum factoring algorithm using photonic qubits." Physical Review Letters 99, no. 25 (2007): 250504.

[19] B. P. Lanyon, T. J. Weinhold, N. K. Langford, M. Barbieri, D. F. V. James, A. Gilchrist, and A. G. White. "Experimental demonstration of a compiled version of shor's algorithm with quantum entanglement." Physical Review Letters 99, no. 25 (2007): 250505.

[20] A. Politi, J. C.F. Matthews, and J. L. O'brien. "Shor's quantum factoring algorithm on a photonic chip." Science 325, no. 5945 (2009): 1221-1221.

[21] I. Chuang, R. Laflamme, P. Shor, and W. Zurek. "Quantum computers, factoring, and decoherence." arXiv preprint quant-ph/9503007 (1995).

[22] R. Landauer. Is quantum mechanically coherent computation useful?. IBM Thomas J. Watson Research Division, 1994.

[23] W. G. Unruh. "Maintaining coherence in quantum computers." Physical Review A 51, no. 2 (1995): 992.

[24] National Institute of Standards and Technology. "The Quantum Zoo." Available at: http://math.nist.gov/quantum/zoo/.

Design of n-bit Reversible Adder with LNN architecture

Md. Belayet Ali
GS of Electrical Engineering and Computer Science
Iwate University
4-3-5 Ueda Morioka Iwate, 020-8551 Japan
Email: belayet@kono.cis.iwate-u.ac.jp

Takashi Hirayama, Katsuhisa Yamanaka, Yasuaki Nishitani
Department of Systems Innovation Engineering
Iwate University
4-3-5 Ueda Morioka Iwate, 020-8551 Japan
{hirayama,yamanaka,nisitani}@kono.cis.iwate-u.ac.jp

***Abstract*—We present a new design for reversible adder that realizes quantum arrays in one-dimensional Ion Trap technology. In this architecture all gates are built from $2x2$ quantum primitives that are located only on neighbor qubits in a one dimensional space, which is also called Linear Nearest Neighbor (LNN) architecture. This proposed reversible adder circuits are different from most of reversible adder circuits obtained by synthesis methods that use only high level quantum cost based on the number of quantum gates. This means that most of the previous works have not considered the cost depending on the distance between two qubits in a gate, even if these qubits are located far away in physical space one from another. From a practical point of view and with respect to nanotechnologies such as quantum optics, nuclear magnetic resonance (NMR), and Linear Ion Trap technology, our proposed design of reversible adder is very cost effective.**

***Keywords*-Reversible Adder circuits; Quantum Cost; Quantum gate Linear Nearest Neighbor Architecture; Nanotechnology;**

I. Introduction

The basis of thermodynamics of information processing was shown that conventional irreversible circuits unavoidably generate heat because of losses of information during the computation. It has been shown by Landauer that for every bit of information lost in logic computations that are not reversible, $kT*log_2$ joules of heat energy per computing cycle is generated, where k is Boltzmanns constant and T the absolute temperature at which computation is performed [1]. This resulting dissipated heat also causes noise in the remaining circuitry, which results in computing errors. Bennett showed that the dissipated energy directly correlated to the number of lost bits, and that computers can be logically reversible, maintain their simplicity and provide accurate calculations at practical speeds [2]. Therefore, logical reversibility is a necessary (although not sufficient) condition for physical reversibility. In fact less power dissipation in logic circuits is possible only if a circuit is composed of reversible logic gates.

Most papers in the literature about automated synthesis of quantum and reversible (permutative) circuits are not related to any particular quantum realization technology [3], [4], [5], [6], [7], [8]. The model used in most of the previous permutative quantum circuit synthesis assumes that there can exist a gate located between any two qubits, even if these qubits are located far away in physical space (in vector) one from another. This assumption was accepted in a theoretical framework but from a practical point of view and with respect to particular technologies (such as Ion trap in this case) creating gates on arbitrary qubits is not only extremely difficult but also cost ineffective; each gate has to be properly converted and realized in an LNN architecture. Thus, in general architecture independent synthesis models are sufficient to approximate the real cost of small circuits. For larger quantum circuits realized in the future as well as for currently realizable circuits with about 12 qubits, architecture dependent cost models and synthesis methods are required. For instance in quantum optics [6], [7] such architectural models require more development to take into account more complex constraints such as time propagation and physical size. In quantum optics, qubits also interact by proximity using optical wires or crystals [6], [9], [10]. Therefore, it is safe to assume that the LNN cost model is currently one of the most appropriate models for current technologies. Circuits realized in LNN use quantum gates defined only on neighbor qubits and the gates are built from 1x1 and 2x2 quantum primitives. We believe that LNN model should be used for Ion Trap and similar technologies and new quantum cost models should be developed for other specific technologies.

Adders are a key element in any arithmetic logic unit. It is therefore important to have fast reversible adders. Early implementations of binary adders using reversible logic, such as [11], suffered from the generation of garbage bits (n in the case of an n-bit ripple-carry adder). The novel approach to ripple-carry adders introduced by Vedral et al. [12] (VBE-adder) solves the problem with generation of garbage. It is obvious that the VBE-adder is not optimal in the number of gates, logic width and logic depth. Two n-bit adders based on this optimization were suggested by Cuccaro et al. [13] (CDKM-adder). Another improvement is the VanRentergem-adder [14], which has the lowest gate costs compared to the VBE-adder and the two CDKM-adders. However, all the mentioned adder only uses for reversible modular arithmetic and we know that, the modular arithmetic result does not reflect the overflow (carry out) of

ISBN: 1-60132-434-0, CSREA Press ©

arithmetic operations. Moreover, some quantum technology required to realize the circuit in an LNN architecture but all the existing works neglect the restrictions of LNN based quantum circuits. The implementation of adder circuit using LNN model is not yet done by any researcher. With respect to general quantum circuits the LNN model was introduced by Fowler et al [14] for designing a Quantum Fourier transform circuit and their work was improved in [15]. The paper [16] considers theoretical aspects of techniques for translating quantum circuits between various architectures.

This paper shows that the realization of an efficient reversible adder with LNN for a programmable computing device is possible and that the proposed design is a very versatile approach to the design of quantum networks. In this work we proposed n-bit adder circuits using LNN model and we believe that LNN model of adder circuits should be used for Ion Trap and similar technologies and new quantum cost models should be developed for other specific technologies. The work [18] uses the concept of merge gates if both Controlled-NOT and Controlled-V or Controlled-$V^\dagger$ acting on the same two qubits in a symmetric pattern, their total cost is considered as unit. Due to the fact that gates in quantum circuits can be reordered in many different ways. So, the question is whether more than two quantum gates acting on the same two qubits can be replaced with a single two-qubit gate of unit cost. This question also suffices which elementary quantum gate sequences can be formed as single gates that can be used cost effectively, and hence results in reduced the search space in synthesis procedure. Paper [19],[23] has shown that if a sequence of 1-qubit and 2-qubit quantum primitives in a circuit act on the same two qubits, then the sequence of gates can be represented by a unitary matrix, and the logic operations in the sequence can be performed as a two-qubit function of unit cost. In this work, the rectangle box represented as a single gate with unit cost. This section shows a complete set of reversible adders for 1-bit and n-bit with LNN model. We apply optimization technique [20] and 2-qubit gate library [19] to obtain the optimized circuit after the elementary gate representation and LNN circuit of the adder circuit.

The paper is organized into the following sections. Section 2 is an overview of reversible logic, quantum computing and motivation for the LNN model. Section 3 is the proposed design. Result analysis of the proposed design is presented in section 4 and conclusions are contained in section 5.

II. Preliminaries

We present the basic concepts of reversible and quantum circuits with logic operation and the motivation for the LNN model for quantum technology in this section.

In a binary boolean context, a reversible gate is an elementary circuit component that realizes a bijection. To satisfy this requirement, the function must have the same number of inputs and outputs and commonly used traditional NOT gate is the only reversible gate. A reversible function can be realized by cascading reversible gates with fanout-free and feed back free realization. Many reversible gates have been proposed, in which Toffoli, Peres and Fredkin are conventionally used to synthesize reversible circuits.

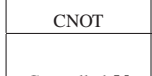
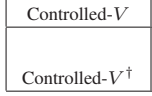
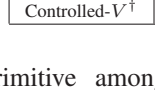
On the other hand, the logic representation in quantum computation is quite different from the logic representation in classical computation. The basic unit of information in quantum computation is a qubit represented by a state vector. The states $|0\rangle$ or $|1\rangle$ are known as the computational basic states. The state of an arbitrary qubit is described by the following vector [23]

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}, \tag{1}$$

where α and β are complex numbers which satisfy the constraint $|\alpha^2| + |\beta^2| = 1$. The measurement of qubit results in either 0 with probability $|\alpha^2|$, that is, the state $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ or 1 with probability $|\beta^2|$, that is, the state $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$. Contrary, a classical bit has a state either 0 or 1, which is analogous to the measurement of a qubit state either $|0\rangle$ or $|1\rangle$ respectively. The main difference between bits and qubits is that a bit can be either state 0 or 1 whereas a qubit can be a state other than $|0\rangle$ or $|1\rangle$ according to (1).

Many quantum gates have been defined and studied but we concentrate on the elementary quantum gates NOT, CNOT, Controlled-V and Controlled-$V^\dagger$, also known as quantum primitives. This gates have been widely used to synthesis of binary reversible functions. The elementary gates are represented by their graphical representation [23] as shown in Table I.

Table I
Elementary quantum gates and their graphical representations.

Gate Name	Gate Symbols
NOT	
CNOT	
Controlled-V	
Controlled-$V^\dagger$	

Any one primitive among CNOT, Controlled-V and Controlled-$V^\dagger$ can be formed by cascading the other two primitives, referred to as splitting rules [23] that are shown in Fig. 1 (a) and (b) respectively. Moreover, Controlled-V and Controlled-$V^\dagger$ can be replaced with each other resulting in two more splitting rules shown in Fig. 1 (c) and (d) respectively. The inverse of splitting rules is referred to as merge rules. However, in quantum computation, the splitting of a quantum primitives does not increase the number of

two-qubit operations. If two adjacent gates are identity then delete two gates and known as deletion rules in quantum primitives [23]. Therefore, two NOT gates, two CNOT gates and an adjacent V , $V^\dagger$ pair (any order) with the same target and control can be removed that are shown in Fig. 2.

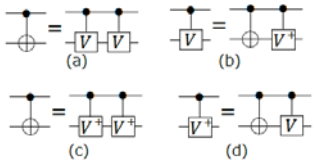


Figure 1. Splitting and Merge rules in Quantum Primitives

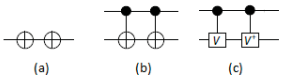


Figure 2. Deletion rules in Quantum Primitives

Definition 1: The size of a circuit C is defined as the number of its gates and denoted by $|C|$. The size of an NCV circuit is also known as quantum cost.

From the denition [23], we can say that the number of elementary quantum gates required for an implementation of a reversible circuit.Consider the following Fig.3, which is the elementary quantum gates representation of Toffoli-3 gate with quantum primitives. The quantum cost of Toffoli-3 is 5 as it required 5 quantum primitives for implementation.

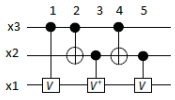


Figure 3. Toffoli-3 gate with Quantum Primitives

The mobility of gates is determined by the moving rule that relies on the following property [24]:

Property 1: Two adjacent gates g_1 and g_2 with controls c_1 and c_2 and targets t_1 and t_2 can be interchanged if $c_1 \cap t_2 = \emptyset$ and $c_2 \cap t_1 = \emptyset$.

To prove this property, consider Fig. 3 where gates g_1 and g_2 with controls x_3 and x_3 and targets x_1 and x_2 can be interchanged because $x_3 \cap x_2 = \emptyset$ and $x3 \cap x1 = \emptyset$. After that, gates g_1 and g_3 with controls x_3 and x_2 and targets x_1 and x_1 can be interchanged because it satisfies the condition. In this way, gate g_1 can be moved anywhere in the circuit.

A gate between any two qubits would mean an immediate direct interaction between any two qubit in the circuit, which is physically impossible some technology such as Ion Trap due to space separation [21], [22]. In the simplest case, all ions in Ion Trap are placed linearly (as a One-Dimensional vector). Every qubit can interact with at most one neighbor above and one neighbor below. This physical constraint of 2-neighbor quantum layout of the substrate has much influence on practical designs. Fig.4 shown the LNN circuit for Toffoli-3 with quantum cost 9 which is however quite expensive. It has 9 2x2 gates in 2-neighbors-only topology after the minimization of certain gates. Conventional way to calculate the quantum cost of the gate as a function of number of inputs regardless of what is the distance of the qubits used in this gate. This is not accurate when the circuit is realized in linear Ion Trap technology. Nor is it good for quantum optics or NMR technology that is currently in use. There are other ways to realize this gate in layout, even without ancilla bit. They are however even more expensive when realized in linear Ion Trap.

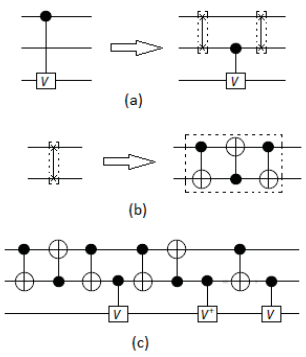


Figure 4. Toffoli-3 gates when mapped to linear-neighborhood quantum array

III. Proposed Model

A. Half Adder

Reversible half adder is implemented with two reversible gates of which one Toffoli-3 and one NOT gate is shown in Figure 5(a) [25]. The number of garbage outputs is one represented as G and garbage input is one represented by logical zero. The equation of the half adder is as follows.

$$Sum = A \oplus B$$

$$C_{out} = AB$$

The elementary quantum gate realization is shown in Figure 5(b). Initial quantum cost of the elementary quantum gate realization of half adder circuit is six. After optimized the circuit shown in figure 5(b) quantum cost is four as it requires two controlled-V gates each costing one, one Controlled-$V^\dagger$ gate with costing one and one NOT gate with cost one which is shown in figure 5(c). This would be fine if every two qubits can interact directly but they cannot. So we required to transform Figure 5(c) to an LNN circuit. To obtain the LNN circuits we use swap gate. Finally we optimized the LNN circuits using quantum primitives rules

such as splitting, merging and deletion rules and also used gate library [19]. The final circuit for the half adder from Figure 5(d) is then shown in Figure 5(e). The quantum cost of LNN based half adder is 6 which is however expensive than previous one.

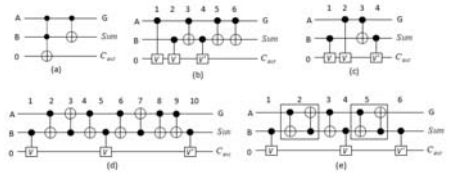


Figure 5. (a) reversible half adder using reversible gate, (b) elementary quantum gate realization, (c) optimized quantum circuit, (d) LNN realization of optimized quantum circuit, and (e) optimized LNN realization

B. Full Adder

The design of reversible full adder with two Toffoli-3 gates and two NOT gates is as shown in Figure 6(a) [25]. The three inputs are A, B and C_{in} and the outputs are Sum and C_{out}. The number of garbage input is one represented by logical zero. The garbage outputs are two represented by G. The equation of the full adder circuit is as follows.

$$Sum = A \oplus B \oplus C_{in}$$

$$C_{out} = AB \oplus (A \oplus B)C_{in}$$

Initial quantum cost of the elementary quantum gate realization full adder circuit is eight. After optimized the adder circuit, quantum cost is six as it requires three Controlled-V gates each costing one, one Controlled-$V^{\dagger}$ gate with costing one and two NOT gate with cost one which is shown in Figure 6(b). If we observe the optimized elementary quantum gate realization of full adder circuit we can see that gate 3 and gate 5 cannot interact directly. So we required to transformations Figure 6(b) to create LNN circuits. We obtained LNN circuits by using swap gate which is shown in Figure 6(c). Finally we optimized the LNN circuits using quantum primitive rules such as splitting, merging and deletion rules and also using gate library [19]. The rectangle box represents single gate which is obtained from gate library. The final optimized circuit for the full adder from Figure 6(c) is then shown in Figure 6(d). The quantum cost of LNN based full adder circuit is ten. Our main goal is to build n-bit adder circuit with LNN architecture. We can build n-bit adder circuit by integrating the full adder and half adder. We found that to build n-bit adder circuit with LNN architecture the final optimized LNN circuit for full adder shown in Figure 6(d) not satisfy the LNN restrictions when integrate with half adder or full adder. To satisfy the LNN restrictions we modify the design of full adder by moving line shown in Figure 7(a). Optimized quantum circuit using elementary quantum gate realization is shown in Figure 7(b). LNN realization of the optimized quantum circuit and optimized LNN circuit are shown in Figure 7(c) and 7(d) respectively. The new quantum cost of the modified optimized LNN realization is sixteen which is however expensive than the previous realization. But the new optimized LNN realization is maintain the LNN restriction.

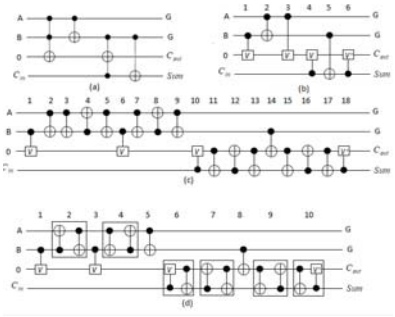


Figure 6. (a) reversible full adder using reversible gate, (b) optimized quantum circuit, (c) LNN realization of optimized quantum circuit, and (d) optimized LNN realization

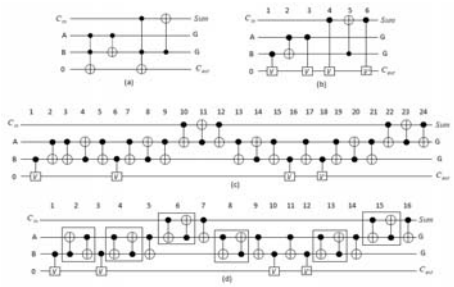


Figure 7. After moving C_{in}(a) reversible full adder using reversible gates, (b) optimized quantum circuit, (c) LNN realization of optimized quantum circuit, and (d) optimized LNN realization

Table II
QUANTUM COST OF ADDER CIRCUITS WITHOUT LNN ARCHITECTURE

	Half Adder	Full Adder
Quantaum Cost	4	6

Table III
QUANTUM COST OF ADDER CIRCUITS WITH LNN ARCHITECTURE BEFORE AND AFTER OPTIMIZATION

	Half Adder		Full adder		Full adder (Modified)	
	Before	After	Before	After	Before	After
Quantaum Cost	10	6	18	10	24	16

C. Reversible n-bit binary adder

Integrating the optimized LNN circuit of half adder and full adder, reversible n-bit adder can be constructed [25]. First we construct the n-bit adder using reversible gates

which shown in Figure 8(a) for n-bit adder and Figure 9(a) for n-bit adder with C_{in}. Note that in both adders the carry is propagated immediately from one stage to the next. This property helps to construct a fast n-bit adder to reducing the circuit delay by immediately propagating the carry-out. Figure 8(b) and 9(b) show the n-bit adder circuits using LNN model for Figure 8(a) and 9(a) respectively. Table V shows the results of the n-bit adder in terms of quantum cost. As we discussed in the previous sections, we use new optimized LNN realization of full adder circuit to build the n-bit adder circuit with LNN architecture due to the LNN restrictions. It will increase the quantum cost of the n-bit adder circuit but this circuit is applicable in the recent technologies. We also tried to design full adder with different line arrangement but our new design shows the better result than other arrangement.

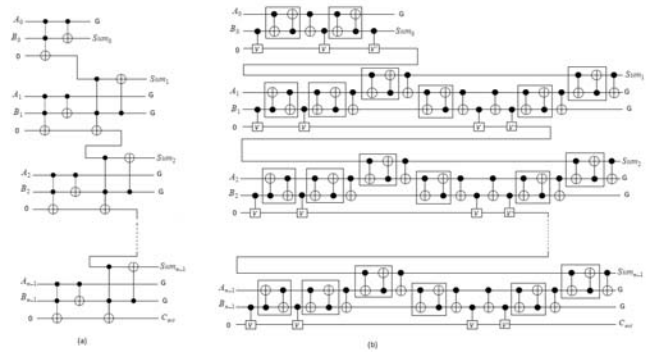


Figure 8. (a) reversible n-bit adder, (b) and optimized LNN realization of reversible n-bit adder

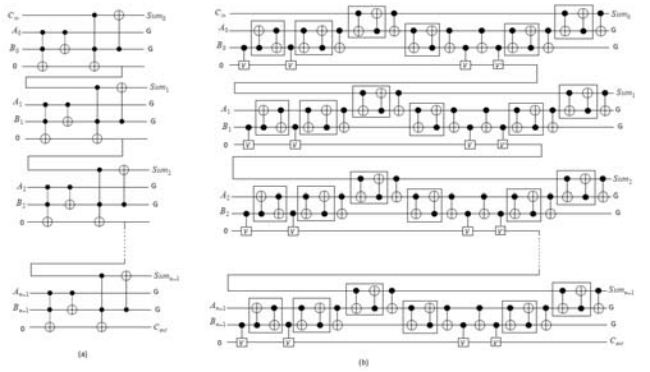


Figure 9. (a) reversible n-bit adder using full adder with C_{in}, (b) and optimized LNN realization of reversible n-bit adder with C_{in}

Table IV
RESULT OF n-BIT ADDER WITHOUT LNN ARCHITECTURE

	N-bit Adder	N-bit Adder with C_{in}
Quantaum Cost	$6n-2$	$6n$

Table V
RESULT OF n-BIT ADDER WITH LNN ARCHITECTURE

	N-bit Adder		N-bit Adder with C_{in}	
	Before	After	Before	After
Quantaum Cost	$24n-14$	$16n-10$	$24n$	$16n$

IV. RESULT ANALYSIS

There are many existing designs of reversible full adder circuits which mainly concentrate the quantum cost in terms of the number of elementary quantum gates required to design. They assume that there can exist a gate located between any two qubits, even if these qubits are located far away in physical space one from another. This assumption was accepted in a theoretical framework but from a practical point of view and with respect to particular technologies (such as Ion trap) creating gates on arbitrary qubits is not only extremely difficult but also cost ineffective; each gate has to be properly converted and realized in an LNN architecture. Considering the physical constraint we proposed new design for n-bit adder circuits. It has been shown that the quantum cost of LNN circuits are higher that non LNN circuits. The Tables III and V show the result of our proposed model. Tables II and III show the quantum cost of adder circuits without LNN architecture and with LNN architecture before and after optimization respectively. Without LNN the quantum cost of half adder and full adder is 4 and 6 respectively which is the best known result till now. To make this circuit applicable for the LNN architectures, SWAP gates are applied for the each non-adjacent quantum gate. More precisely, SWAP gates are added in front of gate with non-adjacent control line to move the control line of gate towards the target line until they become adjacent. We can move the target line towards the control line until they become adjacent as well. Afterwards, SWAP gates are added to restore the original ordering of circuit lines. The initial quantum cost of the LNN circuit was 10 but after optimization we obtain a quantum cost of 6. Similar way, the quantum cost of the initial LNN circuit of full adder was 18 and after optimization the quantum cost is 10. However, to build n-bit adder circuit we need to satisfy the LNN restriction. To do so we modify the design of full adder by moving line which we already discussed earlier section. In Table III, the last two columns show the result of the modified design of full adder circuit. The initial quantum cost of the LNN circuit was 24 but after optimization we obtain a quantum cost of 16. However, as can easily be seen, synthesizing quantum circuits for LNN architectures using this method often leads to a significant increase in the quantum cost but, from the practical point of view and with respect to particular technologies this circuit is applicable. Table IV and V show the result of n-bit adder circuits with and without LNN architecture.

V. Conclusions

This paper presents reversible adders for n-bit using LNN architecture which should be used in quantum optics, linear Ion Trap, NMR and similar technologies where every qubit can interact with at most one neighbor above and one neighbor below. This physical constraint of 2-neighbor quantum layout of the substrate has much influence on practical designs. We also have shown that to design n-bit adder circuit we need to design full adder by line reordering, which leads to a significant increase in the quantum cost. With respect to practical point of view and recent technologies this circuit is applicable. In future, the design can be extended to other different types of Adder/Subtractor unit, Multipliers, Dividers and finally full phase low power Reversible ALUs for recent technologies such as quantum optics, linear Ion Trap, NMR where LNN architecture has much influence on practical designs.

References

[1] R. Landauer, *"Irreversibility and Heat Generation in the Computing Process "*, IBM J. Research and Development, vol. 3, pp. 183-191, July 1961.

[2] C. Bennett, *"Logical Reversibility of Computation "*, IBM Journal of Research and Development, vol. 17, pp. 525-532, 1973.

[3] A. Mischenko and M. Perkowski, *"Logic synthesis of reversible wave cascades "*, Proceedings of IWLS, pp. 197-202, 2002.

[4] A. Khlopotine, M. Perkowski, and P. Kerntopf *"Reversible logic synthesis by gate composition "*, Proceedings of IWLS, pp. 261-266, 2002.

[5] D. M. Miller, D. Maslov, and G. W. Dueck, *" Synthesis of quantum multiple-valued circuits"*, Journal of Multiple-Valued Logic and Soft Computing, vol. 12, no. 5-6, pp. 431-450, 2006.

[6] G. Yang, W. Hung, X. Song, and M. Perkowski, *"Majority-based reversible logic gates "*, Theoretical Computer Science, vol. 334, no. 1-3, 2005.

[7] W. N. N. Hung, X. Song, G. Yang, J. Yang, and M. Perkowski, *"Optimal synthesis of multiple output boolean functions using a set of quantum gates by symbolic reachability analysis "*, IEEE Transaction on Computer-Aided Design of Integrated Circuits and systems, vol. 25, no. 9, pp. 1652-1663, 2006.

[8] N. Alhagi, M. Hawash, and M. Perkowski, *"Synthesis of Reversible Circuits for Large Reversible Functions "*, Facta Universitais, Series: Electronics and Energetics, vol. 24, no. 3, pp. 273-289, December 2010.

[9] Lihui Ni, Zhijin Guan, and Wenying Zhu, *"A General Method of Constructing the Reversible Full-Adder "*, Third International Symposium on Intelligent Information Technology and Security Informatics, pp. 109-113, 2010.

[10] Irina Hashmi and Hafiz Md. Hasan Babu, *"An Efficient Design of a Reversible Barrel Shifter "*, Twenty Third International Conference on VLSI Design, pp. 93-98, 2010.

[11] H Thapliyal and N Ranganathan, *"Design of Reversible Latches Optimized for Quantum Cost, Delay and Garbage Outputs "*, Proceedings of Twenty Third International Conference on VLSI Design, pp. 235-240, 2010.

[12] Vlatko Vedral, Adriano Barenco, and Artur Ekert, *"Quantum networks for elementary arithmetic operations "*, Physical Review A, 54(1):147-153, Jul 1996.

[13] S. Cuccaro, T. Draper, S. Kutin, and D. Moutlon, *"A new quantum ripple-carry addition circuit "*, quant-ph/0410184, 2004.

[14] Yvan Van Rentergem and Alexis De Vos, *"Optimal Design of a Reversible Full Adder "*, International Journal of Unconventional Computing, vol. 1, pp. 339-355, 2005.

[15] A. Fowler, S. Devitt, and L. Hollenberg, *"Implementation of shors algorithm on a linear nearest neighbor qubit array "*, Quantum Information and Computation, vol. 4, no. 4, pp. 237-251, 2004.

[16] Y. Takahashi, N. Kunihiro, and K. Ohta, *"The quantum fourier transform on a linear nearest neighbor architecture "*, Quantum Information and Quantum Computation, vol. 7, no. 4, pp. 383-391, 2007.

[17] D. Chang, D. Maslov, and S. Severini, *"Translation techniques between quantum circuits architecture "*, http://www.iqc.ca/severin/qipabs.pdf.

[18] W. Hung, X. Song, G. Yang, J. Yang, and M. Perkowski, *"Optimal synthesis of multiple output Boolean functions using a set of quantum gates by symbolic reachability analysis "*, Transactions on Computer Aided Design, vol. 25, no. 9, pp. 1652-1663, 2006.

[19] M. B. Ali, T. Hirayama, K. Yamanaka and Y. Nishitani, *" New Two- Qubit Gate Library with Entanglement "*, Note on Multiple-Valued Logic in Japan, Vol.38, No.1, (8):1-8, 12-13, September, 2015.

[20] M. B. Ali, T. Hirayama, K. Yamanaka and Y. Nishitani, *"Quantum Cost Reduction of Reversible Circuits Using New Toffoli Decomposition Techniques "*, International Conference on Computational Science and Computational Intelligence (CSCI'15), pp. 59-64 Las Vegas, USA, December 7-9, 2015.

[21] J. Cirac and P. Zoller, *"Quantum computation with cold trapped ions "*, Physical Review letters, vol. 74, no. 20, p. 4091, 1995.

[22] D. Wineland and T. Heinrichs, *"Ion trap approaches to quantum information processing and quantum computing "*, A Quantum Information Science and Technology Roadmap, vol. N/A,2004.

[23] M. M. Rahman, *"Synthesis of Reversible Logic "*, Doctor of Philosophy, In the Graduate Academic Unit of Faculty of Computer Science, The University of New Brunswick, December, 2014.

[24] D.Miller R. Wille and R. Drechsler, *"Reducing reversible circuit cost by adding lines"*, IEEE International Symposium on Multiple-Valued Logic, pp. 217-222, 2010.

[25] R. Feynman, *"Quantum mechanical computers"*, Optics News 11, pp. 11-20, 1985.

Oscillation Model for Network Dynamics Caused by Asymmetric Node Interaction Based on the Symmetric Scaled Laplacian Matrix

Masaki Aida
Graduate School of System Design
Tokyo Metropolitan University
Hino-shi 191–0065, Japan
Email: aida@tmu.ac.jp

Chisa Takano
Graduate School of Information Sciences
Hiroshima City University
Hiroshima-shi 731–3194, Japan
Email: takano@hiroshima-cu.ac.jp

Masayuki Murata
Graduate School of Information Sciences
Osaka University
Suita-shi 565–0871, Japan
Email: murata@ist.osaka-u.ac.jp

***Abstract*—Since recent development and dissemination of ICTs activate information exchange on social networks, the dynamics for describing propagation of activities on the networks has became an interesting research object. This paper proposes an oscillation model describing the propagation of activities on social and information networks. In order to analyze such dynamics, we generally need to model asymmetric interaction between nodes. This paper discusses a symmetric matrix-based model that can describe some types of link asymmetry. Although the proposed model is simple, it can reproduce well-known indices of node centrality and can be considered as the underlying mechanism of network dynamics. As an application of the proposed model, we show a framework to estimate natural frequency of networks by utilizing resonance.**

***Keywords*—Laplacian matrix, coupled oscillators, node centrality, resonance**

I. Introduction

Information exchange on social networks is being activated by the popularity of information networks. So, complex dynamics for describing propagation of activities on the social and information networks is a rich source of research topics. In complex network analysis, there are a lot of indices that can describe the characteristics of networks, including degree distribution, clustering coefficient, and many kinds of node centralities [1], [2], [3].

Spectral graph theory is a key approach for investigating the structure of networks [4], and the eigenvalues and the eigenvectors of the Laplacian matrix are important when investigating network structure. Spectral graph theory is applicable to many problems including clustering of networks, graph drawing, graph cut, node coloring, and image segmentation [4]. One of the most significant properties of spectral graph theory is the fact that we can introduce graph Fourier transformation [6], [7], which is the diagonalization of the Laplacian matrix. The advantage of graph Fourier transformation can be found in its ability to decompose network dynamics into scales appropriate for the network's structure. As a result, complex network dynamics can be understood as the superposition of simple dynamics for each Fourier mode, and network dynamics can be completely understood algebraically.

However, the decomposition of dynamics into Fourier modes is effective only if the Laplacian matrix is symmetric. This is because symmetric matrices always can be diagonalized. User dynamics on a social or information networks is generated by the interaction of nodes on the networks. This interaction is generally asymmetric. In other words, the actions between nodes depend on the direction of links. To represent asymmetric actions on links, directed graphs are frequently used. Since the structure of a directed graph is normally expressed by an asymmetric matrix, graph Fourier transformation cannot be applied.

One proposal on spectral graph theory for directed graphs transforms asymmetric Laplacian matrixes in Jordan canonical form via elementary transformation [8], [9]. However, since asymmetric Laplacian matrices cannot always be diagonalized, decomposition of the dynamics into simple Fourier modes remains unavailable.

This paper focuses on some types of link asymmetry that can be represented as node characteristics, and represents the structure of a directed graph by a symmetric scaled Laplacian matrix. In addition, we analyze oscillation dynamics on networks to describe the propagation of activities on directed networks by using symmetric scaled Laplacian matrixes.

Typical examples of the asymmetric interaction of links include the relationship between a popular blogger and the followers. The strength of the interaction between them depends on the direction of links, and the strength of activity propagation on links is asymmetric. However, link directionality in this case can be reduced to node characteristics. Furthermore, since similar relations frequently appear in human relations, we expect that various asymmetric links on networks can be analyzed in terms of node characteristics. By using a symmetric matrix to model asymmetric links, we can apply graph Fourier transformation based on the symmetric scaled Laplacian matrix and thus analyze oscillation dynamics on asymmetric networks. Our framework adopts the mass of the node as the node characteristic.

In our model, oscillation dynamics on directed networks can be expressed by the equation of motion of the harmonic oscillator for each Fourier mode. Since the phase of the oscillation cannot be determined by the equation of motion, oscillation dynamics may exhibit complicated behavior that inhibits any intuitive understanding. Our solution is to use the oscillation energy of each node, a phase-free index, to

ISBN: 1-60132-434-0, CSREA Press ©

represent the strength of node activity. In simple cases, the oscillation energy can reproduce well-known node centralities of the degree centrality and the betweenness centrality. In general, the oscillation energy depends on the propagation attributes of network activity. Therefore, the oscillation energy is an extended notion of the well-known node centrality. So, we can expect that the proposed oscillation model is the underlying mechanism of activity propagation on networks.

Since the oscillation energy can be measured as the strength of node activity, the way of the usage of the measured value of energy is important for applications. We introduce models that describe the damped oscillation and the forced oscillation on networks. As an application, we propose a method for estimating the eigenvalues of the scaled Laplacian matrix; called the network resonance method. The network resonance method can estimate the eigenvalues by applying resonance of the forced oscillation on networks even if components of the scaled Laplacian matrix is not known.

This paper is organized as follows. In Section II, after defining the Laplacian matrix for directed networks, we introduce a scaled Laplacian matrix that allows asymmetric node interactions to be described by a symmetric matrix. In Sec. III, we analyze oscillation models to describe the propagation of node activity on networks by using the scaled Laplacian matrix. In Sec. IV, we propose the oscillation energy of each node as an extended index of node centrality and discuss the relationship to the well-known node centralities. In Sec. V, we propose the network resonance method to estimate eigenvalues of the scaled Laplacian matrix. Finally, we conclude this paper in Sec. VI.

II. Scaled Laplacian Matrix for Describing Asymmetric Link Direction

A. Definition of the Laplacian Matrix

Network structure is frequently expressed as a matrix. Let us consider loop-free directed graph $\mathcal{G}$ with n nodes. Let the set of nodes be $V = \{1, 2, \ldots, n\}$ and the set of directed links be E. In addition, let the link weight for link $(i \to j) \in E$ be $w_{ij} > 0$. We define the following $n \times n$ square matrix $\mathcal{A} = [\mathcal{A}_{ij}]$ as

$$\mathcal{A}_{ij} := \begin{cases} w_{ij} & ((i \to j) \in E), \\ 0 & ((i \to j) \notin E). \end{cases} \tag{1}$$

This matrix represents link presence and weights, and is called the (weighted) adjacency matrix.

Next, we define the weighted out-degree, d_i, of node i $(i = 1, 2, \ldots, n)$ as

$$d_i := \sum_{j \in \partial i} w_{ij}, \tag{2}$$

where ∂i denotes the set of nodes adjacent to node i. Also, weighted out-degree matrix $\mathcal{D}$ is defined as

$$\mathcal{D} := \mathrm{diag}(d_1, d_2, \ldots, d_n).$$

If all link weights are $w_{ij} = 1$ for $^{\forall}(i \to j) \in E$, d_i is reduced to out-degree, i.e. the number of outgoing links from node i.

Based on the above preparation, we define the Laplacian matrix $\mathcal{L}$ of directed graph $\mathcal{G}$ as follows [4], [5].

$$\mathcal{L} := \mathcal{D} - \mathcal{A}. \tag{3}$$

The Laplacian matrix is also called the graph Laplacian.

B. Symmetrization of Laplacian Matrix and the Scaled Laplacian Matrix

Let us consider left eigenvectors ${}^t\boldsymbol{m}$ and their eigenvalues λ as

$${}^t\boldsymbol{m}\,\mathcal{L} = \lambda\,{}^t\boldsymbol{m}. \tag{4}$$

If there is a (left) eigenvector ${}^t\boldsymbol{m} = (m_1, m_2, \ldots, m_n)$ associated with eigenvalue $\lambda = 0$, and $m_i > 0$ satisfies

$$m_i\, w_{ij} = m_j\, w_{ji} \quad (\equiv k_{ij}), \tag{5}$$

then the link asymmetry of $\mathcal{L}$ can be expressed by using a symmetric Laplacian matrix. Note that the oscillation dynamics discussed in the following sections satisfies these conditions. The procedure to represent $\mathcal{L}$ by a symmetric matrix is shown as follows. First, we consider a undirected graph and introduce its Laplacian matrix L as $L := D - A$, where $A = [A_{ij}]$ is defined as

$$A_{ij} := \begin{cases} k_{ij} & ((i,j) \in E), \\ 0 & ((i,j) \notin E), \end{cases} \tag{6}$$

and $D = \mathrm{diag}(\sum_j A_{1j}, \sum_j A_{2j}, \ldots, \sum_j A_{nj})$. Since $k_{ij} = k_{ji}$, L is a symmetric Laplacian matrix for a certain undirected graph. By using L, the asymmetric Laplacian matrix $\mathcal{L}$ is expressed as

$$\mathcal{L} = M^{-1}\, L, \tag{7}$$

where $M := \mathrm{diag}(m_1, m_2, \ldots, m_n)$ means the scaling factors of nodes. Figure 1 shows a simple example of the above procedure: where $w_{ij} = k_{ij}/m_i$ is decomposed into $1/m_i$ and k_{ij}.

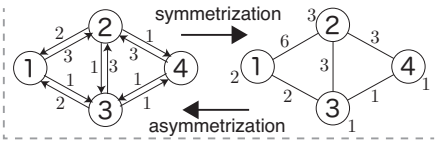
Here, we introduce the scaled Laplacian matrix that is defined as

$$S := M^{-1/2}\, L\, M^{-1/2}. \tag{8}$$

Note that S is a symmetric matrix. Let $\boldsymbol{x} = {}^t(x_1, x_2, \ldots, x_n)$ be a (right) eigenvector associated with an eigenvalue λ, that is, $\mathcal{L}\,\boldsymbol{x} = \lambda\,\boldsymbol{x}$. By multiplying $M^{+1/2}$ to the eigenvalue equation from the right, we obtain

$$M^{+1/2}\,\mathcal{L}\,\boldsymbol{x} = S\,(M^{+1/2}\,\boldsymbol{x}) = \lambda\,(M^{+1/2}\,\boldsymbol{x}). \tag{9}$$

$$\mathcal{L} = \begin{bmatrix} 4 & -3 & -1 & 0 \\ -2 & 4 & -1 & -1 \\ -2 & -3 & 6 & -1 \\ 0 & -3 & -1 & 4 \end{bmatrix}$$



$$= \begin{bmatrix} 4 & -6/2 & -2/2 & 0 \\ -6/3 & 4 & -3/3 & -3/3 \\ -2/1 & -3/1 & 6 & -1/1 \\ 0 & -3/1 & -1/1 & 4 \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ 0 & 1/3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 8 & -6 & -2 & 0 \\ -6 & 12 & -3 & -3 \\ -2 & -3 & 6 & -1 \\ 0 & -3 & -1 & 4 \end{bmatrix}$$

Fig. 1. An example of the Laplacian matrix for a directed graph and its symmetrization.

This means the scaled Laplacian matrix S has the same eigenvalues of $\mathcal{L}$, and its eigenvector is $\boldsymbol{y} := M^{+1/2}\,\boldsymbol{x}$. Since the quadratic form of S is

$$ {}^t\boldsymbol{y}\,S\,\boldsymbol{y} = \sum_{(i,j)\in E} k_{ij}\left(\frac{y_i}{m_i} - \frac{y_j}{\sqrt{m_i m_j}}\right)^2 \geq 0, $$

the eigenvalues of S (also $\mathcal{L}$) are nonnegative. Let us sort the eigenvalues in ascending order,

$$ 0 = \lambda_0 \leq \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_{n-1}. $$

We can choose eigenvector $\boldsymbol{v}_\mu$ $(\mu = 0, 1, \ldots, n-1)$ as the orthonormal eigenvector associated with λ_μ. That is,

$$ S\,\boldsymbol{v}_\mu = \lambda_\mu\,\boldsymbol{v}_\mu, \quad \boldsymbol{v}_\mu \cdot \boldsymbol{v}_\nu = \delta_{\mu\nu}, \tag{10} $$

where $\delta_{\mu\nu}$ denotes the Kronecker delta.

III. Oscillation Models on Networks

A. Oscillation Model Based on Asymmetric Interactions

To describe the propagation of activity of a node through networks, let us consider oscillation dynamics on networks. The relationship between the oscillating phenomena and well-known indices for network dynamics is be discussed in Sec. IV.

Let weight x_i of node i be displacement from the equilibrium, and let its restoring force be proportional to the difference in the displacements of adjacent nodes. Figure 2 is a representative image of our oscillation model. Although the figure shows a 1-dimensional network, it is easily extended to general networks. To represent diverse oscillating behavior, we allow the spring constant of each link to be different and the mass of each node to also be different.

Here, it is worthy to note about the validity of oscillation model whose restoring force is proportional to the difference of displacements. Let the restoring force of node i be a function $f(\Delta x)$ of the difference $\Delta x := x_i - x_j$ of the displacements of adjacent nodes i and j. It is natural to assume $f(\Delta x) = 0$ if $\Delta x = 0$. For small Δx, we can expand $f(\Delta x)$ as

$$ f(\Delta x) = -k_{ij}\,\Delta x + O(\Delta x^2), $$

where k_{ij} is a positive constant corresponding to the spring constant. So, our oscillation model can be considered as the basic and universal model if nonlinear effects in $O(\Delta x^2)$ are relatively small.

Incidentally, there is a well-known oscillation model, called the Kuramoto model (Fig. 3)[10]. This model consists of the same (or similar) oscillators coupled by weak interaction, and mainly describes the synchronization of these oscillators. Thus our oscillation model differs from Kuramoto model.

We assign a spring constant to each link and express it as link weight $k_{ij} > 0$. In addition, we assign mass $m_i > 0$ to each node i. Let x_i be the displacement of node i and p_i be its conjugate momentum. Then, Hamiltonian $\mathcal{H}$ of our coupled oscillator system is expressed as

$$ \begin{aligned} \mathcal{H} &:= \sum_{i\in V} \frac{(p_i)^2}{2\,m_i} + \sum_{(i,j)\in E} \frac{k_{ij}}{2}\,(x_i - x_j)^2 \\ &= \sum_{i\in V} \frac{(p_i)^2}{2\,m_i} + \frac{1}{2}\,({}^t\boldsymbol{x}\,L\,\boldsymbol{x}). \end{aligned} $$

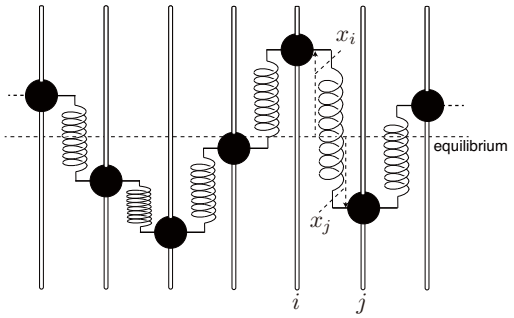


Fig. 2. Oscillation model on networks.

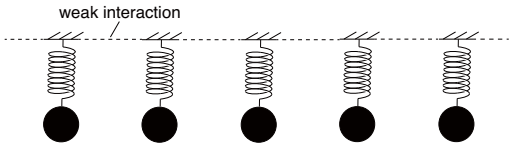


Fig. 3. Kuramoto model.

By applying canonical formalism, the equations of motion are derived as follows.

$$ \frac{\mathrm{d}p_i}{\mathrm{d}t} = -\frac{\partial \mathcal{H}}{\partial x_i} = -\sum_{j=1}^{n} L_{ij}\,x_j, \quad \frac{\mathrm{d}x_i}{\mathrm{d}t} = \frac{\partial \mathcal{H}}{\partial p_i} = \frac{p_i}{m_i}. $$

By eliminating p_i from these equations, we have the following wave equation as the equation of motion,

$$ m_i\,\frac{\mathrm{d}^2 x_i}{\mathrm{d}t^2} = -\sum_{j=1}^{n} L_{ij}\,x_j, $$

or written in vector form as

$$ M\,\frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2} = -L\,\boldsymbol{x}, \tag{11} $$

where $M := \mathrm{diag}(m_1, \ldots, m_n)$ and $\boldsymbol{x} := {}^t(x_1, \ldots, x_n)$. By multiplying M^{-1} from the left, we have the equation of motion as

$$ \frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2} = -M^{-1}\,L\,\boldsymbol{x} = -\mathcal{L}\,\boldsymbol{x}; \tag{12} $$

note that it is based on asymmetric interactions. To diagonalize the equation of motion, we introduce vector $\boldsymbol{y}$ which is defined by

$$ \boldsymbol{y} = M^{1/2}\,\boldsymbol{x}, $$

and the equation of motion can be written as

$$ \frac{\mathrm{d}^2\boldsymbol{y}}{\mathrm{d}t^2} = -S\,\boldsymbol{y}. \tag{13} $$

It follows that the equation of motion will yield the eigenvalue problem of the symmetric scaled Laplacian matrix, and node mass can be understood as the node scaling factor. Node mass can represent the strength of inertia, and is related to the strength of the asymmetric influence to adjacent nodes. In addition, the spring constant of links can represent the strength of influence between each pair of adjacent nodes. Furthermore, the condition (5) corresponds to Newton's 3rd

law (about equivalency of the strength between an action and its reaction).

Let $\boldsymbol{y} = \boldsymbol{y}(t)$ be expanded by the eigenbasis of S, $\boldsymbol{v}_\mu$, as $\boldsymbol{y}(t) = \sum_{\mu=0}^{n-1} a_\mu(t)\,\boldsymbol{v}_\mu$ and solve the equation of motion for the Fourier mode $a_\mu(t)$ $(\mu = 0, 1, \ldots, n-1)$. The procedure of expansion by eigenbasis is known as graph Fourier transformation [6], [7]. The solution is given by

$$a_\mu(t) = c_\mu\, e^{\mathrm{i}\,(\omega_\mu\, t + \theta_\mu)}, \tag{14}$$

where $\omega_\mu^2 = \lambda_\mu$, $\mathrm{i} = \sqrt{-1}$, θ_μ denotes phase, and c_μ is a constant. The solution of oscillation on networks (the solution of (12)) is expressed as

$$\boldsymbol{x}(t) = M^{-1/2} \left(\sum_{\mu=0}^{n-1} c_\mu\, e^{\mathrm{i}\,(\omega_\mu\, t + \theta_\mu)}\, \boldsymbol{v}_\mu \right). \tag{15}$$

Note that the phase cannot be determined by the equation of motion, but the oscillation behavior varies widely with the phase. Consequently, to understand the universal aspect of oscillation dynamics, a kind of phase-free index is required. This issue is discussed in Sec. IV.

B. Damped Oscillation Model

In actual situations, any oscillation is damped with time. This subsection shows a model for the damped oscillation on networks.

Let us consider the equation of motion for the damped oscillation

$$M\,\frac{\mathrm{d}^2\boldsymbol{x}(t)}{\mathrm{d}t^2} + \gamma\, M\,\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = -L\,\boldsymbol{x}(t), \tag{16}$$

where γ is a constant. Here $\gamma\, M$ means the viscous damping coefficient, where it is important to note that the viscous damping coefficient is assumed to be proportional to node mass. By using vector $\boldsymbol{y} = M^{1/2}\,\boldsymbol{x}$, we can diagonalize the equation of motion as

$$\frac{\mathrm{d}^2\boldsymbol{y}(t)}{\mathrm{d}t^2} + \gamma\,\frac{\mathrm{d}\boldsymbol{y}(t)}{\mathrm{d}t} = -S\,\boldsymbol{y}(t).$$

The equation of motion for Fourier mode $a_\mu(t)$ is expressed as

$$\frac{\mathrm{d}^2 a_\mu(t)}{\mathrm{d}t^2} + \gamma\,\frac{\mathrm{d}a_\mu(t)}{\mathrm{d}t} + \omega_\mu^2\, a_\mu(t) = 0, \tag{17}$$

where $\omega_\mu^2 = \lambda_\mu$. To analyze the solution of this equation, we assume the solution takes the form of $a_\mu(t) \propto e^{\alpha t}$. By substituting this into the equation of motion, we obtain the characteristic equation

$$\alpha^2 + \gamma\alpha + \omega_\mu^2 = 0. \tag{18}$$

There are three different solutions to the equation of motion according to the solution of the characteristic equation, $\alpha = -(\gamma/2) \pm \sqrt{(\gamma/2)^2 - \omega_\mu^2}$. In the case of $(\gamma/2)^2 < \omega_\mu^2$, the solution describes damped oscillations,

$$a_\mu(t) = c_\mu\, e^{-(\gamma/2)t}\, \cos\left[\sqrt{\omega_\mu^2 - (\gamma/2)^2}\, t + \theta_\mu\right], \tag{19}$$

where c_μ and θ_μ are constants. In the case of $(\gamma/2)^2 = \omega_\mu^2$, the solution describes critical damping,

$$a_\mu(t) = (a_\mu(0) + c_\mu\, t)\, e^{-(\gamma/2)\,t}, \tag{20}$$

where c_μ is a constant. Finally, in the case of $(\gamma/2)^2 > \omega_\mu^2$, the solution describes overdamping. Let α_+ and α_- (both values are negative) denote the solutions of the characteristic equation, the solution of the equation of motion is

$$a_\mu(t) = c_\mu^+\, e^{\alpha_+ t} + c_\mu^-\, e^{\alpha_- t}, \tag{21}$$

where c_μ^+ and c_μ^- are constants.

C. Forced Oscillation Model

This subsection introduces a forced oscillation model on networks. Let us consider the situation that we impose forced oscillation with angular frequency ω on a certain node, j, as an external force. The equation of motion of the forced oscillation is

$$M\,\frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2} + \gamma\, M\,\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} + L\,\boldsymbol{x}(t) = (F\,\cos\omega t)\,\mathbf{1}_{\{j\}}, \tag{22}$$

where F is a constant and $\mathbf{1}_{\{j\}}$ is only the j-th component that is 1, all other components are 0, that is,

$$\mathbf{1}_{\{j\}} = {}^t(0, \ldots, 0, \overset{j}{\overset{\vee}{1}}, 0, \ldots, 0).$$

By using vector $\boldsymbol{y} = M^{1/2}\,\boldsymbol{x}$, the equation of motion can be diagonalized as

$$\frac{\mathrm{d}^2\boldsymbol{y}(t)}{\mathrm{d}t^2} + \gamma\,\frac{\mathrm{d}\boldsymbol{y}(t)}{\mathrm{d}t} + S\,\boldsymbol{y}(t) = \frac{F\,\cos\omega t}{\sqrt{m_j}}\,\mathbf{1}_{\{j\}}. \tag{23}$$

Since $\boldsymbol{y}(t)$ depends on ω, we redefine $\boldsymbol{y}(\omega, t) := \boldsymbol{y}(t)$. By expanding $\boldsymbol{y}(\omega, t)$ and $\mathbf{1}_{\{j\}}$ using the eigenbasis of the scaled Laplacian matrix S, we introduce the Fourier modes $a_\mu(\omega, t)$ and b_μ as

$$\boldsymbol{y}(\omega, t) = \sum_{\mu=0}^{n-1} a_\mu(\omega, t)\,\boldsymbol{v}_\mu, \quad \mathbf{1}_{\{j\}} = \sum_{\mu=0}^{n-1} b_\mu\,\boldsymbol{v}_\mu. \tag{24}$$

The equation of motion of Fourier mode $a_\mu(\omega, t)$ is expressed as

$$\frac{\partial^2 a_\mu(\omega, t)}{\partial t^2} + \gamma\,\frac{\partial a_\mu(\omega, t)}{\partial t} + \omega_\mu^2\, a_\mu(\omega, t) = \frac{F\,\cos\omega t}{\sqrt{m_j}}\, b_\mu \tag{25}$$

The solution of the inhomogeneous equation (25) is the sum of the solutions of the corresponding homogeneous equation (17) and the particular solution of (25). Since the solution of homogeneous equation (17) is dampened with time, only the oscillation of the particular solution of (25) remains after some long time. Since the angular frequency of the particular solution should be ω, the particular solution can be expressed as

$$\begin{aligned} a_\mu(\omega, t) &= A_\mu(\omega)\cos(\omega t + \theta_\mu) \\ &= A_\mu(\omega)(\cos\omega t\,\cos\theta_\mu - \sin\omega t\,\sin\theta_\mu). \end{aligned} \tag{26}$$

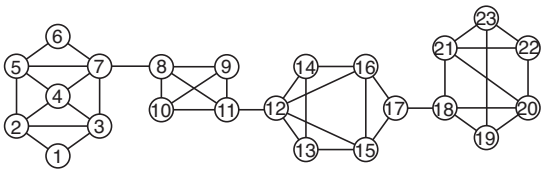


Fig. 4. Network model.

By substituting it into the equation of motion (25), the amplitude $A_\mu(\omega)$ and phase θ_μ of the particular solution are obtained as

$$A_\mu(\omega) = \frac{F\, b_\mu}{\sqrt{m_j}} \frac{1}{\sqrt{(\omega_\mu^2 - \omega^2)^2 + (\gamma\,\omega)^2}},\ \tan\theta_\mu = \frac{-\gamma\,\omega}{\omega_\mu^2 - \omega^2}. \tag{27}$$

IV. Node Centrality

As shown in Sec. III-A, the wave equation (12) cannot describe the phase of oscillations. Since the behavior of oscillating phenomenon has extremely different appearance if the phase changes, it is hard to extract useful information from direct observation of oscillating aspects. Of course, since $a_\mu(t)$ of (14) is a complex-valued function, the value of $a_\mu(t)$ cannot be observed in actual networks. This section introduces the oscillation energy of each node as a non-negative-valued phase-free index, and shows that it can reproduce the well-known indices of node centrality. This means that our oscillation model can be considered as an underlying mechanism of the propagation of activities on networks.

For the oscillation model described in Sec. III-A, we define node activity as the oscillation energy of the node. From (14), the amplitude of the Fourier mode $a_\mu(t)$ is $c_\mu = |a_\mu(t)|$. In addition, let $\boldsymbol{v}_\mu$ be the eigenbasis associated with the eigenvalue λ_μ of scaled Laplacian matrix S, and let its components be expressed as

$$\boldsymbol{v}_\mu = (v_\mu(1),\, v_\mu(2),\, \ldots,\, v_\mu(n)).$$

Since the oscillation of node i is the superposed oscillations for Fourier modes of the node, the oscillation energy E_i of node i is obtained by summing the oscillation energy for each Fourier mode, as

$$E_i = \frac{1}{2}\sum_{\mu=0}^{n-1} \omega_\mu^2\,(c_\mu\, v_\mu(i))^2.$$

To demonstrate the calculation of the oscillation energy of each node, we use the network model shown in Fig. 4, where all the link weights are set at 1. As the initial condition of the wave equation (12), we can give the displacement only at a certain node. We call the node as a source node of activity. First of all, let us consider the situation that the source node of activity is chosen at random. In this case, all the Fourier modes contribute at the same strength. Figures 5 (a) and (b) are the results for evaluation of oscillation energy for each node, for two different scaling factors (node mass): $M = I$ (the unit matrix) and $M = D^2$, respectively. The condition $M = I$ means that the strength of interaction between nodes is symmetric, and the condition $M = D^2$ gives an example of asymmetric node interaction. From Fig. 5 (a), we can recognize that the oscillation energy is proportional to the node degree centrality (the oscillation energy for each node is proportional to its node degree). So, Fig. 5 (b) can be regarded as an extension of node degree centrality considering asymmetry of node interaction.

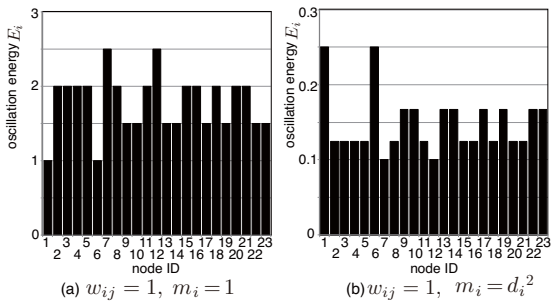


Fig. 5. Oscillation energy for each node for the case that the source node is at random.

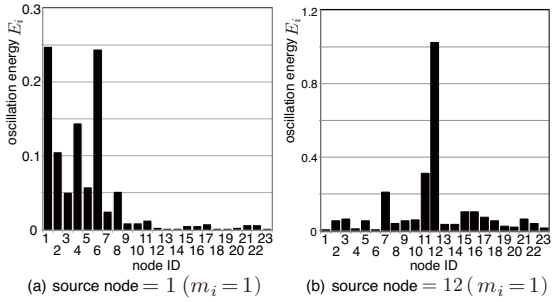


Fig. 6. Oscillation energy for each node for a specific source node.

If a certain specific node is the source of activity, node oscillation energy would be quite different. Figures 6 (a) and (b) show the oscillation energy of each node for different source nodes, 1 and 12, respectively, where the scaling factors is chosen as $M = I$. The results show that the oscillation energy strongly depends on the source node of activity. Therefore, the oscillation energy is changed not only by network topology, but also node mass (Fig. 5) and the propagation scenario of activity on networks (Fig. 6). In other words, the oscillation energy also depends on link asymmetry, and strength and location distributions of source nodes. Since the oscillation energy is reduced to the well-known degree node centrality in the simplest case, the oscillation energy for each node can be understood as an extended notion of the degree centrality.

The betweenness centrality is another well-known node centrality. Let the number of shortest paths between node j and node k be σ_{jk}, and the number of those paths passing through the node i be $\sigma_{jk}(i)$. The betweenness centrality $g(i)$ for node i is defined as

$$g(i) := \sum_{j,\,k\in V\setminus\{i\}} \frac{\sigma_{jk}(i)}{\sigma_{jk}}.$$

The normalized betweenness centrality $\bar{g}(i)$ is defined as

$$\bar{g}(i) := \frac{2\,g(i)}{(n-1)\,(n-2)}.$$

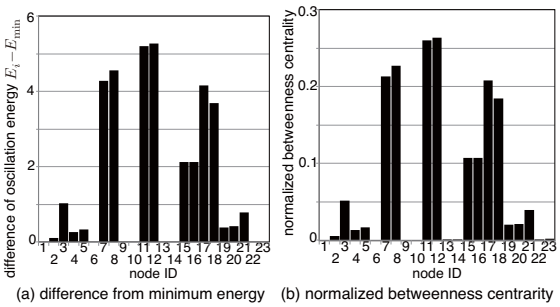


Fig. 7. Relationship between the difference of oscillation energy and the betweenness centrality.

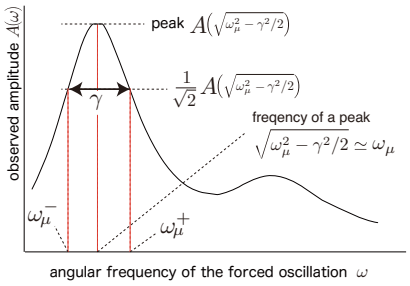


Fig. 8. Concept of network resonance.

The physical meaning of $\bar{g}(i)$ is the ratio of the number of shortest paths including node i to the number of combination of node pairs in $V \backslash \{i\}$, that is $(n-1)(n-2)/2$.

Next, we set the link weight k_{ij} of the network model shown in Fig. 4 as the number of the shortest paths passing through the link (i, j). Figure 7 (a) shows the difference between the oscillation energy E_i for each node and the minimum energy $E_{\min}$ defined as

$$E_{\min} := \min_{i \in V} E_i.$$

Figure 7 (b) shows the normalized betweenness centrality $\bar{g}(i)$ for each node. We can recognized that the difference between the oscillation energy is proportional to the betweenness centrality. From the same reason for degree centrality, the oscillation energy E_i gives an extension of the well-known betweenness centrality.

The oscillation energy gives extensions of node centralities even if we consider the damped oscillation on networks. Detailed discussion is presented in [11].

V. Network Resonance Method for Investigating the Eigenvalues of Network Dynamics

Since the actual structure of a network is difficult to know, it is almost impossible to measure components of the scaled Laplacian matrix S, directly. For example, in social networks, the strength and significance of friendships (links) are hard to observe. Thus the eigenvalues of S, the key to describing the oscillation dynamics on networks, cannot be calculated from S. However, since the oscillation energy is related to the node centrality that is the strength of activity of node on networks, we probably be able to measure the oscillation energy as a real solid object. The oscillation energy is related to the natural frequency and the amplitude. In this section, we discuss a way to estimate natural frequency (square root of eigenvalue) of S from observation of the amplitude that is obtained from observation of the oscillation energy.

As recognized from discussion in Sec. III-C, amplitude $A_\mu(\omega)$ of (27) takes maximal value at

$$\omega = \sqrt{\omega_\mu^2 - \gamma^2/2}.$$

This phenomenon is called the resonance. When we observe the oscillation of a node caused by forced oscillation, the mixture of oscillation (26) for each μ, that is, $\boldsymbol{y}(\omega, t)$ of the first equation of (24) is observed. We propose a method to estimate eigenvalue λ_μ (or $\omega_\mu = \sqrt{\lambda_\mu}$) and damping factor γ from observations of the amplitude $A(\omega) := |\boldsymbol{y}(\omega, t)|$ of the response oscillation (Fig. 8). In actual, the amplitude $A(\omega)$ is indirectly obtained from observations of oscillation energy.

The Q-factor represents the sharpness of amplitude $A_\mu(\omega)$ with respect to ω. On both sides of the peak of amplitude $A_\mu(\omega)$, we define frequencies ω_μ^+ and ω_μ^- that give the amplitudes $A_\mu(\omega_\mu^+)$ and $A_\mu(\omega_\mu^-)$ that are $1/\sqrt{2}$ times the peak value of $A_\mu(\omega)$ ($\omega_\mu^+ > \omega_\mu^-$). Since oscillation energy is proportional to the square of the amplitude, $\omega_\mu^+ - \omega_\mu^-$ means the half width for energy. The Q-factor is defined as

$$Q_\mu := \frac{\sqrt{\omega_\mu^2 - \gamma^2/2}}{\omega_\mu^+ - \omega_\mu^-}.$$

We assume $\gamma \ll \omega_\mu$ and approximate $A_\mu(\omega)$ around $\omega = \omega_\mu$. By using $\omega^2 - \omega_\mu^2 \simeq 2\,\omega_\mu\,(\omega - \omega_\mu)$,

$$\begin{aligned} A_\mu(\omega) &\simeq \frac{F\,b_\mu}{\sqrt{m_j}} \frac{1}{\sqrt{(2\,\omega_\mu\,(\omega - \omega_\mu))^2 + (\gamma\,\omega_\mu)^2}} \\ &= \frac{F\,b_\mu}{\sqrt{m_j}\,\omega_\mu} \frac{1}{\sqrt{4\,(\omega - \omega_\mu)^2 + \gamma^2}}. \end{aligned} \tag{28}$$

Therefore,

$$A_\mu(\omega_\mu) \simeq \frac{F\,b_\mu}{\sqrt{m_j}\,\omega_\mu\,\gamma}, \quad A_\mu(\omega_\mu \pm \gamma/2) \simeq \frac{1}{\sqrt{2}}\,A_\mu(\omega_\mu),$$

and we have $\omega_\mu^\pm = \omega_\mu \pm \gamma/2$ (double-sign indicates correspondance). Consequently, we have

$$Q_\mu \simeq \frac{\omega_\mu}{\gamma}. \tag{29}$$

These relations enable us to estimate natural frequency ω_μ (or the eigenvalue $\lambda_\mu = \omega_\mu^2$) and damping factor γ.

We use the network model shown in Fig. 4, where all link weights are 1 and node mass is also set to $M = I$. Figures 9 (a) and (b) show examples of network resonance for external force input by node 1 and 12, respectively: the amplitude $A(\omega) = |\boldsymbol{y}(\omega, t)|$ is observed at node 1 (red line) and node 10 (blue line) as the response of the external force with angular frequency ω. Depending on the pair of input node and observed node selected, the amplitude $A(\omega)$ exhibits a different aspect. Therefore, we expect that eigenvalues of the scaled Laplacian matrix can be estimated from appropriate pairs of input and observed nodes.

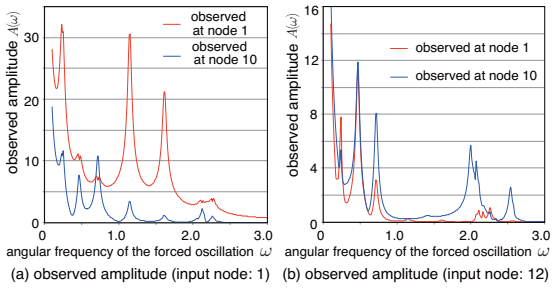


Fig. 9. Examples of network resonance: the amplitude of oscillation as a response to input oscillation with ω.

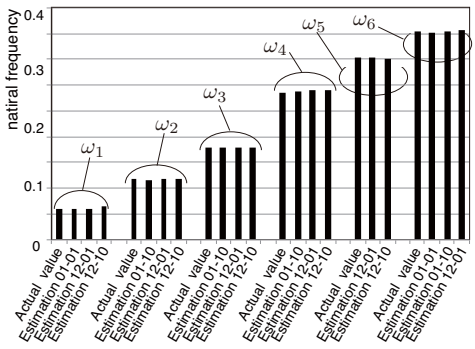


Fig. 10. Estimation of the natural frequency ω_μs.

Figure 10 compares the actual values of natural frequencies $\omega_1, \omega_2, \ldots, \omega_6$ and their estimated values obtained from Fig. 9 (a) and (b). For example, "Estimation 01-10" in Fig. 10 means the input node is 1 and the observed node is 10. The estimated natural frequencies are close to the actual values. Depending on the positions of both the input node of forced oscillation and the observed node, there are natural frequencies that cannot be observed. For example, the values of ω_2, ω_3, ω_4 and ω_5 cannot be estimated from the node pair of input node 1 and observed node 1. Selecting the appropriate pair of input and observed nodes avoids this problem.

VI. Conclusions

This paper showed how to use the scaled symmetric Laplacian matrix to model oscillation dynamics on networks caused by a certain kind of asymmetric interaction between nodes. Although our asymmetric node interactions are restricted to models that are characterized by the intrinsic property (mass) of the nodes themselves, these interactions are common in actual networks (e.g., the relations between a popular blogger and his/her followers).

Although solutions of the proposed oscillation model are complex numbers and they cannot be observed directly, the oscillation energy can reproduce well-known node centrality on the networks. In addition, the oscillation energy is an extended notion of node centrality reflecting propagation scenarios of activity on networks, and we expect that the proposed oscillation model can be considered as the underlying mechanism of activity propagation on networks.

Since the oscillation energy is expected to be observed via measurement of the strength of node activity, the framework of usage of the measured value of energy is important. We proposed a network resonance method that can estimate the eigenvalues of the scaled Laplacian matrix and the damping factor, from measurements.

We also expect that this method can estimate the absolute value of the component of eigenvectors. If the sign of the components are determined by orthogonal condition of eigenvectors, we obtain pairs of eigenvalues and eigenvectors of the scaled Laplacian matrix, from measurements. This means the original Laplacian matrix (including the weight of directed link and network topology) can be reproduced. So, our framework is applicable to investigate network structure that is not observed directly; for example, social networks of users, networks of malicious hosts generating cyber attacks, etc.

In security application, we probably can use the framework of [12], for example. First, we access malicious web site with the frequency of ω. These accesses induce that malicious users attack to a honeypot, and we observe their response. The framework corresponds to the network resonance method based on forced oscillation.

Acknowledgment

The authors would like to thank Mr. Satoshi Furutani of Tokyo Metropolitan University for his help with the numerical experiments. This research was supported by Grant-in-Aid for Scientific Research (B) No. 26280032 from JSPS.

References

[1] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*, Cambridge University Press, 1994.

[2] P.J. Carrington, J. Scott and S. Wasserman, *Models and Methods in Social Network Analysis*, Cambridge University Press, 2005.

[3] A. Mislove, M. Marcon, K.P. Gummadi, P. Druschel and B. Bhattacharjee, "Measurement and analysis of online social networks," Proc. of *ACM SIGCOMM conference on Internet measurement*, pp. 29–42, 2007.

[4] D. Spielman, "Spectral graph theory," Chapter 18 of *Combinatorial Scientific Computing* (Eds. U. Naumann & O. Schenk), pp. 495–524, Chapman and Hall/CRC, 2012.

[5] M.E.J. Newman, "The graph Laplacian," Section 6.13 of *Networks: An Introduction*, pp. 152–157, Oxford University Press, 2010.

[6] D.K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory," Applied and Computational Harmonic Analysis, vol. 30, no. 2, pp. 129–150, 2011.

[7] D.I. Shuman, S.K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains," IEEE Signal Process. Mag., vol. 30, no. 3, pp. 83–98, 2013.

[8] A. Sandryhaila and J.M.F. Moura, "Discrete signal processing on graphs," IEEE Trans. Signal Process., vol. 61, no. 7, pp. 1644–1656, 2013.

[9] A. Sandryhaila and J.M.F. Moura, "Big data analysis with signal processing on graphs: Representation and processing of massive data sets with irregular structure," IEEE Signal Process. Mag., vol. 31, no. 5, pp. 80–90, 2014.

[10] Y. Kuramoto, *Chemical Oscillations, Waves, and Turbulence*, Dover Books on Chemistry, 2003.

[11] C. Takano and M. Aida, "Proposal of new index for describing node centralities based on oscillation dynamics on networks," *IEICE Tech. Rep.*, CQ2015-123, 2016. (in Japanese)

[12] M. Akiyama, T. Yagi, Y. Kadobayashi, T. Hariu and S. Yamaguchi, "Client honeypot multiplication with high performance and precise detection," *IEICE Trans. Inf&Syst.*, vol. E98-D, no. 4, pp. 775–787, 2015.

SESSION

ALGORITHM EFFICIENCY STUDIES AND PERFORMANCE RELATED ISSUES

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

Assimilated Deliberation of Spatial and Temporal Complexities of Computational Algorithms

A. Tarek[1], **and A. Farhan**[2]
[1]Engineering, Physical and Computer Sciences, Montgomery College, Rockville, Maryland, United States of America
[2]College of Arts and Sciences, University of Pennsylvania, Philadelphia, Pennsylvania, United States of America

Abstract— *Computational complexity is an important epitome in computing research. However, a vast majority of the related results involving temporal and spatial complexities are purely of theoretical interest. Only a few have treated the model of complexity as applied to real-life computation. Besides, some of these results would be hard to realize in practice. A majority of these papers lack in simultaneous contemplation of temporal and spatial complexities, which collectively depict the overall computational scenario. This paper surpasses some of these limitations through a clearly depicted model of computation and a meticulous analysis of spatial and temporal complexities. The paper deliberates computational complexities due to a wide variety of coding constructs arising frequently in practice. Moreover, a structured, formal approach to temporal and spatial complexities treated harmoniously as applied to real computation is also explored. There is a prevailing muddle as per the big_oh complexity is concerned. Common trend is to consider it as a function rather than as a set of functions. This perplexity is clarified with illustration. Disparate recursive data structures are contemplated for spatial complexity due to their vital role in computing. Experimental results acquired through physical measurements are also accentuated. At length, both temporal and spatial complexities are deliberated in a single scaffold, and the concept relating to space-time bandwidth product is introduced, and significance revealed. The space-time bandwidth products of common sorting algorithms are contemplated.*

Keywords: Big_oh complexity, Set of functions, Sorting algorithms, Space-time bandwidth product, Spatial complexity, Temporal complexity

1. Introduction

Computation time and memory space requirements are two major constraints in computer implementation of real-life algorithms. With a wide variety of formal notation for expressing these computational constraints, big-oh is the most commonly used. This paper primarily focuses on upper-bounds as expressed through big-oh notation for temporal and spatial complexities.

Temporal complexity is the CPU time necessitated by an algorithm for its computer implementation. Spatial complexity is the number of memory cells that an algorithm truly requires for computation. A good algorithm tends to keep both of these requirements as small as possible. Computer memory is a *re-usable resource* from the operating system standpoint and may be released for further reallocation. Temporal resources are *consumable*, and once spent, there is no return to that point in time. Though there is a significant difference between temporal and spatial complexities from reallocation standpoint, spatial complexity shares many of the same features due to temporal complexity.

To express the upper bound in computational resource requirements, big_oh (O) notation is used. For expressing the lower bound in computational resources, Ω notation is adopted. To express both of these resource constraints in a single framework, the Θ notation is prevalent. Among all three different notations, big_oh (O) complexity is the most prominent one. Often it becomes necessary to have an estimate on upper-bound of the computational resource requirements. Therefore, the focus of this paper is on big_oh notational temporal and spatial complexities.

In Section 2, specific terms and notations used in this paper are discussed briefly. Section 3 deals with a variety of coding constructs frequently arising in realistic computation and provides big-oh time complexity for each. Section 4 considers spatial complexity. Section 5 explores realistic issues in temporal and spatial complexity models discussed in this paper. Section 6 is the conclusion based on models of analysis in the paper.

2. Terminology and Notations

In this paper, following notations are used.
n: Input or instance size.
$g(n)$: Highest order term in complexity expression without coefficients.
$f(n)$: Complexity function with an input size, n.
$\acute{f}(n)$: Complexity function without constant coefficients.
$O(g(n))$: Big-oh complexity with problem size, n, representing a set corresponding to the complexity class, $g(n)$.
$T(n)$: Temporal complexity of an algorithm or a function with an input size n.
$S(n)$: Spatial complexity of an algorithm or a function with an input of size, n.

$B(n)$: Space-time Bandwidth Product for input size n.
C: Any constant value.
$DSPACE(f,n)$: $DSPACE$ stands for the Deterministic Space Computation. Therefore, $DSPACE(f,\ n)$ denotes the total number of memory cells used during deterministic computation of $f(n)$. Here, f indicates the algorithm or function under consideration, and n is the input of size, $|n|$. It is often abbreviated as $DSPACE\ (f)$. Hence, $DSPACE(f,\ n) \in O(g(n))$. However, $DSPACE\ (f)$ is not defined if the computation of $f(n)$ does not halt.

Please refer to [2] and [4] for the definition of big-oh complexity. Following forms the basis of Complexity Order using big-oh, and provides the *complexity class hierarchy*.
$log_2 n < n < nlog_2 n < n^2 < \ldots < n^k < 2^n < C^n < n!$

3. Temporal Complexity of Algorithms

In determining temporal complexity, there are: *operations count* and *step count*. Operations count is the number of additions, multiplications, comparisons and other operations used during computation. Success in operations count depends on the ability to identify crucial operations that contribute most to temporal complexity. Step count accounts for all time spent in all parts of the program/function.

Big-oh complexity is usually expressed by the *fastest growing term* in the complexity function. There are 4 algorithmic steps in determining the big-oh complexity, which may be implemented as a computer program. The algorithm is described below:

Algorithm big_ohComplexity(n)
Purpose: This algorithm determines big_oh complexity with instance size n (single instance variable).
Input: Complexity function, $f(n)$ on n with k terms.
Output: Big_oh complexity, $O(g(n))$.
int[] $arr1$ = new $int[k]$ {array, $arr1$ holds the power of n for each term in the k-term complexity function, $f(n)$.}
int $highest\_power$ = $arr1[0]$ {Because the terms are not ordered, determine the true highest order term in $f(n)$.}
for j=1 to $k-1$ **do**
  if $arr1[j] > highest\_power$ **then**
    $highest\_power$ = $arr1[j]$
  end if
end for
g_n = $power(n,\ highest\_power)$
Print $O(g\_n)$ as the big_oh complexity.

Above algorithm is general, and may be extended to complexity functions involving multiple instance characteristics.

Example 1: Consider the complexity function, $f(n,m) = (h(n,m)^2 + 4n - 3m + 2)$ on n and m. Function, $h(n,m) = 2n^2 + m$. Here, $h(n,m)$ is nested within function, $f(n,m)$. Hence, $f(n,m) = (2n^2 + m)^2 + 4n - 3m + 2 = 4n^4 + m^2 + 4n^2m + 4n - 3m + 2$. Removing constant coefficients, $f(n,m) = (n^4 + m^2 + n^2m + n + m + 1)$. As the order of n^2m is, 3, the highest order term without coefficients is, $g(n,m) = n^4$, and $f(n,m) \in O(n^4)$. This Upper-bound is independent of m. Examples are the Graph algorithms, where n corresponds to the set of vertices, and m represents the set of edges.

3.1 Big_oh Complexity As a Set

The big_oh complexity as denoted by O is a set rather than a single function. However, it is a common practice in the prevailing literature to use notions, such as $f(n) = O(g(n))$ [8]. A function cannot be equal to a set. In fact, $O(g(n))$ is a set of functions that incorporates all functions in the order of $g(n)$ as well as any lower order function. For instance, consider $f_1(n) = 2n^2 + 3n + 5$ and $f_2(n) = 7n^2 + 9$. If $g(n) = n^2$, then the set $O(g(n)) = \{f_1(n),\ f_2(n),\ \ldots\}$, which incorporates any function in the order of n^2 and any lower order function. Therefore, $f(n) \in O(g(n))$.

Another common practice is to use expression, such as $f(n) = h(n) + O(g(n))$ [8]. However, a function may not be added to a set. With sets, valid operations are set union, set intersection, etc. A function may be added to another function. Therefore, the proper notation would be, $f(n) \in O(h(n) + g(n))$. In this context, following result is obvious.

Theorem 2 (Big_oh Complexity Set Theorem): If a function $f(n)$ is a member of the set $O(h(n))$, then it is also a member of the set $O(h(n) + g(n))$.

Proof: There are three different cases that are required to be considered.

1) Case 1: $h(n)$ and $g(n)$ are of the same order: The function $y(n) = h(n) + g(n)$ will have the same order as that of both $h(n)$ and $g(n)$. As $f(n) \in O(h(n))$, therefore, $f(n) \in O(y(n))$, since both $y(n)$ and $h(n)$ have the same order. Hence, $f(n) \in O(h(n) + g(n))$.
2) Case 2: $h(n)$ has a higher order than $g(n)$: The function obtained through addition $y(n) = h(n) + g(n)$ will have the same order as that of $h(n)$ since the order of $h(n) > g(n)$. As $f(n) \in O(h(n))$, therefore definitely, $f(n) \in O(y(n))$ or $f(n) \in O(h(n) + g(n))$.
3) Case 3: $h(n)$ has a lower order than $g(n)$: In this case, the overall function $y(n) = h(n) + g(n)$ will have the same order as that of $g(n)$ as the order of $h(n) <$ the order of $g(n)$. From the above discussions, $O(h(n) + g(n))$ is a set that incorporates any function in the order of $y(n) = h(n) + g(n)$ or any lower order function. Therefore, it will also include the lower order function, $f(n)$ in the set. To be precise, the order of $h(n) + g(n)$ is the order of $g(n)$, which is higher than the order of $h(n)$, and thus, higher than the order of $f(n)$. Therefore, $f(n) \in O(y(n))$ or $f(n) \in O(h(n) + g(n))$.

Only a combination of two functions are considered in Theorem 2. Instead of 2, the above result may be extended to any number of m such functions. Following result is obvious.

Corollary 3 (Combination of Functions Corollary): If a function $f(n)$ is in the set $O(h_1(n))$, then it is also in the set $O(h_1(n) + h_2(n) + \ldots + h_m(n))$.

Proof: Similar to Theorem 2, there are three different cases that are required to be considered.

1) Case 1: $h_1(n)$, $h_2(n) \ldots h_m(n)$ are of the same order: The function $y(n) = h_1(n) + h_2(n) + \ldots + h_m(n)$ will have the same order as that of $h_1(n)$, $h_2(n) \ldots h_m(n)$. As $f(n) \in O(h_1(n))$, therefore, $f(n) \in O(y(n))$, since both $y(n)$ and $h_1(n)$ have the same order. Hence, $f(n) \in O(h_1(n) + h_2(n) + \ldots + h_m(n))$.
2) Case 2: $h_1(n)$ has a higher order than any other function $h_j(n)$, $j \geq 2$ in the combination: The function obtained through addition $y(n) = h_1(n) + h_2(n) + \ldots + h_m(n)$ will have the same order as that of $h_1(n)$ since the order of $h_1(n) >$ the order of $h_j(n)$, for some j where $j \geq 2$. As $f(n) \in O(h_1(n))$, therefore definitely, $f(n) \in O(y(n))$ or $f(n) \in O(h_1(n) + h_2(n) + \ldots + h_m(n))$.
3) Case 3: $h_1(n)$ has a lower order than some other function $h_j(n)$, $j \geq 2$ in the combination: In this case, the overall function $y(n) = h_1(n) + h_2(n) + \ldots + h_m(n)$ will have the same order as that of $h_j(n)$, $j \geq 2$ as the the order of $h_1(n) <$ the order of $h_j(n)$, for some j, where $j \geq 2$. From Theorem 3, $O(h_1(n) + h_2(n) + \ldots + h_m(n))$ is a set that incorporates any function in the order of $y(n) = h_1(n) + h_2(n) + \ldots + h_m(n)$ or any lower order function. Therefore, it will also include the lower order function, $f(n)$ in the set. To be precise, the order of $h_1(n) + h_2(n) + \ldots + h_m(n)$ is the order of $h_j(n)$, for some j, where $j \geq 2$, which is higher than the order of $h_1(n)$, and thus, higher than the order of $f(n)$. Therefore, $f(n) \in O(y(n))$ or $f(n) \in O(h_1(n) + h_2(n) + \ldots + h_m(n))$.

3.2 Determining Temporal Complexity

Common coding constructs with guidelines to their temporal complexity analysis are presented as follows. Real-life coding contains one or more of these fundamental structures.

1) *Simple Statement Sequence:* A set of independent statements following one after another. A general structure is, $S_1;\ S_2;\ \ldots; S_k$, where k is a constant, and each S_i, $1 \leq i \leq k$ represents an independent program statement. If c_i designates the CPU time consumed to execute statement, S_i, for $i = 1, 2, \ldots, k$, then the total time consumed is, $\sum_{i=1}^{k} c_i = C$, which is a constant. Applying the algorithm to determine the big-oh complexity, the complexity order is, $O(1)$.
2) *Simple Loops:* Following is a prototype for loop found in many programs.

   for($i = 0;\ i < n;\ i++$) $\{S_t;\}$

   Total time to execute the loop = (the number of times the loop executes)$\times$ (the time required for each execution of the loop). The statement sequence, S_t consumes C amount of constant time. With the loop, S_t executes n different times. Therefore, aggregate time spent in the loop is, Cn. Utilizing algorithm for big-oh complexity, this simple loop is, $O(n)$.
3) *Nested Loops:*

   for($i = 0;\ i < n;\ i++$)
   for($j = 0;\ j < n;\ j++$) $\{S_t;\}$

   Here, the statement sequence S_t consumes C amount of constant time. The nested loop executes n times for each execution of the outer for loop. Therefore, it executes $n \times n = n^2$ times in total. Total CPU time consumed is, $C \times n^2$. Big-oh complexity is, $O(n^2)$. If there are k nested loops, each executing n times, the complexity order will be $O(n^k)$.
4) *Inner Loop Index Depends on Outer Loop Index:*

   for($i = 0;\ i < n;\ i++$)
   for($j = 0;\ j < i;\ j++$) $\{S_t;\}$

   Here, for n execution of the outer loop, the nested loop executes, $(0 + 1 + 2 + \cdots + (n-1))$ or $\sum_{k=0}^{n-1} k$ or $\frac{(n-1)\times(n-1+1)}{2}$ times. The complexity order is, $O(n^2)$.
5) *If-then-else statements:* With *If-then-else statements,* the worst-case time complexity is important. The worst-case time is the time required by the test, plus either the *then part* or the *else part* time, whichever is larger. Consider the following code:

   if (x is equal to y) **then**
   return false;
   else
   { for($i = 0;\ i < n;\ i++$) { S_t; }
   return true; }
   end if

   In this example, either the *if* or the *else* part will be executed. Let the time for *if test* be c_0. If each return statement takes c_1 amount of time, then the else part (larger) will yield with the time complexity function, $f(n) = (c_0 + c_1 + Cn)$. This *if-then-else* structure has a linear time complexity, $O(n)$.
6) *Loop Index Varies Nonlinearly:* These are also known as Logarithmic loops, since their complexity order is always logarithmic. There are three types of logarithmic loops: Multiplication loops, Division loops, and a Combination.

   a) *Multiplication Loops:* Here, the loop control variable is initialized to its minimum value, which is usually 1. The control variable then increases by a constant real or integer multiplication factor greater than 1.0 ($k > 1.0$) at each loop iteration up to the upper-bound (n). Once the value of the loop control variable exceeds the upper-bound, the looping terminates. The following is an example:

$j = 1$;
//In above, j is the loop control variable.
while ($j \leq n$) { S_t; $j = j * k$; }

Suppose the $while$ loop executes i times. Therefore, $1 \times k^i \leq n$. Using properties of logarithm, $i \leq log_k(n)$. Hence, the maximum possible value for i, $i_{max} = log_k(n)$. If each execution of the $while$ loop takes C amount of constant time, the total time required is, $C \times i_{max} = C \times log_k(n)$. Using the properties of logarithm, $log_k(n) = log_2(n) \times log_k(2)$. But $log_k(2)$ is a constant. Hence, the big-oh complexity for the multiply loop is, $O(log_2(n))$.

b) *Division Loops:* A division loop commences with the loop control variable initialized to its upper bound (the maximum value, n), which then gradually decreases by a constant division factor (k) greater than 1.0 ($k > 1.0$) at each loop iteration up to the given lower-bound (usually 1). Once the value of the control variable becomes less than the pre-set minimum, the loop terminates. Following is an example:

$j = n$;
// Here, j is the loop control variable.
while ($j \geq 1$) { S_t; $j = j/k$; }

Suppose the $while$ loop executes i times. Therefore, $\frac{n}{k^i} \leq 1$. This provides, $n \leq k^i$, or $log_k(n) \leq i$. Hence, the minimum i is, $i_{min} = log_k(n)$. Each execution of the $while$ loop consumes C amount of time. Therefore, the total time required is, $C \times i_{min}$. Also, $log_k(n) = log_2(n) \times log_k(2)$, with $log_k(2)$ being a constant. Using the algorithm, big-oh complexity for the division loop is $O(log_2(n))$.

c) *Multiplication and Division Loops Combined:* Complex coding constructs might include a combination of multiplication and division loops, one being nested within the other. The complexity order of such nested loops is, $O((log_2 n)^r))$. Here, r is the total number of loops. An example follows.

$j = n$;
// Here, j is a loop control variable.
while ($j \geq 1$) { $l = 1$;
// Here, l is another loop control variable.
while ($l \leq n$) { S_1;
$l = l * k_1$; } // Here, $k_1 > 1.0$.
S_2;
$j = j/k_2$; }
// Here, k_2 is an integer or a real factor.
// Always, $k_2 > 1.0$.

Suppose that the inner loop executes d_1 times and the outer loop executes d_2 times. The inner loop continues as long as, $1 \times {k_1}^{d_1} \leq n$. This eventually provides, $log_{k_1}(n) \geq d_1$. The maximum possible iteration of the inner loop is, $d_{1_{max}} = log_{k_1}(n)$. Similarly, the minimum possible iteration of the outer loop is, $d_{2_{min}} = log_{k_2}(n)$. For each execution of the outer loop, the inner loop executes a maximum of $log_{k_1}(n)$ times. Hence, for $log_{k_2}(n)$ iteration of the outer loop, the inner loop executes for a total of $log_{k_2}(n) \times log_{k_1}(n)$ times. Suppose each execution of the inner loop takes C_1 amount of constant time. Hence, total time consumed in executing the inner loop is, $C_1 \times log_{k_2}(n) \times log_{k_1}(n)$, which is the highest order term in complexity function. Using the properties of logarithm, $log_{k_2}(n) \times log_{k_1}(n) = log_2(n) \times log_{k_2}(2) \times log_2(n) \times log_{k_1}(2)$. Here, both $log_{k_2}(2)$ and $log_{k_1}(2)$ are constants. Assuming $log_{k_2}(2) \times log_{k_1}(2) = C$, the highest order term becomes $C_1 \times C \times log_2(n) \times log_2(n)$. Therefore, the complexity order is, $O((log_2(n))^2)$. If there are r nested loops, exercising a similar approach, the overall time complexity is, $O((log_2(n))^r)$. Here, r is an integer, and $r \geq 2$.

Following result is obvious.

Theorem 4 (Loop Control Factor Theorem): For Multiplication and/or Division Loop(s) to converge, the multiplication and/or the division factor, k used together with the Loop Control Variable(s) should be an integer or a real number strictly greater than 1.0.

Proof: The factor k is used to gradually reduce the loop control variable (LCV) to converge it to its set pre-determined maximum for multiplication loop(s), or to the pre-determined minimum for division loop(s). Looping continues until k reaches or exceeds the preset value. If $k < 1.0$, the LCV value will decrease for multiplication loop(s), and will increase for the division loop(s). As a result, the variable will never reach or exceed the preset maximum and/or minimum value, and will generate an infinite loop. If $k = 1.0$, the LCV value will never change, and as a result, will never reach or transcend the preset value, also resulting in an infinite loop. Therefore, strictly, $k > 1.0$.

From the above theorem, following result is obvious.

Corollary 5 (Nested Multiplication/Division Loops): For any number, m of nested Multiplication and/or Division Loops, if the corresponding loop control factors are $k_1, k_2, \ldots, k_m$, respectively, and if $k = k_1 \times k_2 \times \ldots \times k_m$ represents the overall loop control factor for the entire nested coding construct, then k is an integer or a real number strictly greater than 1.0.

Proof: For nested Multiplication and/or Division loops, the loop control factors, k_j, $j = 1, 2, \ldots, m$ for each loop needs to be strictly larger than 1.0 for the individualized loop to terminate. The factor k is the overall loop control factor for

the entire nested coding construct, and $k = k_1 \times k_2 \times \ldots \times k_m$. Therefore, using Theorem 4, $k_1 > 1.0$, $k_2 > 1.0$, ..., $k_m > 1.0$. From the principle of mathematics, the multiplication of any number of factors each strictly greater than 1.0 will result in a value strictly larger than 1.0. Hence, $k > 1.0$, for the nested coding construct to terminate.

4. Spatial Complexity Through Big-Oh

Big-oh complexity may provide upper-bound on memory requirements as well. Recursive and dynamic data structures most frequently influence the big-oh space complexity. For spatial requirement, there are 2 parts. The Fixed Part, which is independent of input and instance characteristic, includes the instruction space and the data space for holding simple variables, constants, compiler generated temporary variables, etc. The Variable Memory Requirement depends on the size of the input parameters and the particular problem instance being solved. This part involves dynamically allocated memory, recursive stack space, etc. The total computational memory (RAM for PCs) requirement may be expressed as, $S = S_f + S_v = C + S_v$. Here, S is total spatial requirement, S_f is the Fixed Space (which is a constant C for a specific computer program), and S_v is the Variable Space. Using the $DSPACE$ notation introduced earlier in section 2, $DSPACE(f, n) = S = (C + S_v)$. Following represents a comprehensive guideline to spatial complexity analysis of algorithms and functions.

4.1 Spatial Complexity of Algorithms

Spatial complexity plays a major role in dynamic memory allocation and recursive computation. When it comes to recursion, it is not recommended to look at the Average Space Complexity, since failure to comply with the recursive space requirement would eventually lead to program crash. Therefore, in most of the cases, the analysis is done for the worst-case spatial complexity.

1) *Simple Dynamic Array:* Java frequently supports dynamic array declaration inside its heap space.

```
// Following determines the minimum in an array.
int k = 0, n;
String n_string = keyboard.readLine();
n = Integer.parseInt(n_string);
int A[] = new int[n];
Random generator = new Random(1000000);
for (int i = 0; i < n; i++) {
A[i] = (generator.nextInt(3 * n) + 1); }
for (int i = 1; i < n; i++) {if (A[i] < A[k])
                              { k = i; } }
```

Assuming, each of an integer and a string type element occupies 1 unit of memory, $C_1 = 1 + 1 + 1 + 1 = 4$ for holding the values of k, n_string, n and i. Also, *generator* takes up a fixed k units of memory space. Hence, $C = (4 + k)$. The actual value of n will depend on the keyboard input, which in turn determines the dynamic array size. Hence, $S_v = n$. Therefore, $DSPACE(f, n) = n + 4 + k$, and $DSPACE(f, n) \in O(n)$.

2) *Nested Dynamic Arrays:* Java supports dynamically allocated nested arrays on its heap space.

```
// Java code that performs matrix addition.
int n, i, j;
String n_string = keyboard.readLine();
n = Integer.parseInt(n_string);
int P[ ][ ] = new int[n][n];
int Q[ ][ ] = new int[n][n];
int R[ ][ ] = new int[n][n];
Random generator = new Random(1000000);
for (i = 0; i < n; i++) {
   for (j = 0; j < n; j++) {
     P[i][j] = (generator.nextInt(4 * n) + 1);
     Q[i][j] = (generator.nextInt(4 * n) + 1);
     R[i][j] = P[i][j] + Q[i][j]; } }
```

There are 3, $n \times n$ matrices required for the above code inside dynamic memory area. The size of each matrix is contingent on input size, n. Hence, total dynamic memory requirement, $S_v = 3n^2$. The fixed space for storing i, j, n_string, n and $generator$ is, $C = 4 + k$ units. Therefore, $DSPACE(f, n) = 3n^2 + 4 + k$. Accordingly, $DSPACE(f, n) \in O(n^2)$, which is a quadratic space complexity.

3) *Recursive Data Structures:* In any recursive method, space is always required to hold the stack frames [4] created by the recursive calls inside the dynamic memory area. Maximum dynamic memory required to hold the stack frames is, $SF_n =$ (size of each stack frame)$\times$ (the depth of recursion). Here, n is the input size. For a specific recursive data structure, the size of each stack frame is a constant, C_{sf}. Hence, $SF_n = C_{sf} \times$ (depth of recursion). Stated another way, $SF_n \propto$ (depth of recursion). The depth of recursion, DR_n is a function of input size, n, and $SF_n \propto DR_n$.

4) *Sorting Algorithms:* Quick Sort works by partitioning the list elements, and is an *in-place sorting algorithm*. The algorithm is fundamentally recursive [3], and fits nicely with the recursive data structure. All extra space required for sorting with quick sort comes from the stack of the recursive calls in the *environment stack space*. Its space complexity is $O(n)$, since in worst-case computation, the number of activation records [4] on the recursive stack space can be in $O(n)$.
The average Merge Sort exacts an amount of space proportional to the *average height* of the tree structure describing the recursive calls, which is $O(log_2(n))$ [3].

Hence, the algorithm designers might think that $O(log_2(n))$ space is sufficient in general. If applications are designed with the average space in mind, and if for sufficiently large values of n, the worst case space complexity occurs due to a given problem instance, and the size of the allocated memory is exceeded, the program will simply crash!

Merge Sort works by coalescing the sorted sublists. If implemented as a recursive algorithm, the depth of recursion for Merge Sort with an input of size n is, $log(n)$. Hence, for Recursive Merge Sort, the space complexity is, $O(log(n))$. For Iterative Merge Sort using an array, the dynamic memory requirement depends on input size, n. The iterative merge sort is usually performed using a scratch pad buffer in the dynamic memory area, which is proportional to the size of the array. Hence, the iterative merge sort space complexity is, $O(n)$. This is also the lower-bound on memory requirement due to the size of the scratch pad buffer. Hence, $S(n) \in \Theta(n)$. *Table* 1 shows the spatial complexity of common sorting algorithms.

Table 1: Spatial complexity of sorting algorithms.

Sorting Algorithm	*Space Complexity*
Quick Sort	$O(n)$
Merge Sort(recursive)	$O(log(n))$
Merge Sort(iterative)	$O(n)$
Bubble Sort	$O(1)$
Selection Sort	$O(1)$
Insertion Sort	$O(1)$
Shell Sort	$O(1)$
Heap Sort	$O(1)$
Radix Sort	$O(n)$

5. Coalescing Complexities

Both temporal and spatial complexities largely influence algorithm design. Algorithms for Fibonacci sequence are widely researched in computing. There are four major algorithms for the Fibonacci sequence.

1) *Recursive Fibonacci Sequence:* The following program recursively computes the Fibonacci sequence. In the program code, $F[n]$ computes the nth fibonacci number.

   int F(int n) { if ($n \leq 2$) return 1;
   else return $F[n-1]$ + $F[n-2]$; }

   The temporal complexity of this recursive version is, $T(n) \in O(t^n)$, which is *exponential*. Here, $t = 1.61803$, and is the *golden ratio*. Space complexity is, $S(n) \in O(n)$, which is linear on the size of the recursive stack space. The algorithm needs to perpetuate two recursive stacks at the same time due to the nature of recursion. The space-time bandwidth product, $B(n) \in O(nt^n)$, is *pseudo-exponential*.
2) *Dynamic Programming (DP) Fibonacci Sequence:* The dynamic programming-based fibonacci sequence uses the following coding construct.

   int[ ] fibonacci(int n) {
   int[ ] F = new int[$n+1$];
   int $F[1]$ = $F[2]$ = 1;
   for (int $i = 3$; $i \leq n$; $i++$)
   $F[i]$ = $F[i-1] + F[i-2]$;
   return F;}

   The recursive fibonacci algorithm is *exponentially slow*, since for many of the intermediate values, the algorithm recomputes the same subproblems for the same fibonacci numbers repeatedly. The DP version surmounts this problem by storing the intermediate fibonacci numbers in a table inside the computational memory when they are initially computed once. As a result, it is necessary to have a one dimensional array F with $(n+1)$ elements. Each fibonacci number $F[i]$ within the array is computed iteratively, by adding together $F[i-1]$ and $F[i-2]$, which can be retrieved from the dynamically filled table within the computer's main memory area.

   Time complexity function, $f(n)$ for this DP version is, $f(n) = (n-1) + (n-2) + 3 = 2n$. The constant additive term 3 in $f(n)$ is due to the fact that the first 3 statements are always executed. Also, $T(n) \in O(n)$. Java implementation of the algorithm requires a dynamic array of size $(n+1)$ to hold the fibonacci numbers. As a result, the spatial complexity for this implementation is also, $S(n) \in O(n)$, which may be reduced to $O(1)$ through using a static instead of a dynamic array.
3) *Space Efficient Fibonacci Sequence Algorithm:* The DP algorithm for fibonacci sequence may be modified to use a much smaller amount of computational memory.

   // Code to compute the nth fibonacci number.
   int fibonacci(int n) { int $a = 1$, $b = 1$;
   for (int $i = 3$; $i \leq n$; $i++$) {int c = $a + b$;
   a = b; b = c;}
   return b;}

   In the above program, c represents $F[i]$, b represents $F[i-1]$, and a represents $F[i-2]$. The 2 extra assignments after the sum shift those 2 values over in preparation for the next iteration. The time complexity for this iterative algorithm is, $T(n) \in O(n)$.

   With this modified version, each step through the loop uses only two previous values of $F(n)$. Instead of storing these values in a static or in a dynamic array, they are stored as two independent variables.

Though this requires some swapping around of values, the computational memory requirement substantially reduces.

4) *Fibonacci Sequence Through Binet's Formula:* Binet's formula is efficacious considering both time & space, since it does not use recursion or iteration.

// Binet's Formula: The nth fibonacci number.
$F(n) = Round(\frac{(\frac{(\sqrt{5}+1)}{2})^n}{\sqrt{5}})$;

6. Conclusion

Both the temporal and the spatial requirements are of paramount importance in computer implementation of algorithms. Time and Space requirements are expressed jointly through a unique parameter known as the *space-time bandwidth product*, denoted as $B(n)$. For a temporal complexity, $T(n) \in O(g_1(n))$, and the corresponding spatial complexity, $S(n) \in O(g_2(n))$, the space-time bandwidth product is, $B(n) \in O(g_1(n) \times g_2(n))$. Bandwidth product shows the *upper-bound* on computational resources requirements for a given coding construct. Following table shows the bandwidth products for frequently used sorting algorithms. In *Table* 2, n is the input size and k is the number of digits in each input.

Table 2: Space-Time Bandwidth Product for sorting algorithms.

Sorting Algorithm	*Bandwidth Product*
Quick Sort	$O(n^2 log_2(n))$
Merge Sort(recursive)	$O(n(log_2(n))^2)$
Merge Sort(iterative)	$O(n^2 log_2(n))$
Bubble Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Shell Sort	$O(n^{1.25})$
Heap Sort	$O(nlog_2(n))$
Radix Sort	$O(kn^2)$

The *heap sort* has the best bandwidth product, which is $O(nlog_2(n))$. The algorithm also has the best temporal complexity in the order of $O(nlog_2(n))$. However, the best known sorting algorithm with $O(nlog_2(n))$ temporal complexity is the quick sort [4]. Hence, the algorithm with the best bandwidth product is not necessarily the algorithm with the best performance. The bandwidth product is just an indicator of the upper-bound on the overall computational resource requirement.

Spatial complexity plays a significant role in the Recursive Models of Computation. Recursive data structures consume considerable amount of dynamic memory inside the computer's recursive stack space [6]. Knuth's Spatial Complexity Theorem [6] relates the lower bound on the computational space requirement for an algorithm to its temporal complexity. According to Knuth, an algorithm that takes up $O(g(n))$ time consumes $\Omega(log(g(n)))$ space. The result implies that if an algorithm consumes $g(n)$ time, then for executing the algorithm, at least $log(g(n))$ space will be required. Suppose an algorithm is exponential and executes in C^n time. Hence, $g(n) = C^n$, where C is a constant, and $C \geq 2$. Therefore, the algorithm requires at least $log(g(n))$ $= log(C^n)$ space for its execution. However, $log(C^n)$ = $n \times log(C)$. As C is a constant, for any logarithmic base, $log(C)$ = C_1 will also be a constant. Hence, the minimum space requirement $= C_1 \times n \in \Omega(n)$. Hence, any exponential time algorithm will require at least a linear space for its implementation. Space-efficient exponential time algorithms run in linear space in the order of n.

There is a simple packet classification algorithm in computer networking that takes up $O(m)$ time to perform a lookup. Here, m is the number of packet fields. However, the algorithm requires $\Theta(n^m)$ storage, where m is the number of packet fields and n is the number of rules. Therefore, the algorithm may not be used with large databases. There are alternative algorithms that require only $O(nm)$ storage, which are more suitable for large database applications. Therefore, it is always prudent that polynomial or pseudo-polynomial complexity algorithms are preferred over the exponential complexity algorithms. However, the factorial complexity algorithms are the worst.

In future, the model presented in this paper will be considered with real-life computation. This avenue is not well-explored yet, and may unveil a new paradigm in computational research.

References

[1] A. Tarek, "A Generalized Set Theoretic Approach for Time and Space Complexity Analysis of Algorithms and Functions," *WSEAS TRANSACTIONS ON MATHEMATICS*, issue 1, vol. 6, pp. 60-68, Jan. 2007.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 3 Sorting and Searching*, 2nd ed., New Jersey, USA: Addison-Wesley, 1998.

[3] E. L. Leiss, *A Programmer's Companion to Algorithm Analysis*, Florida, USA: CRC Press, 2006.

[4] R. F. Gilberg, and B. A. Forouzan, *Data Structures - A Pseudocode Approach with C++*, California, USA: Brooks/Cole, Thomson Learning, 2001.

[5] A. Tarek, and A. Farhan, "A Realistic Approach to Spatial and Temporal Complexities of Computational Algorithms," in *Proc. MCBE'13*, 2013, paper 70606-106.

[6] A. Tarek, "A New Paradigm for the Computational Complexity Analysis of Algorithms and Functions," *INTERNATIONAL JOURNAL OF APPLIED MATHEMATICS AND INFORMATICS*, issue 1, vol. 1, pp. 5-12, 2007.

[7] A. Tarek, "Computational Complexity Simplified," in *Proc. WSEAS American Conference on Applied Mathematics*, 2008, p. 130-135.

[8] A. Tarek, "A Generalized Set Theoretic Approach for Time and Space Complexity Analysis of Algorithms and Functions," in *Proc. 10th WSEAS International Conference on APPLIED MATHEMATICS*, 2006, p. 316 - 324.

[9] G. Varghese, *Network Algorithmics - An Interdisciplinary Approach to Designing Fast Networked Devices*, California, USA: Morgan Kaufmann, 2005.

On Performance of Distributed Computer Systems

H Cai[1], S Monkman[2], I Schagaev[2] and O Santos Naval[3]
[1]Shantou University, China
[2]IT-ACS LTD, Stevenage, UK
[3]Londonmet, UK

Abstract - *Any system is evaluated in terms of performance, taking into account performance of elements and system as a whole. Good systems exceed performance of their components, sometimes exceed a production of component performance. Poorly designed systems overall performance is much less that production, or even sum of performances of their components. In terms of this classification our computer systems are poorly designed. These notes are about performance and ways to analyze it. We also introduce some simple models for thinking through system performance and ways to improve it.*

Keywords: Distributed Computing, System Performance, Parallel Computing, Amdahl Ratio, Run-time Systems

1 Introduction

Suppose one element - active zone *az* on the Figure 1 has a performance Pi; then system of n elements if we can add performance will have maximum performance as n*Pi; i.e., linear growth is assumed. Unfortunately, properties of both: an external interaction zone (EIZ on Figure 1) and task structure reduce our expectations about unlimited, or just linear performance growth, while we introduced more elements Figure 1.

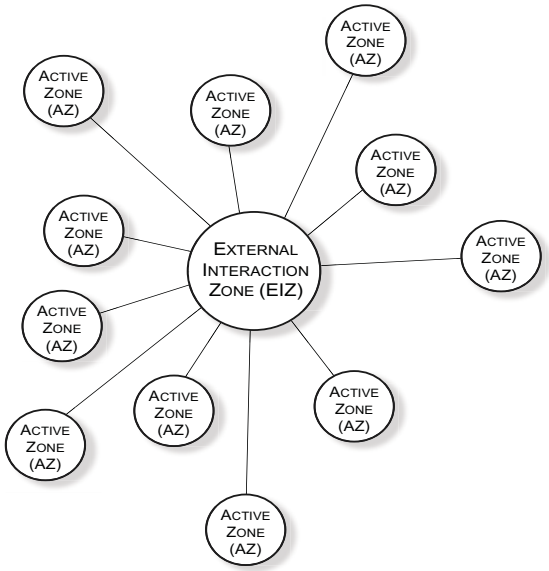


Fig. 1 Distributed system, *az* - active zone

System performance we can see as a function of performance of elements Pi, number of elements and EIZ:

$$Ps = f(Pi, EIZ, n)$$

Thus, the structure of EIZ and its dynamic features (the ability to connect, transparently, an arbitrary number of elements with heavy interactions (in our case information exchange)) will impact on performance of both: system level of performance and element level.

It is clear that element with interaction will waste some own performance to provide interaction with others: Figure 2.

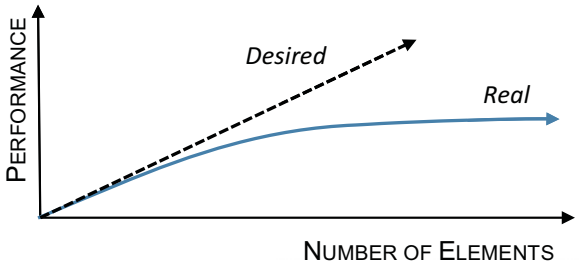


Fig. 2. Performance grow limitations by structure

Above all, a program structure that impacts on the performance of system is crucial, as well as ability of program to split into independent elements.

In late 80's John Gustafson did show [Gustafson88] that when map of a program fit map of available hardware almost linear growth of system performance is possible. This result in fact denies Von Neumann architecture arguing that special purpose system should be build to execute special purpose programs.

Turning back to analysis of what is possible and feasible a program structure should be for sure analyzed in details. One of possible approach consider program as three connected graphs. Three graphs that represent a program: control, predicate, and data dependency [Blaeser14] can help to analyze limits of performance gain for various types of program and on performance of the system as a whole.

2 Information processing aspects

On the information processing level, we can consider a system as a black box with input *x* and output *y*, with arbitrary function *F*, Figure 3;

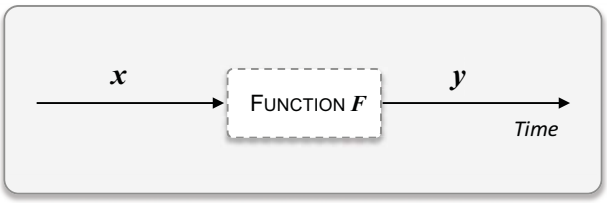


Fig. 3. System as a black box, with arbitrary function F

A function *F,* or a task of this function execution - a box on Figure 3 illustrates the delivery of result (*y*) from an input *x* within allocated time.

For some functions or programs that execute them we do not need all inputs available to begin execution. In principle, input and output timing very often are loosely dependent, input *x* might have its own duration in time while readiness of output *y* has its own duration, both might be overlapped, see Figure 4.

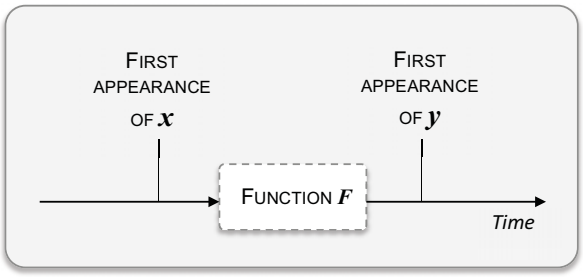


Fig 4. Input appearance overlapped with outcome

Thus we can consider x and y not as variables but as functions or processes; this distinct computing from mathematics. Further research might be required to compare ways and durations of x and y possible overlapping.

2.1 Information systems task wise performance

Information systems are a combination of three-wares: userware, software and hardware: UW, SW and HW respectively, see Figure 5.

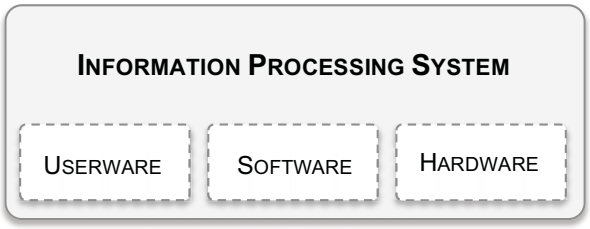


Fig. 5. Information system components

For information processing system one has to consider a performance of all three: UW, SW, HW. In the long run performance and efficiency of the system depends on user-ware, software and hardware performance. Their combination varies in various applications - one case is illustrated by Figure 6.

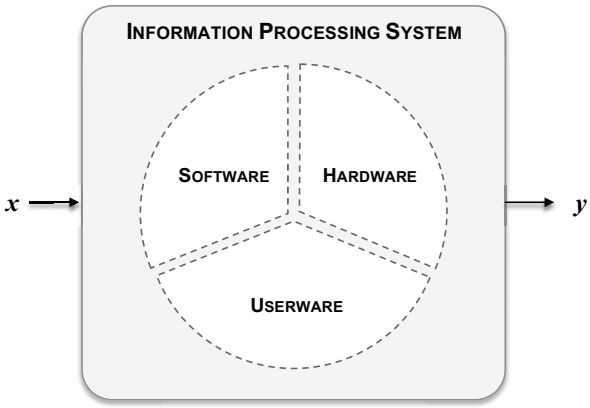


Fig 6. IS box: x+time = y

Over the last 40 years, user features of computer applications were largely ignored in all domains of computer market: general purpose, embedded, high performance computer systems.

This is a subject of special study on UW-SW-HW systems. Here we present a simple model that helps to estimate an impact of system software (SSW) on overall system performance. We attempt to answer a question

What is the performance and efficiency of a computer system in terms of missions or tasks?

Performance is about task completion in time allocated. The task allocation and analysis of this process should account for a hierarchy of components: UW, SSW, HW. In principle, one might introduce into this estimation a role and efficiency of management, but it goes well beyond the scope of our work.

2.2 SSW-HW performance model

User tasks that were developed as a program (usually called "an application") present a sequence of instructions executed by hardware. Several supportive programs – (parts of run-time system), accompany any user task; these programs are generally called system software, further SSW. Figure 7 illustrates a sequence of user tasks accompanied by system software tasks.

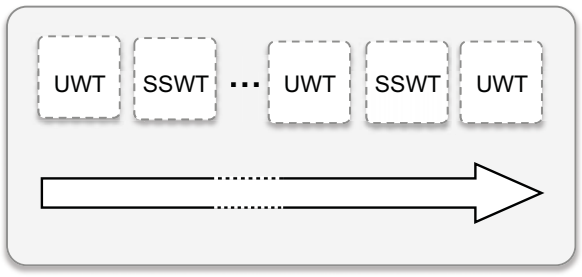


Fig. 7 User and System task sequence

An efficiency of a system in terms of user tasks depends on use of *informational* and *structural* and *time* resources [Castano15].

A total amount of hardware workload in number of instructions W_{ux} to perform user task x can be expressed as (1) where i, j indexes stand for number of hardware instructions required to complete supportive actions (system software need) and user ones (1):

$$W_{ux} = \sum_{j=1}^{m} h_j(sswt) + \sum_{i=1}^{n} h_i(uwt) \quad (1)$$

Indexes m and n stand for system software and user software instructions execution time. Assuming that all hardware instructions have similar execution time (for RISC systems it is essential design condition) one might introduce an efficiency of an architecture or a system as a ratio as shown below, (2):

$$E_{ux} = \frac{\sum_{i=1}^{n} h_i(uwt)}{\sum_{j=1}^{m} h_j(sswt) + \sum_{i=1}^{n} h_i(uwt)} \quad (2)$$

We will dig deeper on performance and efficiency but here it is worth summarizing an existing relation of efficiency and performance.

Definition 1. Efficiency Eux of computer system is a ratio number of instructions required to perform user task to the total number of instructions performed by computer system.

Naturally, efficiency Eux → 1, while m → 0, and no matter what frequency a processor is using if **m >> n**, Eux → 0.

Regretfully, it is the case for current state of the art in computer systems.

Regretfully again, our implementations in terms of efficiency are causing mostly pessimistic observations:

- Application of Java, or support of modified standard operating system unavoidably reduce system efficiency of general system;
- In the case of embedded systems runs out our computer batteries for nothing;
- For military systems availability and reactiveness of the system is substantially lower than it could be;
- For office systems – in terms of user efficiency - employees are sitting and waiting for Windows or Cisco service more than they work…

2.3 Distributed computing

In the late 1960s, an idea for the parallelization of computer program using distributed computing paradigm instead of single processor scheme was proposed [Amdahl67].

It was declared that parallelization of tasks and programs and use of available distributed hardware for support of parallel execution is the most feasible way to boost system performance.

Later, Sun [Sun94] introduced "system fallacies" of distributed computing (Table 1). Omitting topologic factors and paying attention to Fallacy 2, 3 and 7, we discover that these fallacies fit into the area of parallel, closely connected computers with multiprocessors – in fact, all modern computers.

Table 1 Sun fallacies of distributed computing

1 The network (distributed system) is reliable;
2 Latency is zero;
3 Bandwidth is infinite;
4 The network is secure;
5 Topology doesn't change;
6 There is one administrator;
7 Transport cost is zero;
8 The network is homogeneous;

If we look harder these fallacies might be not strong enough and some of declared features described became obsolete.

Besides, again, when definition includes eight other elements that are not connected or have vague relation to each other it seems odd or at least inconsistent.

If we follow Sun definition we are not including Internet into the distributed computing even as a supportive hardware infrastructure. Anyway, we've proposed our own définition of distributed computing:

Definition 2. ***Distributed computing is a paradigm that assumes an execution of functionally connected tasks as a single process over distributed media and resources.***
Clearly, a joint collaborative work of thousands of processors at once might bring substantial profit for both - loosely connected tasks (when they share HW resources, but not logically connected, such as Google cluster), or closely tight models that include of several thousands of DE.

But in the second case, it is much harder to get the gain from distributed computing, and it is not a surprise.

Amdahl described drawbacks of distributed computing in the late 1960s [Amdhl67], highlighting that even small parts of a program must be parallelized to reach their full potential. This way linear growth of speedup is not possible at all.

In other words, if 1 is a length of a sequential program and we have managed to parallelize p fraction of it then sequential part is shrinking down to 1-p, while parallel part requires p/n time where n stands for number of processors, (3) and Fig. 8.

$$S = \frac{1}{1 - p + p/n} \tag{3}$$

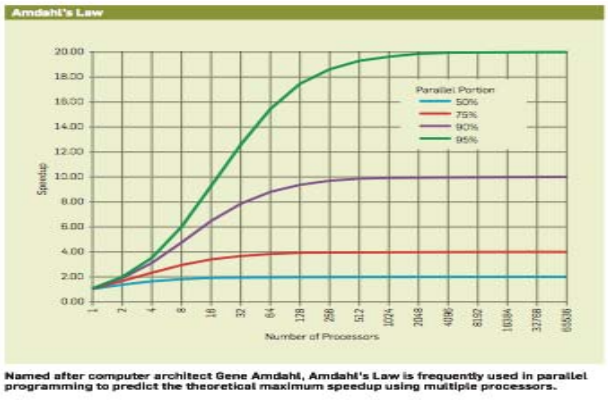


Fig. 8 System speedup by Amdahl [Goth09]

3 Real Performance & Amdahl "Law"

Relative gain in performance is usually referred to as "Amdahl's law". Well, "law" in terms of science, not society is "a regularity in the material world" (Shorter Oxford English Dictionary, 6e, Vol 1). Thus, defining "the law" as two simple proportions (3), (4) of performance after improvement P_{ai} with performance before improvement P_{bi} is, to put it politely, too ambitious.

$$\boldsymbol{Speedup} = \frac{\boldsymbol{P_{ai}}}{\boldsymbol{P_{bi}}} \tag{4}$$

But this proportion is useful to evaluate a success of the modification of processor structure in re-iterative design. What is interesting here is that the expectation of linear growth of performance by improving element performance (Figures 1 and 2) has nothing near to the real situation.

It means that if we make super parallel execution of 80% of a program, we still have to complete another 20% sequentially. Amount of speedups *vs*. number of processors as a family of functions, is presented on Figure 8 above, taken from [Goth09].

3.1 A Fine-tuning of Parallel Speedup Model

The theory behind computational work in parallel has some limitations that reduce the advantages of parallelization. Usually, the goal in large-scale computation is to get as much work done as possible in the shortest time within the budget.

Furthermore, the system can be considered good and well-designed when it is able to get a big job done in less time, or a bigger job done in the same amount of time without any problem; in other words, a system should be a scalable.

Therefore, the power of a computational system can be represented as the amount of computational work done, divided by the total time it takes to do it. It is important to emphasize that usually the aim is to increase power per unit cost, or more importantly nowadays, cost-benefit, and in this regard physics and economics conspire to limit the raw power of individual single processor systems available to perform any particular piece.

It is agreed within the research community that the cost-benefit scaling of increasingly power single processor systems is usually non-linear and very poor. For instance, one processor that is twice as fast might cost four times as much, yielding only half the cost-benefit per pound.

Physics sets its own limit as well – a so-called "thermal barrier" [Castano15] - an amount of heat that material is capable to dissipate is limited making endless increase of frequency of operation impossible.

These two arguments are usually applied to justify an alternative solutions and development of parallel designs. There are some drawbacks though, as Amdahl pointed out, and they are serious.

Let us rewrite Amdahl ratio in terms of time: T(N) will be the time necessary to finish the task on N processors. The speedup S(N) is expressed by the ratio (5):

$$S(N) = \frac{T(1)}{T(N)} = \frac{Ts + Tp}{Ts + Tp / N} \quad (5)$$

In many cases the time T(1) possesses, as represented above, both the serial part Ts and the parallel-able part Tp.

Unfortunately, Amdahl ratio ignores a role of run-time system tasks (see a section 2.1) that must be taken into account when a parallel execution is assumed.

A more detailed analysis of parallel speedup would include two more parameters of interest, namely:

- Ts – the original single-processor serial time;
- Tis – the average additional serial time spent performing for example inter-processor communication (IPCs), see Figure 1 where is it introduced as EIZ, setup, and so forth in parallelized tasks. It is important to note that this time can depend on N in a variety of ways, nonetheless the simplest assumption is that each system has to spend this much time one after the other, so that the additional serial time is for example N*Tis;
- Tp – the original single-processor parallel-able time;
- Tip – the average additional time spent by each processor performing just the setup and work that it does in parallel, this may as well include idle times, which is also very important and should be accounted for separately.

The most important element that contributes to Tis is the time required for communication between the parallel sub-tasks. This communication time is always there – even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must begin and be controlled with message passing over the system.

In systems with more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the distributed computing system for the calculation to proceed, which can be very costly in scaled time. The (average) additional serial time (Tis) plays an extremely important role in defining the speedup scaling of a given calculation.

Most computer systems process information sequentially. Lines of code in a computer program get translated into assembly language by the compiler, and the latter gets decoded into microcode in the processor. Everything and every step along the way is done sequentially. For example, a flowchart processing usually includes multiplication or comparison of two digits, it starts with the first digit, then the second digit is introduced and the working register is set to 0.

To explain what is real and what is not and why Amdahl rule is mostly misleading we have developed a simple model – so called "fence making model", illustrated by Figure 9 and following expert recommendations [Doit].

3.2 Parallel vs. Sequential: A fence model

Our task is to make a fence with *N* planks and two horizontal rails; each plank needs two nails and has to be "pre-processed". Two rails have to be placed at the assembling site. Each plank needs to be placed at site and finally nailed. We also need hammers and nails and sequence and instruction to operate.

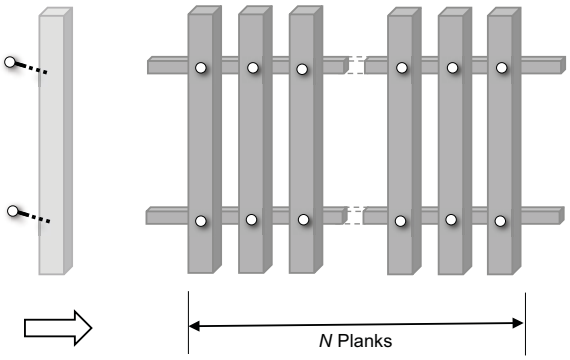


Figure 9 Fence model of processing

Task requirements: number of planks *N*; number of rows – 2.

Each plank needs to be nailed half-way through before placement for final processing and assembling a fence.

There are two principally different options to make this fence:
A) by distributing tasks;
B) by making all tasks on site sequentially.

In A) case, distributing task scheme assumes existence of agents-workers and distributers and their abilities to act:

- *N* workers for plank processing are available and ready;
- a distributor of the nails is in place;
- a distributer of the hammers is in place;
- a distributer of the planks is in place;
- a distributer of rails is in place;
- a collector of the fence segments initially is and placing the planks;
- nailing the planks at two rows are performed by workers;
- collecting the hammers is performed;
- garbage collector is in place and complete the task execution.

Case B) assumes that the same worker is doing all actions, like "a jack for all trade", has one hammer, bucket of nails and does the following:

- takes nails;
- planks where they are;
- half-nail planks;
- places them on the rails;
- nails them all;
- place fence where necessary, collect garbage.

Let us consider the process of making the fence from *N* planks in more details for both cases, assuming that nails, hammers, planks and rails are ready and placed in the local warehouse (storage and executed by "a system officers", while workers execute user task). Sequences are presented in Table 2.

Table 2 Parallel vs. sequential execution in more details

PARALLEL OPERATION	SEQUENTIAL OPERATION
Distributor	***Distributor***
Gets pack of planks	Activate worker
Distribute planks	Check garbage left
Distribute rails	
Distribute nails	
Distribute hammers	
Distribute planks along rails	
Activate *N* workers start	
Collect hammers and left garbage	
Place two rails in assembling area	
Clean garbage	
Worker	***Worker***
Receive planks	Gets packs of planks
Receive nails	Gets buckers of nails
Receive hammer	Gets a hammer
Preprocess plank (two nails nailed half-way through)	Places (distribute) planks to the assembling area
Spread planks along rails (fine tuning)	Places rails in assembling area
Nail plank (two nails) to the rails at the final assembling	Preprocess *N* planks (two nails per each)
Prepare to final assembling	Places (distribute) planks along the rails
	Nails *N* planks
	Assemble fence
	Clean garbage

Our task now is about: giving elementary time slot t_e and constant coefficients equal for both variants of fence processing prepare two variants of the fence completion as a sequence of steps for A and B cases. This will illustrate a gain from distribution of works.

We need to compare these cases as well as explain what is possible to prepare in preprocessing and what is possible only during operation. One might find useful to make a table of all works mentioned and using own experience and case estimate a concrete gain for concrete case.

Now we have to answer the following questions:

When distributed computing is efficient in comparison with sequential;

What impact system software makes on parallelization of task and efficiency of a system.

It is clear that *planks* are data, *nails* and *hammers* are programs to process data on site, *and distributer* is run-time system;
Let us leave an arithmetic exercise with various values of parameters from job descriptions above to good master students.

Our estimation indicates that overheads of run-time system for distributed execution might achieve almost 60% of user task cost (time). We add in denominator of (5) a coefficient *k,* a relative value of system software overheads per user task (6).

$$y=\frac{1}{(1-p)+k+\frac{p}{x}},\ x=\{1,2,...,10\},\ p=\{0.85\},\ k=\{0,0.1,0.4\} \quad (6)$$

Following (6) the graph of Figure 10 presents three curves in three colors: green, blue and red k=0,0.1, 04 respectively. . The top one stands for known "pure" Amdahl ratio (k=0).
Figure 10 shows that for extremely good run-time system one can double performance with 4 cores… It is still too optimistic statement, especially recalling Multics 85% and Window 65% of total workload time.

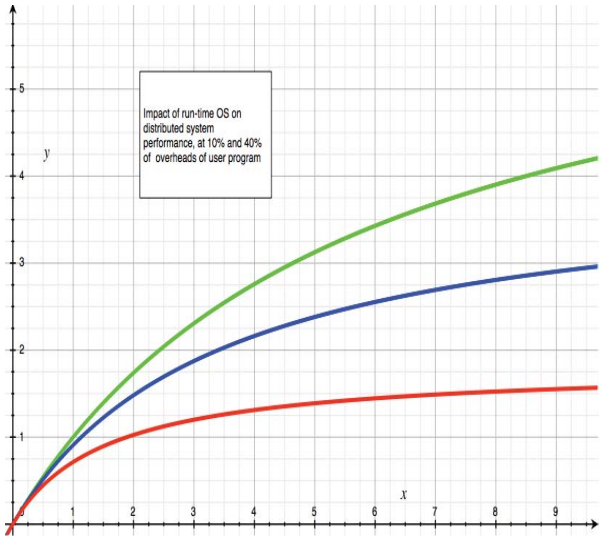


Figure 10. System software role in distributed computing

4 Conclusions and observations

The obvious observations/conclusions here are:

- Properties of distributed computing paradigm in terms of performance gain/loss are outlined.
- Models of performance and efficiency are proposed from the point of view of information processing.
- Shown how to evaluate an efficiency of computer system including role of application and system software as well as hardware.
- Amdahl ratio is analyzed taking into account system software overheads.
- System software, applications and hardware designs should be considered together when we design or try to analyze system efficiency, this paper is just one step into this direction.

5 Acknowledgements

Authors would like to express their very great appreciation to two reviewers of this paper for their valuable and constructive suggestions especially pointing us to Prof. Gustafson works.

6 References

[Amdahl67], Amdahl G. M. [1967]. "Validity of the single processor approach to achieving large scale computing capabilities," Proc. AFIPS Spring Joint Computer Conf., Atlantic City, NJ, (April) 483–85. (Written in response to the claims of the Illiac IV, this three-page article describes Amdahl's law and gives the classic reply to arguments for abandoning the current form of computing).

[Blaeser14] Blaeser L.Monkman S. Schagaev I. Evolving system, in Proceedings of the International Conference on Foundations of Comp. Science FCS'14 2014 CSREA Press, ISBN: 1-60132-270-4

[Castano15] Castano V, Schagaev I Resilient Computer System Design, Springer 2014 ISBN 978-3-319-15069-7

[Goth09] Goth G. Entering parallel universe. DOI:10.1145/1562164.1562171, CACM Sept 2009 vol 52. No/9 Pp13-16

[Gustafson88] Gustafson J. Reevaluating Amdahl's Law, Communicating of the ACM 31(5), 1988.pp 532-533

[Sun94] https://blogs.oracle.com/jag/resource/Fallacies.html

[Doit] Building wood fence
www.doityourself.com/stry/buildwoodfences

SESSION

FORMAL METHODS AND LANGUAGES

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

A simple instructional approach for proving the Non-RE status of Non-monotonic properties of formal languages

Dr. Gary L. Newell (newellg@nku.edu)
Department of Computer Science, Northern Kentucky University, Highland Heights, Kentucky, US

Abstract **-** This paper presents a relatively simple and easy to apply technique for proving a significant number of formal language properties to be Non-RE. Using the results of Rice's Theorem and the more generalized observations of the Rice-Shapiro Theorem, we derive a result that is easy for students to apply when attempting to prove that a formal language property is not Recursively Enumerable (Non-RE) . To our knowledge, this particular instructional approach has not been presented in this applicable form to date.

While Rice's theorem provides an easy to apply technique for proving the undecidability (Non-recursiveness) of a problem/language, it cannot be applied directly to determine whether said undecidable problem is RE-Non-recursive or Non-RE. This paper provides an easy to state and relatively easy to understand approach to proving that a language lies outside of the Recursively Enumerable (RE) family of languages and is thus Non-RE.

Keywords: Computational Theory, Rice's Theorem, computer science instruction, undecidabilty.

1 Introduction

A key aspect of most Theory of Computation courses is the study of what is known as the Chomsky Hierarchy of languages. Although the particular approaches or models used to explore these families of languages (e.g. Recursive Function Theory, Automata, etc.) may vary, as may the specific language categories explored, it is generally accepted that a fundamental understanding of the hierarchy is essential to a quality theoretical comprehension and background for students [1,10,11].

1.1 Chomsky Hierarchy

In some sense, the key question addressed in most theory courses is "What are the boundaries of computing?". That is, are there problems for which no complete computational solution exists? The Chomsky Hierarchy is instrumental in understanding the nature of these boundaries and the variety of language families that are known to exist and are frequently encountered in the computational solving of problems.

For the purposes of this paper, it will suffice for us to explore only a simplified version of the hierarchy. In particular we are concerned with problems that are decidable (i.e. Recursive Languages), semi-decidable (i.e. RE Non-Recursive languages) and those that are strongly undecidable (i.e. Non-RE languages).

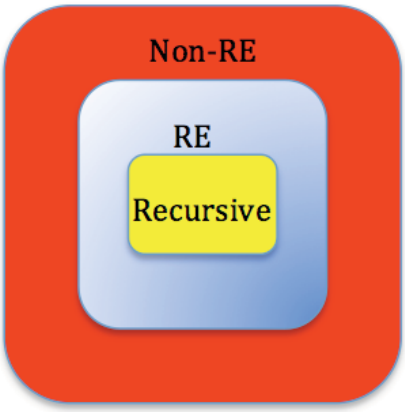


Figure 1. The specific version of the hierarchy that we are addressing.

As the figure indicates, we are interested, for any given language, which of the three regions of the chart it lies. If the language lies within the recursive family of languages then we know that it does yield a computational solution that will meet the definitions of an algorithm and can thus be programmed. We are *neither* addressing nor interested in the question of whether or not the problem yields an efficient solution but whether or not it has *any* solution at all.

Should the language in question lie within the Recursively Enumerable region (but not within the Recursive domain), then we say that it is semi-decidable. Although these languages are formally undecidable, in the sense that they yield no true algorithmic solutions guaranteed to halt for all input cases, they do yield recognizers/acceptors. For our purposes, a recognizer is a program that is guaranteed to halt and correctly recognize an input word if it is, in fact, a member of the language but provide no such guarantees if the input word is not a member of the language.

The final possibility for any language considered is that it lies within the Non-Recursively Enumerable languages (Non-RE). We shall refer to these languages as *strongly-undecidable* as they yield no algorithms that are guaranteed to halt on input words whether or not the word is a member of the language.

2 Traditional approaches to teaching and proving undecidability

As most textbooks approach the topic of undecidability through the exploration of Turing machines, we shall briefly examine the traditional, historic approach to the topic.

2.1 Early pedagogical approaches

Early theory of computation/formal languages textbooks, though rigorous, formally correct, and effective in providing the underlying theoretical justifications for the discipline of computer science, often lacked the intuitive exposition that many students require for an effective and operational understanding of the concepts at hand. Hidden within 2-3 page proofs, the key observations and techniques were often lost on students who failed to see the forest for the trees. A rare exception to this rule was the inclusion in most textbooks of Rice's theorem.

2.2 Standard Turing machine approach

A common approach to exploring the topic of decidability is to first explore the languages accepted by a standard Turing Machine [2,10]. Although the specific Turing Machine model may vary from text to text or course to course, students are usually taught that any number of variations in Turing Machine constructions/rules yield no additional power with respect to language recognition (e.g. multiple tapes, multiple tape-tracks, non-determinism, etc.).

Once students are comfortable with the basic operations of the model, they are often exposed to concept of a Turing Machine encoding and the Universal Turing Machine. This universal machine is one which expects, as input, two items, an encoded Turing Machine and an input word for the encoded machine. The universal machine can effectively simulate the step by step operations of the given encoded machine and provide the result that the encoded machine would give on the supplied input word. It is, in effect, an interpreter capable of simulating any supplied program on any supplied input.

The concepts of Recursive languages and RE languages, in general, are usually also discussed. Students learn that a language that can be processed by a Turing Machine that is guaranteed to always halt in finite time and correctly identify an input word's membership or lack thereof in the language is known as a "recursive" language (Figure 2).

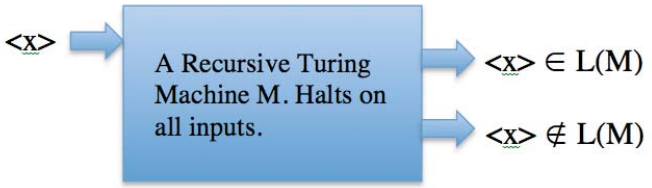


Figure 2. A recursive Turing Machine that is guaranteed to halt on all inputs and provide a correct membership determination for all inputs.

Similarly, if the membership in the language can only be recognized by a Turing Machine that is guaranteed to halt on inputs that are members of the language but are not guaranteed to halt and reject non-members, then the language is said to be "Recursively Enumerable but not recursive" (RE Non-Recursive – Figure 3). It is then usually explained that the latter are not true algorithms, as they are not required to halt in a finite amount of time on all possible inputs and thus are not decidable languages/problems [2,9,10].

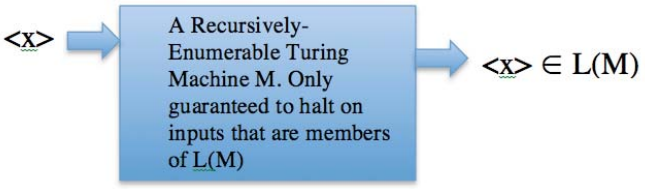


Figure 3. An RE Non-recursive Turing Machine which is only guaranteed to halt and accept on inputs that are members of the machine's recognized language.

2.3 Presentation of an undecidable problem

The next essential step in exploring decidability is to introduce a language that is provably not recursive/decidable [3,4,5]. The standard approach to introducing this concept is to use a diagonalization argument similar in style to Cantor's proof technique for proving the uncountable infinity of the cardinality of Real numbers. In short, one conceptually constructs and infinite matrix whose rows and columns are indexed by a countable enumeration of binary strings {0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, …} . The rows of the matrix are assumed to be the binary encodings of Turing Machines, while the columns are assumed to represent the possible binary strings that can be used as input to a Turing Machine. The actual entries of the table are Boolean values where Matrix[x] [y] == *true* if the Turing Machine whose encoding is the binary string x *halts and accepts* the input binary string represented by y and is *false* otherwise. Figure 4. represents a hypothetical portion of the countably-infinite matrix. Therefore, one can consider any row x of the matrix as representing the language accepted by Turing Machine x. That is, if entry Matrix[x][y] is *true* then binary string y is a member of the language of machine x. Conversely, if string y is a member of the language recognized by machine x then the entry Matrix[x][y] is *true*.

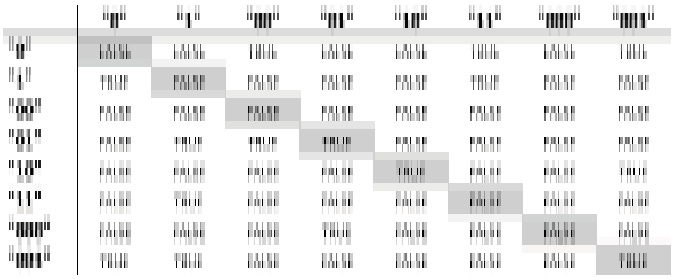


Figure 4. A portion of the countably-infinite, Boolean Matrix[x][y].

2.3.1 The introduction of diagonalization and L_d

Students are then asked to consider what the entry Matrix[x][x] represents. Most will quickly realize that the Boolean entry at position Matrix[x][x] indicates how Turing Machine x behaves when given its own binary encoding as input. If the entry is *true*, then machine x accepts its own encoding as a member of its language and if it is *false* then it does not. Please note that a *false* entry does not indicate that the machine in question halts and rejects the membership, it simply indicates that the machine does not halt and accept.

Once the nature of the matrix is understood, students are usually asked to imagine the complement of the matrix, ¬Matrix. That is, the matrix with all *true* values complemented to *false* and all *false* values complemented to *true*.

Next, students are asked to consider what the newly complemented *diagonal* represents. The diagonal from ¬Matrix[0][0] through ¬Matrix[$\aleph_0$][$\aleph_0$]. That is, they are asked, "What does the language $\forall x$ ¬Matrix[x][x] actually represent?" After some consideration, most students will recognize that the language in question, which we will call L_d, is actually the language consisting of the binary encodings of Turing Machines that do *not* accept their own encodings. That is L_d = { <M> | <M> $\notin$ L(M)} where <M> represents an encoding of some Turing Machine M and L(M) represents the language accepted by a Turing Machine M. Figure 5. Presents the complemented diagonal (L_d) for the portion of the matrix shown in Figure 4.

L_d = { TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE, FALSE…}

Figure 5. L_d for complemented Matrix shown in Figure 4.

Once they understand the meaning and nature of the diagonalization language L_d, students are reminded of how the original matrix was actually constructed. The rows of the matrix were indexed by an enumeration of *all* Turing Machine encodings. Therefore, if a Turing Machine M can be constructed, then it can be encoded and thus it's encoding will index some row of the matrix. If L_d is a language that is recognizable by some Turing Machine M then M must have a binary encoding z and thus row z of the matrix would represent the language L_z == L_d. It is at this point that students are usually posed the question "What is in entry ¬Matrix[z][z]?"

If the student answers "*true*", then it is quickly pointed out that this would mean that Turing Machine z accepted its own encoding z which contradicts the actual definition of L_z == L_d which is the language of machines that do <u>*not*</u> accept their own encodings. It is also shown that if the entry was *false*, then this would mean that machine z did not accept its own encoding which means it *should be* in the language L_z == L_d and thus should have had an entry of *true*. This resulting paradox leads us to the conclusion that there could not exist any Turing Machine for any row z of the matrix which corresponds to our language L_d. Therefore, there does not exist any Turing Machine whose accepted language is L_d and thus L_d cannot be RE and as a result cannot possibly be recursive/decidable.

2.3.2 The Universal language

Once students are introduced to a language that is provably undecidable and in fact is Non-RE, it is often the case that they are asked to look at the language L_u (the "Universal Language") which consists of machine/string pairs (represented <M,w>) such that the given machine M halts and accepts the given word/string w. That is, L_u = { <M,w> | w $\in$ L(M) }. Students are often asked whether L_u has a Turing Machine that accepts it. It is not uncommon for many students to respond "*No*" as they have recently seen a language (L_d) for which no Turing Machine exists that recognizes it. However, it can usually be easily explained that L_u is simply the language of the Universal Turning Machine. At this point, student are asked if the Universal Turing Machine is always guaranteed to halt on all <M,w> input pairs. Since the Universal Machine is simply an interpreter that simulates the actions of its given input machine it is a relatively simple observation to note that if the input machine M were to go into an infinite loop or an endless computation on some input w then clearly the Universal Turing Machine would also infinitely execute on the input <M,w>. Thus, the language L_u has a machine that accepts/recognizes it (the Universal Machine) but said machine is *not* guaranteed to halt and reply correctly for all possible inputs and thus L_u *is* Recursively Enumerable but *not* Recursive and like L_d it is *not* decidable/recursive (though L_u is Turing recognizable/semi-decidable).

2.4 Turing Reductions

Once languages such as L_u and L_d have been explored and their placement in Chomsky's hierarchy determined, most theory courses will explore other undecidable problems and techniques to their decidability-status. The most common method presented is that of Turing Reducibility [3,10,11].

2.4.1 Fundamental concepts of Turing reductions

Simply stated, the idea of any Turing reduction is to determine the classification of some unknown language (which we ill refer to as $L_?$) via a proof by contradiction. For example, to prove that $L_?$ is not recursive, we would assume that it is recursive and then using this assumption we would show that this would imply that L_u is also recursive which has already been shown to be false and thus we have our contradiction and can assume that our assumption was wrong and $L_?$ must not be recursive. Similarly, if we wished to show that $L_?$ was Non-RE, we would begin by assuming that it was RE and then show that this assumption would lead to L_d also being RE which has been shown to be false and therefore the resulting contradiction implies that $L_?$ cannot be RE. Via

exposure to reductions, students should come to recognize that the problems that are undecidable are those that deal with *properties* of the languages that arbitrary Turing Machines accept [3,10,11,12].

A key aspect of any Turing reduction is the construction of a Turing machine to serve as input for the assumed machine for $L_?$ [3,4]. This machine is constructed from the input to the L_u or L_d machine under construction. That is, the input for L_u or L_d is assumed to be the encoding for some Turing Machine <M> and using this input, a machine M_x is constructed which is designed so that the language it accepts either has the language property P in question for $L_?$ or it does not have said property depending upon whether the input machine <M> accepts its own encoding or not when run using the Universal Turing machine [3,4].

2.5 Rice-Myhill-Shapiro Theorem

The Rice-Myhill-Shapiro theorem, commonly known as "Rice's Theorem", states that any *non-trivial* property of the language accepted by an arbitrary Turing machine is undecidable [4,6]. Although the original paper deals with Partial Functions, it extends directly to the subject of computability theory. The Rice-Shapiro Theorem is a more generalized presentation of the key ideas expressed in the original paper.

2.5.1 Properties and Rice's Theorem

A *property* of Turing machine languages is simply the set of machine languages that exhibit the said property. For example, "languages that contain more than 5 strings", "languages that are regular", "languages that are context-free", "non-empty languages" and so on. A *trivial* property is one that holds either for the languages of *all* Turing machines or for *no* Turing machine languages [4,5,7,8].

Rice's theorem has long been a godsend to many students studying the theory of computation. It states that any language property can be easily shown to be undecidable (non-recursive) simply by exhibiting that at least one but not all Turing machines recognize languages with the property in question. For example, the property P_{reg} = { <M> | L(M) is a regular language }. Clearly there are some Turing machines that accept regular languages but there are others that accept languages that are not regular (e.g. context-free languages such as the language consisting of all palindromes) and therefore it is an undecidable problem to determine whether or not the language of an arbitrary Turing machine is regular.

A common way of introducing students to Rice's Theorem is by explaining that it provides a generalization of Turing Reducibility for the language L_u. That is, it generalizes the construction of the machine M_x for input into the assumed machine for the language $L_?$. In essence, we need only recognize that our default construction of M_x *always* recognizes the empty set ∅ as one of its two possible languages. Therefore we need only ask the question "Does the empty set have the property P or not?"

If the empty set does have property P, then we need only identify a Turing recognizable language that does *not* have the property and define M_x to accept this language as its second possible language. If the empty language does not have the property P, then we accept, as our second possible language, a Turing recognizable language that *does* have the property. A generalized pseudo-code construction of an M_x for the case that the empty language does *not* have the property P, inspired by Rice's Theorem is presented in figure 6.

```
Program MX( w : input_string )
{
 If (<M>  ∈ L(M)) then
   accept w  ∈ L | L ∈ P     // L is possibly Σ*
 Else
   reject w
}
```

Figure 6. Example M_X for the case that the empty language does not have the given property P.

3 Proving languages to be Non-RE with an analogue of Rice's Theorem

It is essential to understand that Rice's Theorem is intended and used only to determine the decidability of a language. That is, it is *not* designed, as stated, to distinguish whether or not the language property is RE Non-recrusive (semi-decidable) or Non-RE (strongly undecidable) [1,4,12].

3.1 Confusion and limitations of Rice's Theorem

The fact that Rice's Theorem does not distinguish between the two classes of undecidable languages often results in difficulties for many students studying theory. They have learned that there are two classes of problems that prove to be undecidable – those that are semi-decidable (i.e. RE Non-recursive) and those that are strongly undecidable (i.e. Non-RE). A common query that most instructors in such courses inevitably receives is "How do I use Rice's theorem to prove that a language in Non-RE?" The usual response is "You don't use Rice's theorem to prove Non-RE status you use it to prove that a language is not recursive and thus is undecidable. It says nothing about the property of the language other than it is not decidable."

The response is obviously correct but often leaves students confused as to why Rice's theorem is the godsend they initially believed it to be (and were told it is). If it does not assist them in determining if a language is semi-decidable or strongly undecidable then it appears to be less than optimal for answering the many homework and exam questions they are confronted with which ask them to categorize and

unknown language as decidable, semi-decidable, or strongly undecidable.

Although Rice's Theorem answers the question that is of utmost importance to computer scientists which is "Is the given language/problem recursive/decidable or not?" it does not provide any further information as to the nature of the exact classification of the undecidable problem.

3.2 Corollaries of Rice's Theorem

A corollary of Rice's Theorem implies that if a language L_1 has property P which is Recursively Enumerable and $L_1 \subset L_2$ for some language L_2 then L_2 will also have property P and be Recursively Enumerable. That is, the Recursively Enumerable languages are monotonic due to the lattice structure of RE languages and thus if a language is RE with property P then all supersets of the language are also RE with property P. Rice's Theorem also implies that if a language property P is *not* monotonic, then it is Non-RE [3,9,10].

It must be noted that not all Non-RE properties are non-monotonic. For example, any property P that requires, by definition of P itself, that the language L_1 must have infinite cardinality may not yield a secondary superset language L_2 that does not have the property. For example, the language property L_{inf} = { <M> | L(M) is infinite} or L_{Σ^*} = { <M> | L(M) == Σ^*}. Such properties are clearly monotonic as once satisfied by a language L_1 all supersets of L_1 will also have the property. Such languages require the use of closure properties (e.g. Showing that $\neg L_{inf}$ = { <M> | L(M) is finite } is RE non-recursive) or via diagonalization proofs.

3.3 Altering and extending Rice's Theorem

An examination of Rice's Theorem, along with its corollaries leads us to a relatively minimal alteration to our construction of the machine M_x as implied by Rice's theorem and provides us with a machine that does not have to recognize the empty language as a default in the case that <M> $\notin$ L(M).

3.3.1 Altering the M_x of Rice's construction

The alteration to the traditional Rice approach is simply to have the constructed machine M_x "pre-filter" its inputs and accept them when they satisfy some Turing recognizable language.

In particular, M_x could examine its input before testing the machine encoded by <M> upon its own encoding and accept it if it satisfies the recognition condition. For example, if we wished the machine M_x to recognize the language of all palindromes as a minimal language, then prior to testing the machine M upon its encoding <M> we would examine M_x's own input to see if it was in fact a palindrome and if so accept it and not even bother testing M upon its own encoding. This would result in a constructed machine M_x which either accepted L_{pal} (the language of all palindromes) or possibly a super-set of the language L_{pal} as indicated by the pseudo-code in figure 7. As shown in figure 7, M_x now either accepts L_{pal} , if <M> $\notin$ M, or it accepts the language of all strings (usually represented as Σ^*) if <M> $\in$ L(M).

```
Program M_X( w : input_string )
If ( w is a palindrome)
   accept w
Else If ( <M> ∈ L(M) ) then
   accept w
Else
  reject w
```

Figure 7. Example M_X used to prove that the language of all palindromes is Non-RE.

The key observation here is that the constructed machine M_x as implied by Rice's Theorem always defaults, when <M> $\notin$ L(M), to the empty language $\varnothing$. The new construction allows M_x to recognize/accept any Turing Recognizable language L that is, by definition of the new M_x , a superset of L .

3.3.2 An intuitive understanding of the approach

Just as an operational application of Rice's Theorem requires the student to ask if the empty language has the property in question or not, this analogue approach asks the student to perform the same operation. If the answer to this question is "Yes", then the student need only identify a superset of the empty set that does not have the property P. For example if the property P was "Languages that have fewer than 20 strings", then clearly the empty language has the property and the student need only select a Turing Recognizable language that does not have the property (e.g. Σ^*). Thus M_x either recognizes $\varnothing$ or Σ^*. Since $\varnothing \subset \Sigma^*$ and has property P and Σ^* does not have property P we know that P is a Non-RE property.

The more interesting case is when the empty language $\varnothing$ does *not* have the given property P. It is then that we need to alter our M_x so that it's default language does have the property P in question. As an example, consider L_{pal} (the language consisting of all palindromes only L_{pal}). Clearly, the empty language does not have the property P ($\varnothing \neq L_{pal}$). Now we alter our default language for M_x to be one that does have the property P. In this case, we pre-filter M_x 's input and if it is a palindrome we accept it without ever running machine M upon its own encoding. Now the default language for M_x is the language consisting of any and all palindromes only (L_{pal}). Next, all we need do is identify a superset of this language which does *not* exhibit the property P. Clearly Σ^* does not consist solely of all palindromes and it will suffice. Since $L_{pal} \subset \Sigma^*$ and L_{pal} has the property P but Σ^* does not we know that the property P is Non-RE.

In general, the approach to proving a language to be Non-RE via the construction of M_x can be expressed as shown in figure 8.

1. ***Ask if ∅ ∈ P?***
2. ***If Yes, then design M_x so that it accepts the empty language ∅ if <M> ∉ L(M) and accepts some other language L≠∅ such that L∉P if M accepts <M>.***
3. ***If No, then design M_x so that it accepts one of two languages L_1,L_2 such that $L_1 \in P$ and is accepted if <M> ∉ L(M) and $L_1 \subset L_2$ and $L_2 \notin P$.***

Figure 8. The algorithm for a Rice-analogue to determine if a property of a language is Non-RE.

Since L_1 will always have property P, and RE languages are monotonic, if a superset of L_1 can be found that does not have property P can be found then P cannot be an RE property and is thus Non-RE.

4 Examples, limitations and inability to prove RE languages to be NON-RE

Like Rice's Theorem, this approach is relatively easy to apply. Also like Rice's Theorem, the analogue cannot be used to prove an incorrect result. Although this approach can be used to prove the vast majority of Non-RE languages to, in fact, be Non-RE, there are a handful of languages for which it cannot be directly used.

4.1 Three examples of applying the approach

Consider the language L_{non5} = { <M> | L(M) does not contain at least 5 distinct strings}. We prove that this language is Non-RE by recognizing that the empty language has the property P (cardinality less than 5). So we select our secondary language to be any language whose cardinality is greater than or equal to 5 (e.g. Σ*). Clearly, the empty set is a subset of L_{non5} and our secondary language does not have property P and thus the language property of "having fewer than 5 strings" is Non-RE.

As a second example, consider $L_{\varnothing}$ = { <M> | L(M) == ∅ }. Clearly the empty language has the property. Selecting our secondary language to be Σ* suffices to prove the property to be Non-RE.

Finally, consider the language $L_{non\text{-}CFG}$ = { <M> | L(M) is not a context-free language}. The empty language does *not* have the property P of being non-context-free. This means that we should select our primary/default language to be one that is *not* context-free. We can select L_1 = { $0^N1^N0^N$ | N≥1} which is known not to be a context-free language and thus has the property P. Now we need only identify a superset of L_1 that does *not* have the property of being non-context-free. We can easily select Σ* as it is context-free. This language is context-free and thus is not an element of the set for property P but it clearly contains the subset L_1. Therefore the property of not being context-free is a Non-RE property.

4.2 Limitations to the approach

Consider the Non-RE language L_∞ = { <M> | L(M) is infinite}. If we follow the algorithm we first ask if the empty language has the property (infinite cardinality). Since the empty language is clearly not infinite, we would proceed to step 3. Now we are tasked with selecting two languages L_1 and L_2 such that L_1 has the property (infinite cardinality) and L_2 is a superset of L_1 that does *not* have the property in question. Clearly, any superset of an infinite set must also be infinite and the approach fails.

Although the approach succeeds for the majority of Non-RE properties, the fact that some Non-RE properties *are* monotonic prevents the approach from 100% effectiveness.

4.3 RE languages cannot be proven to be Non-RE

Many students frequently perceive Rice's theorem to be a form of "magic" and wishful thinking. They often believe that they can use this "magical approach" to prove that any language is not recursive. Similarly, students may believe that the approach presented in this paper suffers from a similar weakness. Neither belief is accurate and this can be easily shown.

Consider an attempt to "prove", using this approach, that the language L_5 = {<m> | L(M) contains at least one string whose length is greater than or equal to 5} is Non-RE.

We ask if the empty language has the property and easily determine that it does not have the property of containing a string of at least length 5. So we select some language L_1 as our primary language that does contain at least one such string which we will call *w*. Now, any attempt to identify a super-set of L_1 that does not contain *w* is doomed to fail, by the definition of a super-set. It cannot be done.

As a second example of the failure to incorrectly apply the approach, consider an attempt to prove that the language $L_{non\varnothing}$ = { <M> | L(M) ≠ ∅ }. That is the language of Turing machines whose languages are non-empty. This language is an RE Non-recursive language. Using the approach described, we ask if the empty set has the property of being non-empty and obviously the answer is "No". So we select a language L_1 as our primary language which has the property of being non-empty. Once again, it really does not matter which Turing recognizable language we select as it must be selected so that it contains at least some string *w*. Any attempt to define a language L_2 as a super-set of L_1 is destined to produce a set with at least the string *w* within it and we cannot find the required super-set that does not have the property P of being non-empty.

The fact that RE languages are monotonic makes the task of identifying a super-set of any RE language that does not share the property impossible and will always fail.

5 Conclusion

This paper presents a relatively easy to understand and apply approach to proving the Non-RE status of language properties based upon the fundamental ideas of Turing reductions and the inherent properties of RE languages.

The majority of computer science students enrolled in theory courses are capable of learning the simple process of identifying a primary and secondary language that either exhibit the language property or do not, respectively. The concept of super-sets is usually well understood and requires little in the way of additional mathematical background for most of these students.

Although this approach is not a necessary or sufficient tool for proving the Non-RE status of a given property, it is usually more quickly understood, accepted, and applied than the more standard approaches of using diagonalization arguments, or traditional Turing Reductions, closure properties or introducing and exploring the concept of Index-sets and monotonic functions. It is also a reasonably close analogue to the traditional application of Rice's Theorem as presented in many theory courses and provides students with a level of "comfort" in identifying and discriminating between the three classes of recursive, RE Non-recursive and Non-RE classes of language properties.

6 References

[1] J.E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (1st ed.), Addison-Wesley, 500 pages, 1979.

[2] J.E. Hopcroft and R. Motwani, *Introduction to Automata Theory, Languages, and Computation* (3rd ed.), Pearson, pp. 315-370, 2007.

[3] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation* (3rd ed.), Pearson, pp. 377-419, 2007.

[4] H.G. Rice, *Classes of Recursively Enumerable Sets and their Decision Problems*, Vol. 74, No. 2, pp. 358-366, March 1953.

[5] R. L. Epstein and W. A. Carnielli. *Computability, Computable Functions, Logic, and the Foundations of Mathematics.*(3rd ed.) Advanced Reasoning Forum, 2008.

[6] I.C. Oliveira and W. Carnielli, *The Ricean Objection: An Analogue of Rice's Theorem for First-order Theories*, Logic Journal of the IGPL 16(6): pp. 585-590,(2008).

[7] A. Tarski. *Undecidable Theories*. North-Holland, 1953.

[8] E. Börger, E. Grädel and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, Berlin, 1997.

[9] R. I. Soare. *Recursively Enumerable Sets and Degrees*. Perspectives in Mathematical Logic. Springer-Verlag, 1987. XVIII + 437 pages.

[10] H. Lewis and C. H. Papadimitriou, *Elements of the Theory of Computation* (2nd ed.), Pearson, 1998.

[11] D. C. Kozen, *Theory of Computation* , Springer-Verlag, London, XIV+418 pages, 2006.

[12] R. Murawski. *Decidability vs. undecidability*. Logico-philosophico-historical remarks, *Annales UMCS Informatica AI* 3, pp. 105–117, 2005.

Formal Methods: A First Introduction using Prolog to specify Programming Language Semantics

Lutz Hamel
Department of Computer Science and Statistics
University of Rhode Island
Kingston, Rhode Island, USA
hamel@cs.uri.edu

Abstract

An important fundamental idea in formal methods is that programs are mathematical objects one can reason about. Here we introduce students and developers to these ideas in the context of formal programming language semantics. We use first-order Horn clause logic as implemented by Prolog both as a specification and a proof scripting language. A module we have written facilitates using Prolog as a proof assistant and insures that Prolog implements a sound logic. In order to illustrate our approach we specify the semantics of a small functional language and demonstrate various proof approaches and styles.

1 Introduction

An important fundamental idea in formal methods is that programs are mathematical objects one can reason about [1]. This fundamental idea appears in many areas of software development including algorithm correctness, programming language semantics, compiler correctness, system validation, and system security. For instance, in security sensitive systems one could look at a program as a mathematical object and then formally reason about the safety of that program with respect to some metric. Given the importance of this topic every software developer should be exposed to at least the fundamental concepts and ideas of formal methods [2, 3]. In our curriculum we expose students to ideas in formal methods in the context of formal programming language semantics. Here, programs are structures with corresponding models and the idea is to be able to formally reason about the behavior of programs. The advantage of using programming language semantics as a tool for teaching formal methods is that students have an intuition of what the behavior of a program is and can bring that intuition to the construction of proofs.

After experimenting with many different formalisms including denotational semantics, algebraic semantics, and structural operational semantics we settled on using first-order logic as the formalism for specifying programming language semantics and the corresponding proofs in the context of operational semantic specifications. There are a number of advantages to using first-order logic:

1. It is a formalism most students (and developers) are already familiar with and therefore can concentrate on semantic problems rather than notational issues.
2. It can serve both as a specification language and as a language for constructing proofs.
3. It (or at least the Horn clause subset) is machine executable giving rise to executable specifications and the notion of automatic proof assistants.

We consider the last point extremely important in that students and software developers need to be exposed to automatic theorem proving ideas in the context of formal methods. There exist many first- and higher-order proof assistant systems [4, 5]. However, most of them have difficult notations and concepts of proof construction making them inaccessible for a one or two semester course in formal methods. It turns out that Prolog [6] together with a proof-module that we have developed is more than adequate for an introduction to formal specification of programming language semantics and the construction of the corresponding proofs. Here we describe the proof-module we have developed for Prolog and then we briefly step through an exercise defining the semantics of a small functional programming language together with corresponding proofs.

Using Prolog for the specification of programming language semantics is not new, *e.g.*, [7]. In particular, the work by Christiansen [8] and Mosses [9] stands out because it shares our goal of using Prolog to teach programming language semantics and uses a style of semantic specification similar to the natural semantics style we use in our approach [10]. However, none of the above works takes advantage of Prolog as a theorem prover. The work by Gupta and Pontelli [11] shares our approach by integrating language specification and the corresponding proofs all under the umbrella of logic programming. However, their approach is based on constraint logic programming as opposed to first-order

ISBN: 1-60132-434-0, CSREA Press ©

Horn clause logic [12]. Furthermore, their view of a proof is a single query showing that a particular property holds in their specification. This is very different from our view of a proof as a program over the meta-language of Prolog including queries, assertions, and retractions. It was paramount for us to stay in the confines of first-order Horn clause logic in order to satisfy our teaching goal. As far as we are aware using Prolog as a proof assistant in the context formal specifications is novel.

The remainder of this paper is structured as follows. Section 2 discusses Prolog as a theorem prover or more precisely as an automatic proof assistant. In Section 3 we discuss our approach to the semantic specification of programming languages using Prolog. Section 4 discusses proofs. As mentioned above, we view proofs as programs over the meta language of Prolog and here we showcase a number of different proof techniques applicable to semantic specifications. Finally, in Section 5 we present conclusions and further work.

2 Prolog as a Theorem Prover

2.1 The Logic

The first-order Horn clause logic Prolog implements is perhaps one of the simplest machine executable, Turing complete logics. This makes Prolog attractive as a specification language since its learning curve is not as steep as other logic implementations. Under the following considerations Prolog implements a sound but incomplete logic [13, 14]:

1. The unification algorithm implements the occurrence-check – Most Prologs omit the required occurrence-check for efficiency reasons. However, some Prolog systems such as SWI-Prolog [6] make the occurrence-check user selectable.
2. The proof search strategy is a depth-first search of the refutation proof tree – This is the standard implementation of the search strategy for Prolog due to efficiency reasons.
3. Only ground terms are negated in rule bodies and proof goals.

Our Prolog proof-module for SWI Prolog insures that the three conditions above are met.

For the last condition above it can be shown that under certain circumstances deduction will flounder when negation of non-ground terms is involved [13, 14]. Our module circumvents this problem by introducing a new negation predicate `neg/1` which checks whether the negated term is ground or not:

```
neg(G) :- ground(G),!,call(not(G)).
neg(_) :- throw('term is not ground').
```

Note that it is necessary to abort deduction if a non-ground term is found since simple failure is interpreted as a negation result. The following is a classic example where deduction flounders under negation [14],

```
on_top(X) :- not(blocked(X)).
blocked(X) :- on(Y,X).
on(a,b).
```

Now given the query of 'do there exist any objects Q on top?' Prolog returns the incorrect answer 'false',

```
?- on_top(Q).
false
?-
```

However, it does produce the correct result given the query,

```
?- on_top(a).
true
?-
```

Now, replacing the first line in the program above with the line which includes our new negation predicate,

```
on_top(X) :- neg(blocked(X)).
```

prevents Prolog from performing unsound deductions and will abort the computation.

```
?- on_top(Q).
ERROR: Unhandled exception: term is not ground
?-
```

And it still does produce the correct result given the query,

```
?- on_top(a).
true
?-
```

Even though the incompleteness of the logic is disconcerting it does not have as much an impact on our proofs as one might think due to the fact that we use Prolog as a *proof assistant* along the lines of Coq [4] and Isabelle [5][1] where proofs are composed of many small steps each verified by Prolog rather than a *fully* automatic theorem prover where the system is tasked with also finding the proof steps. That is, we view proofs as programs over the meta-language of Prolog including queries, assertions, and retractions. We refer to these programs as *proof scores*. It is our experience that it is highly unlikely to encounter problems with incompleteness of the logic in this approach. Even if one did, the problems are easily remedied by either reordering the predicates in a proof step (in the case of an infinite search) or including additional lemmas in the proof to work around incompleteness problems due to the restriction of negation to ground terms only.

2.2 Notation

Our style of specification of programming language semantics was inspired by the natural semantics of Kahn [10]. The overall structure of a semantic rule is as follows,

<context>:: <syntax> –» <value> :- <conditions>

The intended interpretation of these rules is: given a context, a piece of abstract syntax is mapped into a semantic value if the conditions hold. In Prolog the symbol `:-` represents the keyword `if`. The rules can be abbreviated to,

<syntax> –» <value> :- <conditions>

if no context is required by the rule. Our module defines this notation to make specifications and proofs more readable.

[1] Neither Coq nor Isabelle is complete due to their use of higher order logics.

2.3 Universally Quantified Queries

Queries in Prolog allow only for existentially quantified variables. However, when constructing proofs it is often necessary to have queries over universally quantified variables. We can simulate universally quantified variables in queries using the following rule from quantification theory [15]:

$$\frac{q \in U \quad P(q)}{\therefore \forall x \in U[P(x)]}$$

If a predicate P is true for an arbitrary object q in some domain U it follows that the predicate is true for all objects in that domain. We can use this to pose universally quantified queries in our semantics such as,

```
?- s:: plus(1,1) -->> 2.
```

where we can interpret `s` as a constant representing some state and the query poses the question whether in some state `s` the operation `plus(1,1)` evaluates to the value 2. If the query is successful then we can use the above quantification rule to conclude that the query holds for all possible states. Since this kind of reasoning is always possible we abuse notation slightly and interpret symbolic constants in queries as universally quantified variables unless it is obvious from context that a particular constant is meant, for example, `s0` for the initial state.

2.4 The `xis/2` Predicate

Prolog implements a machine executable logic. Given this we are interested in using programming language specifications both as executable prototypes as well as for proving properties of the specified language. When we use a specification as a prototype we want to appeal to Prolog's efficiency as a programming language which includes the efficient evaluation of arithmetic expressions. When we want to perform proofs we appeal to the declarative side of Prolog [14]. It turns out that these two notions clash in the evaluation of arithmetic expressions using the `is/2` predicate. The `is` predicate is very efficient for evaluating arithmetic expressions,

```
?- X is 1 + 1.
X = 2.
```

However, when performing proofs it is often necessary to write arithmetic expressions involving universally quantified variables,

```
X is k + 1.
```

and this leads to problems because `is` does not know how to handle these quantities,

```
?- X is k + 1.
ERROR: is/2: Arithmetic: 'k/0' is not a function
```

In order to accommodate proofs involving universally quantified variables our module implements the `xis/2` predicate (eXtended `is`) which behaves just like `is` but allows universally quantified variables,

```
?- X xis k + 1.
X = k+1.
```

It does perform partial evaluation of the expressions where possible,

```
?- X xis 0, Y xis k + 3 * cos(X).
X = 0,
Y = k+3.0.
```

2.5 Additional Predicates

In order to make proofs more readable and easier to follow at runtime our module defines some additional predicates. These predicates do not add new meta-language functionality to Prolog but rather act as wrappers for existing functionality that provide better self-documentation of proofs and a better runtime trace. Among the newly defined predicates are:

`assume/1` – this is the same as `asserta/1`.

`remove/1` – this is the same as `retract/1`.

`show/1` – this is the same as a Prolog query.

Each of these predicates preserves the original functionality but outputs additional information when executed. Here is an example of a very simple (and perhaps silly) proof score:

```
:- consult('preamble.pl').
:- >>> 'assume the commutative property'.
:- >>> 'of integer addition'.
:- assume equiv(A+B,B+A).

:- >>> 'show that expressions X and Y'.
:- >>> 'are related by commutativity'.
:- show
     X xis a + b,
     Y xis b + a,
     equiv(X,Y).
```

Here is the runtime trace of this proof score,

```
%   xis.pl compiled 0.00 sec, 33 clauses
%  preamble.pl compiled 0.00 sec, 45 clauses
>>> assume the commutative property
>>> of integer addition
    Assuming: equiv(_G1202+_G1203,_G1203+_G1202)
>>> show that expressions X and Y
>>> are related by commutativity
    Showing:
    _G1214 xis a+b,
    _G1262 xis b+a,
    equiv(_G1214,_G1262)
% proof-simple.pl compiled 0.03 sec, 1,343 clauses
```

Note that in queries Prolog replaces variable names with internally generated unique names. In the case above, for example, the variable `A` is replaced by `_G1202`. Also, the `consult` predicate at the beginning of the proof score loads our module `preamble.pl`. Also note the "executable" comments.

3 Semantic Specifications

In order to illustrate the use of our semantic rules we will specify the semantics of a small functional language inspired by Winskel's REC language [16]. The abstract syntax for this language is shown in Figure 1 with the concrete syntax shown in brackets.

```
E ::= X
    | I
    | mult(E,E)            [E * E]
    | plus(E,E)            [E + E]
    | minus(E,E)           [E - E]
    | if(B,E,E)            [if B then E else E end]
    | let(X,E,E)           [let X = E in E end]
    | letrec(F,X,E,E)      [let rec F X => E in E end]
    | fn(X,E)              [fn X => E]
    | apply(E,E)           [E E]

B ::= true
    | false
    | le(E,E)              [E <= E]
    | eq(E,E)              [E == E]
    | not(E)               [not E]

I ::= <any integer digit>
X ::= <any variable name>
F ::= <any function name>
```

Figure 1: The abstract syntax of a small functional language.

As usual, we have to give at least one semantic rule for each syntactic unit in the grammar. The distinguishing feature of the semantics for this language is that it has a declaration environment for functions we call `D` and a binding environment for variables we call `S`. Therefore, a state in our semantics is a pair consisting of a declaration environment and a binding environment, *e.g.* `(D,S)`. We start our discussion by giving the rule for the arithmetic operator `mult`,

```
(D,S):: mult(E1,E2) -->> V :-
       (D,S):: E1 -->> V1,
       (D,S):: E2 -->> V2,
        V xis V1 * V2,!.
```

This rule can be paraphrased as follows:

> In the context of state `(D,S)`, the operator `mult(E1,E2)` with subexpressions `E1` and `E2` evaluates to the value `V` if under state `(D,S)` the subexpressions `E1` and `E2` evaluate to the values `V1` and `V2`, respectively, and the integer multiplication of `V1` and `V2` is the value `V`.

In Prolog commas represent the boolean connective `and`. Also, in Prolog variables start with a capital letter, that means `E1`, `E2`, `S`, *etc.* are all variables or more precisely meta-variables, *i.e.*, variables of the specification language. Also noteworthy is the cut `(!)` at the end of the rule. We can interpret this cut in one of two ways. First, from a procedural point of view each semantic rule constitutes a state transition and once a state transition was made it is not allowed to be reversed. Second, from a declarative point of view the set of semantic rules constitute an inductively defined set of rules. Therefore, once it has been shown that a rule has been successfully applied to a piece of syntax all other branches of the proof tree can be safely pruned because they will not contain another success. This holds even if there are multiple rules for a particular syntactic unit because those rules will be mutually exclusive (*e.g.*, see the if-then-else rules).

The rules for `plus` and `minus` are analogous to the rule for `mult`. Next we look at integer constants and variables. The rule,

```
I -->> I :- is_int(I),!.
```

states that integer constants are treated as integer values regardless of state. The following rules interpret variables in expressions. The first rule gives an interpretation to function variables and the second rule to variables that range over integer values,

```
(D,_):: F -->> [[X,E,S]] :-
        is_var(F),
        lookup(F,D,[[X,E,S]]),!.

(_,S):: X -->> V :-
        is_var(X),
        lookup(X,S,V),!.
```

The first rule looks up the name `F` in the function declaration environment `D` and returns the closure of a function which incorporates the formal parameter, the function body, and the binding environment in which the function was defined. We denote closures with a double bracket notation, `[[ ]]`. The second rule looks up the variable `X` in the binding environment `S` and returns the bound integer value. The predicate `is_var` insures that the variable names conform to the lexical rules. This predicate is not strictly necessary but here we are dealing with abstract syntax and we do not have a parser enforcing lexical rules. The `lookup` predicate is an auxiliary predicate defined as part of our semantics. The underscore in the rules represents an anonymous variable meaning that the corresponding structure is matched but ignored by the rule. Next, the if expression has its usual interpretation,

```
(D,S):: if(B,E,_) -->> V :-
        (D,S):: B -->> true,
        (D,S):: E -->> V,!.

(D,S):: if(B,_,E) -->> V :-
        (D,S):: B -->> false,
        (D,S):: E -->> V,!.
```

Here the first rule states that if the boolean expression evaluates to the value true within the context of state `(D,S)` then the first expression is evaluated. The second rule states that otherwise the second expression is evaluated. Let expressions allow us to bind values to variables,

```
(D,S):: let(X,E1,E2) -->> V :-
                is_var(X),
                (D,S):: E1 -->> V1,
        (D,[(X,V1)|S]):: E2  -->> V,!.
```

Here we first evaluate expression `E1` under the original state `(D,S)`. Once we have the corresponding value `V1` we extend the original binding environment `S` with the binding term `(X,V1)` making use of Prolog's list manipulation abilities and evaluate the expression `E2` under this new extended state. The resulting value `V` is the return value of the overall let expression. A special case of the let expression is the let-rec expression which allows us to define recursive functions,

```
(D,S):: letrec(F,X,E1,E2) -->> V :-
                is_var(F),
                is_var(X),
         ([(F,[[X,E1,S]])|D],S):: E2  -->> V,!.
```

The let-rec expression computes the function closure and associates the closure with the function name `F` in the function declaration environment `D`. The expression `E2` is then evaluated in this extended state.

Our programming language also supports anonymous functions envisioned in the style of ML [17]. In the abstract syntax this is denoted by the operator `fn`. As before, the semantic value of a function definition is the closure of the function,

```
(_,S):: fn(X,E) -->> [[X,E,S]] :- is_var(X),!.
```

Finally, we define function application as follows,

```
(D,S):: apply(E1,E2) -->> V :-
                        (D,S):: E1 -->> [[X,E,Sfn]],
                        (D,S):: E2 -->> V2,
              (D,[(X,V2)|Sfn]):: E -->> V,!.
```

Here we see that in order for function applications to make sense the first expression `E1` has to evaluate to a function closure. We then evaluate the second expression `E2` and its value `V2` is used to create a binding term `(X,V2)` where `X` is the formal parameter of the function. This binding term is used to extend the function binding environment `Sfn` and the body of the function `E` is evaluated under this extended state.

The semantics of boolean expressions can be specified analogously to the arithmetic expression with the big difference of course that we only have two constant values: `true` and `false`. A complete listing of all the semantic specification rules is available from the authors website.

4 Proofs

Everything in Prolog is a proof – in particular, running a logic program in Prolog is a proof. However, here we are interested in Prolog as a proof assistant in order to prove characteristics of our language specifications. Our view of proofs as programs over the meta language of Prolog seems to be novel and we explore this here. We explore three types of proofs:

- Tests - which are proofs over a particular input-output pair of a program.
- Proofs of language properties - these proofs examine features of the language such as program equivalence.
- Program correctness proofs - proofs whether a program conforms to a given requirement or not.

Here we take a look at each of these proof categories.

4.1 Tests

In testing we are interested in the behavior of language features and want to show that a certain feature behaves as expected given some particular input value. In Prolog we accomplish this by setting up a proof that relates an input to a program to its expected outcome. The following is a simple proof for integer multiplication in our functional programming language assuming that the language definition has been loaded,

```
?- show (d,[(x,10)|s]):: mult(x,10) -->> 100.
    Showing: (d,[ (x,10)|s])::mult(x,10)-->>100
true.
```

We can paraphrase this proof as follows,

> Show that for all declaration environments `d` and all binding environments `s` that contain the binding term `(x,10)` the code snippet `mult(x,10)` evaluates to the value 100.

In order to illustrate how these tests can be used to explore features let us take a look at function calls. Here is a more ambitious test proof regarding function calls,

```
:- consult('functional-rec-sem.pl').
:- assume program
       let(inc,
           fn(x,plus(x,1)),
           apply(inc,1)).
:- >>> 'we have for all states (d,s), (d,s):: P -->> 2'.
:- show
        program P,
        (d,s):: P -->> 2.
```

The above program can be rewritten in concrete syntax as follows,

```
let inc = (fn x => x + 1) in inc 1 end
```

The actual test checks whether for all possible states the program evaluates to the value 2. Here is the corresponding runtime trace of the proof score assuming that the proof score is called 'proof-inc.pl',

```
?- consult('proof-inc.pl').
%    xis.pl compiled 0.00 sec, 33 clauses
%   preamble.pl compiled 0.00 sec, 45 clauses
%  functional-rec-sem.pl compiled 0.01 sec, 68 clauses
    Assuming: program let(inc,fn(x,plus(x,1)),apply(inc,1))
>>> we have for all states (d,s), (d,s):: P -->> 2
    Showing: program _G117, (d,s)::_G117-->>2
% proof-inc.pl compiled 0.01 sec, 72 clauses
true.
```

We can also experiment with the higher-order nature of our functional programming language using currying,

```
:- >>> 'Higher order functions: curried plus'.
:- assume program
       let(add,
           fn(x,
              fn(y,plus(x,y))),
           apply(apply(add,1),1)).
:- >>> 'we have for all states (d,s), (d,s):: P -->> 2'.
:- show
        program P,
        (d,s):: P -->> 2.
```

In terms of concrete syntax the above program is written as:

```
let add = (fn x => (fn y => x + y)) in add 1 1 end
```

4.2 Proofs of Language Properties

In order to prove properties of a programming language it is convenient to define the notion of program equivalence,

$$p_1 \sim p_2 \text{ iff } \forall s, \exists v_1, v_2 [s :: p_1 \rightarrow v_1 \wedge s :: p_2 \rightarrow v_2 \wedge v_1 = v_2]$$

That is, two programs p_1 and p_2 are equivalent if and only if under all states s they produce the same semantic value. We can use this to prove that the multiplication operator in our language is commutative. Looking at the semantic rule for multiplication defined above it is clear that commutativity follows directly from the commutativity of integer multiplication but it is still nice to actually prove that this is so,

```
:- >>> 'Assume that we have expressions a and b'.
:- assume (d,s):: a -->> va.
:- assume (d,s):: b -->> vb.

:- >>> 'Integer multiplication is commutative'.
:- assume equiv(A*B,B*A).

:- show
     (d,s):: mult(a,b) -->> V1,
     (d,s):: mult(b,a) -->> V2,
     equiv(V1,V2).
```

Next we prove that our functional language implements by-value parameter passing. We show this by proving that function application is equivalent to an appropriate let-expression,

```
:- >>> 'By-value parameter passing'.

:- assume          (d,s):: a -->> va.
:- assume (d,[(x,va)|s]):: e(x) -->> ve.

:- show
        (d,s):: let(x,a,e(x)) -->> V1,
        (d,s):: apply(fn(x,e(x)),a) -->> V2,
        V1=V2.
```

The proof itself is straightforward with perhaps the exception of the second assumption which states that any expression `e` parameterized over the variable `x` evaluates to the value `ve` under some state whose binding environment `s` contains the variable binding `(x,va)`.

The following is a proof that in our functional language without function application all programs terminate, *i.e.*, always produce a value. The proof is by structural induction over the expressions,

```
:- >>> 'Base cases:'.

:- >>> 'Variables'.
:- >>> 'Assume that states are finite'.
:- assume lookup(x,s,vx).
:- show (d,s):: x -->> vx.
:- remove lookup(x,s,vx).

:- >>> 'Constants'.
:- assume is_int(n).
:- show (d,s):: n -->> n.
:- remove is_int(n).

:- >>> 'anonymous function definitions'.
:- assume is_var(x).
:- show (d,s):: fn(x,e) -->> [[x,e,s]].
:- remove is_var(x).

:- >>> 'Inductive cases'.

:- >>> 'Operators'.
:- >>> 'mult'.
:- assume (d,s):: a -->> va.
:- assume (d,s):: b -->> vb.
:- show (d,s):: mult(a,b) -->> va*vb.
:- remove (d,s):: a -->> va.
:- remove (d,s):: b -->> vb.

:- >>> 'the remaining operators and boolean'.
:- >>> 'expressions can be proved similarly'.

:- >>> 'programming constructs'.
:- >>> 'let-expression'.
:- assume (d,s):: a -->> va.
:- assume (d,[(x,va)|s]):: e(x) -->> ve.
:- show (d,s):: let(x,a,e(x)) -->> ve.
:- remove (d,s):: a -->> va.
:- remove (d,[(x,va)|s]):: e(x) -->> ve.

:- >>> 'similarly for the let-rec expression'.

:- >>> 'if-expression with case analysis'.
:- assume (d,s):: e1 -->> v1.
:- assume (d,s):: e2 -->> v2.

:- assume (d,s):: b -->> true.
:- show (d,s):: if(b,e1,e2) -->> v1.
:- remove (d,s):: b -->> true.

:- assume (d,s):: b -->> false.
:- show (d,s):: if(b,e1,e2) -->> v2.
:- remove (d,s):: b -->> false.
:- remove (d,s):: e1 -->> v1.
:- remove (d,s):: e2 -->> v2.
```

The structural induction argument as encoded by this proof score is pretty straight forward. Perhaps the only surprising aspects are the 'remove' statements which remove assumptions from the Prolog database. They are necessary in order to prevent assumptions from one step of the proof to "bleed" into another step of the proof.

4.3 Program Correctness Proofs

Program correctness proofs are very similar to testing as discussed above with the exception that we want to show that the program behaves as expected for *all* inputs. Here we use techniques described in [18] and [19].

We start with the correctness proof a program that computes the maximum of two values. The proof makes use of the Prolog built-in predicate `max/2` as a model for the computation of our program.

```
:- >>> 'show that program'.
:- >>> ' P = "let(z,if(le(n,m),m,n),z)"'.
:- >>> 'computes the maximum of'.
:- >>> 'the values assigned to m and n'.

:- assume program let(z,if(le(n,m),m,n),z).

:- >>> 'assume values for m and n'.
:- assume (d,s):: m -->> vm.
:- assume (d,s):: n -->> vn.

:- >>> 'case analysis on values vm and vn'.
:- >>> 'case vm = max(vm,vn)'.
:- assume vm xis max(vm,vn).
:- >>> 'this implies that'.
:- assume true xis (vn =< vm).
:- show
      program P,
      (d,s):: P -->> vm.

:- remove vm xis max(vm,vn).
:- remove true xis (vn =< vm).

:- >>> 'case vn = max(vm,vn)'.
:- assume vn xis max(vm,vn).
:- >>> 'this implies that'.
:- assume false xis (vn =< vm).
:- show
      program P,
      (d,s):: P -->> vn.

:- remove vn xis max(vm,vn).
:- remove false xis (vn =< vm).
```

The proof performs a case analysis on the values of `m` and `n` and shows that in each case our program evaluates to the correct value for all possible states `s`.

Our next proof is the correctness proof of the factorial function,

```
let
   rec fact x => if x == 1 then 1 else x * fact(x-1) end
in
   fact(1)
end
```

Here is the proof,

```
:- >>> 'Factorial: show that program P:'.
:- assume program
        letrec(fact,
```

```
              x,
              if(eq(x,1),
                 1,
                 mult(x,
                      apply(fact,
                            minus(x,1)))),
       apply(fact,i)).
:- >>> 'is correct for all inputs i > 0'.

:- >>> 'proof by induction on i'.

:- >>> 'base case: i=1'.
:- assume i -->> 1.
:- show
        program P,
       (d,s):: P -->> 1.

:- >>> 'inductive step: i=n'.
:- assume i -->> n.
:- assume false xis n==1.
:- >>> 'inductive hypothesis:'.
:- assume
   apply(fact,minus(x,1)) -->> factorial(n-1).

:- show
        program P,
        (d,s):: P -->> n*factorial(n-1).
```

The proof is by induction over the input to the `fact` function. As a model for the computation we use the factorial operator defined in the standard recursive way for $k > 0$,

$$\text{factorial}(k) = \begin{cases} 1 & \text{if } k = 1 \\ k * \text{factorial}(k-1) & \text{otherwise} \end{cases}$$

5 Conclusions

Every software developer should be exposed to the fundamental idea in formal methods that programs are mathematical objects one can reason about. We introduce this idea in the context of formal programming language semantics. Here, programs are structures with corresponding models and the idea is to be able to formally reason about the behavior of programs. We have shown that the first-order Horn clause logic as implemented by Prolog is a suitable framework to introduce these ideas. Using the specification of a small functional language we have shown that a variety of proof types and styles can be implemented using Prolog as a proof assistant, from simple implication based proofs to induction based arguments. In our view proofs are programs over the metalanguage of Prolog and our custom module assists in writing these proofs. Our module also insures that Prolog deduction is sound and allows the use of universally quantified variables in proofs. The advantages of using Prolog is that it is a straightforward language to learn and the underlying logic is likely a formalism most students and software developers have already encountered.

In the future we interested in developing bisimulation and co-inductive techniques using Prolog which would prove useful when proving compilers and translators correct.

This paper is dedicated to Angel.

References

[1] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys (CSUR)*, vol. 28, no. 4, pp. 626–643, 1996.

[2] S. Skevoulis and V. Makarov, "Integrating formal methods tools into undergraduate computer science curriculum," in *Frontiers in Education Conference, 36th Annual*, pp. 1–6, IEEE, 2006.

[3] A. Zamansky and E. Farchi, "Exploring the role of logic and formal methods in information systems education," in *Software Engineering and Formal Methods*, pp. 68–74, Springer, 2015.

[4] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, 2004.

[5] L. C. Paulson, *Isabelle: A generic theorem prover*, vol. 828. Springer, 1994.

[6] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[7] B. R. Bryant and A. Pan, "Rapid prototyping of programming language semantics using prolog," in *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pp. 439–446, IEEE, 1989.

[8] H. Christiansen, "Using prolog as metalanguage for teaching programming language concepts," *Issues in Information Technology, EXIT, Warszawa*, pp. 59–82, 2000.

[9] P. D. Mosses, "Modular structural operational semantics," *The Journal of Logic and Algebraic Programming*, vol. 60, pp. 195–228, 2004.

[10] G. Kahn, "Natural semantics," in *4th Annual Symposium on Theoretical Aspects of Computer Science (STACS 87)*, pp. 22–39, Springer-Verlag, 1987.

[11] G. Gupta and E. Pontelli, "Specification, implementation, and verification of domain specific languages: a logic programming-based approach," in *Computational Logic: Logic Programming and Beyond*, pp. 211–239, Springer, 2002.

[12] T. Swift and D. S. Warren, "Xsb: Extending prolog with tabled logic programming," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 157–187, 2012.

[13] J. Lloyd, *Foundations of Logic Programming*. Berlin: Springer-Verlag, 1987.

[14] U. Nilsson and J. Małuszyński, *Logic, programming and Prolog*. Wiley Chichester, 1990.

[15] I. Copi, "Introduction to logic (6th ed)," 1982.

[16] G. Winskel, *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[17] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[18] R. Bird *et al.*, *Introduction to functional programming using Haskell*, vol. 2. Prentice Hall Europe London, 1998.

[19] J. A. Goguen and G. Malcolm, *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

SESSION

SURVEYS AND NEW STUDIES

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

Reframing the Gender Gap in Computer Science as a Solvable Problem

Nazli Hardy [1], **Emmali Montgomery**[1]
[1]Computer Science, Millersville University, Millersville, PA, USA

Abstract - *The alarmingly low numbers of women pursuing Computer Science degrees in the United States continues to be significant concern. Because of such low participation, related organizations have been considering potential ways to integrate more women; however, it is the underlying general causes and their sources of the decline that need to be considered methodically from various contexts. In understanding the causes, we can redirect the efforts as a solvable problem.*

Keywords: *Women in Computer Science, Gender Gap in Computer Science, Attracting Women to Computer Science*

1 Introduction

By 2024, there will be half a million Computer Science related jobs that need filled according to the Bureau of Labor and Statistics, and at the current graduating rate of Computer Scientists, the supply is much lower than the demand [1]. Companies are so desperate for employees to fill those positions that many are hiring fresh college graduates or attempting to create more enticing internship programs. Surprisingly, women have been found to outnumber men in Universities, yet very few are pursuing careers in Computer Science. The National Bureau of Economic Research states "in 2003, there were 1.35 females for every male who graduated from a four-year college" [2]. Comparing this to Randal Olson's findings that show less than 18% of Computer Science degrees are earned by women begs the question of what is steering women away from technology [3].

A 2014 Google study [4] found that the four (controllable) key factors in the decrease of numbers are 1) social encouragement, 2) self-perception, 3) academic exposure, and 4) career perception. In addition, the study found peripheral roles (uncontrollable factors) that also have a negative influence in the pursuit of a Computer Science degree by women. These include a) ethnicity b) family income, c) and parental occupation.

For women (and men) studying and working in the field of Computer Science, these numbers and accompanying figures are not new. Having had the experience of being or seeing only a handful of women in a) undergraduate and graduate classes (which were taught by mostly men) and b) in the workplace, the numbers while unsurprising continue to be problematic.

Through training, Computer Scientists are taught to problem solve and to do so in the most efficient manner possible. To Computer Scientists, the world of possibilities lies in the problems that can be solved, logically and systematically; however, to people outside the realm of technology, it is not clear at all what Computer Scientists do or what Computer Science is. That is a problem because the vague understanding of what computer science is and what computer scientists do feed directly to the four key factors identified in the Google study.

2 The Career Landscape for Women and Motivating Factors

In 2014, the fields in which there were more than 75% women employed in the US [4] were secretaries, elementary & middle school teachers, nurses, psychiatric and home health aides, receptionists, office clerks, maids and housekeeping, social workers, secondary school teachers, personal care aides, waitresses, teaching assistants, preschool and kindergarten teachers. A 2012 Wall Street Journal [5] indicated that "women account for a third of the nation's lawyers and doctors, a major and positive shift from a generation ago."

The presence and "popularity" for women in the aforementioned careers could be functions of opportunities available, and also the built-in bias in recruitment in a male dominated workforce over the years. But from a pragmatic perspective, we can consider what these occupations have in common.

1) There is a clear perception regarding the aforementioned careers. People and media have a comfortable summary understanding of what teachers do, what nurses and doctors do, what administrative assistants do, what lawyers do. Women have been pigeon-holed into these professions.

2) There is a sense of manageability of hours which would allow women to "have it all" (an income, a career, independence, while also being able to nurture a functional household/ family.)

3) Counselors, parents, and mentors are able to give social, academic, professional exposure, guidance

and encouragement to students interested in pursuing these careers.

4) Each of the professions is useful and necessary in everyday life, and they involve meaningful social and human interaction.

3 The Solvable Parts of the Problem

Part I: A study on adolescent (teenagers indicative of a pre-college age-group) girls has shown that "girls reported greater likability and similarity to the self for women in appearance-focused occupations compared with women in non-appearance-focused occupations" [10]. The girls were shown photographs of appearance-based career women (model Heidi Klum and actress Jennifer Anniston) and non-appearance based career women (CEO Carly Fiorina, and military pilot Sarah Deal Burrow). The girls rated women in appearance-focused photos as more competent than the other women. However, the study also found that the same teenage girls found CEOs and military pilots to be better role models. The research also concluded from the findings that "girls know they should look up to female doctors and scientists, but they also know that women in appearance-focused jobs get rewarded by society. It is therefore reasonable to think they would prefer women in those jobs." What is encouraging is that the finding also showed that there is an "interest and hunger for a more diverse image of working women in media and advertising."

An additional study found "that image search results for occupations slightly exaggerate gender stereotypes and portray the minority gender for an occupational less professionally" [6]. To address these concerns LeanIn.org and Getty Images have collaborated to create and curate and present the "Lean In Collection," a library of images devoted to the powerful depiction of women, girls, and people who support them. The pictures in this collaborative collection are geared to depict images of female leadership, and equal partnership in contemporary work and life. [7]

Part II: In a survey conducted by the authors of this paper, we identify 1) the lack of clear understanding of what computer scientists do, and 2) the lack of clear perception of the field of computer science, as additional components to the low enrollment of women in computer science in the United States.

4 Methodology

Part I: To assess, compare, and contrast the images of the profession of Computer Science and the image of women in Computer Science, simple searches were carried out on Google. In line with the Google study [4], the objective was to gauge the perception and exposure of Computer Science and Computer Scientists that are projected by online media to precollege female students, and their circle of influence (counselors, peers, parents).

Part II: The authors conducted a survey of a wide range of people as a means of gauging their perceptions and "un-researched" understanding of what computer scientists do, and what computer science is, in general. We asked each respondent to give us their immediate and un-researched answers.

5 Results

Part I:
Googling the words "Computer Science" gives us the following image (Figure 1).

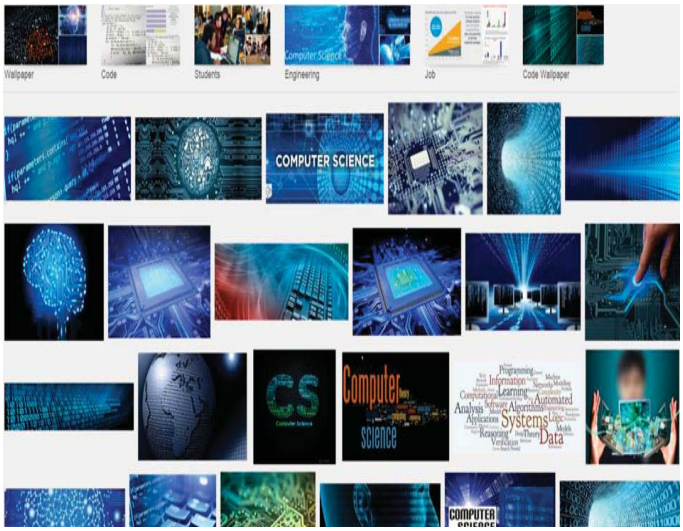


Fig 1: the images retrieved on 3/21/16 upon googling "computer science"

Googling the words "women Computer Science" gives us the following picture (Figure 2).

Fig 2: the images retrieved on 3/21/16 upon googling "women in computer science"

These images are a visual indication of how computer science is perceived and how that perception is further propagated in the media, and funneled through and accepted by high school students, their counselors, and guardians.

All images from the first set were advertisement styled with a plethora of 1s, 0s, and various code bits plastered on objects that generally do not define Computer Science at all. The closest image was one that included a keyboard. This indicates that society, especially those instilled with the job of promoting Computer Science, have a narrow understanding of the field.

The second set shows a lot of promotional material for women, further showing that there is a lack of women and support for the field. Beyond that, it is just people looking at a computer. There is nothing showing what the actual field can accomplish, or that Computer Science and coding is a big part of everyday life. There is nothing illustrating how code affects people and that it is in almost everything we handle in this decade.

The images of "computer science" and a "computer scientist" visually indicate:

1) Computer Scientists and Computer Science are not defined clearly.
2) there is little relevance or connection with the pictures to everyday lives.
3) there is no depiction of social or human interaction.
4) Computer Science is abstract and non-dynamic field.
5) Computer Scientists do little beyond sitting around a computer.
6) there is a widening gap/ problems in the recruitment of women in computer science.

As a contrast to the images for "Computer Science" and "Computer Scientist", the images for the careers that women pursue, or in which there is a majority of women, showcase a different narrative. Figures 3, 4, and 5 represent Google images for elementary school teacher, receptionist, and health care respectively.

Fi 3: the images retrieved on 3/21/16 upon googling "elementary school teacher" (most of us have had one – we know what they do, they made an impression on us)

Fig 4: the images retrieved on 3/21/16 upon googling "receptionist" (images of attractive women, which is important to people)

Fig 5: the images retrieved on 3/21/16 upon googling "health care" (important to all os us, meaningful, we have interaction with someone in the health field social interaction, making a difference, prestige)

Part II
A random group of college educated, non-computer scientists were asked the following questions a) what do you think a Computer Scientist does and b) what exactly is Computer Science. (The complete set of responses is in the Appendix.)

6. Analysis and Discussion

Part I: The images portray computer science to be a counter to the images portrayed for professions that women typically choose. The images are impersonal, abstract, (Fig 1) static, and do not illustrate the varied functions of computer scientists (Fig 2). In addition, the images generally do not demonstrate any women in leadership roles, nor do they exemplify meaningful human or social interaction. The messages for computer science and women in computer science (Fig 1, 2) are in contrast to the more dynamic images depicting "happy" women interacting with others in careers that appear to be meaningful (Fig 3, 4, 5) like elementary teachers, and health care professionals. Adolescent girls are self-conscience and want a career in which women are perceived to be attractive and thus likable and rewarded by society [10]. The abstract images portraying computer science do not allow adolescent girls and their circle of influence (high school counselors, parents. guardians) to connect with the career that is known to be "male-dominated."

Part II: The survey carried out by the authors confirms that there is not a clear or summary understanding of what computer scientists do, or what computer science is. There is a specific connection to coding, and computers, but there is not a comprehensive scope of understanding of the broad spectrum of meaningful contributions that is directly influenced by computer science and computer scientists.

7. Conclusions

To attract women to Computer Science, a field we know to be dynamic, progressive, meaningful, broad, flexible, and impactful, it is necessary to:

1) define clearly what computer science is and what computer scientists do, so that it can be conveyed to girls in middle and high school

2) define clearly to high school counselors, and the general population what computer science is and what computer scientists do, so that they would be articulate the field to potential and interested students

3) work with organizations like LeanIn.org and Getty Images, to create and propagate dynamic images of actual female computer scientists carrying out jobs in computer science, especially ones in which they are engaging with others (e.g. professors, filed researchers, experts speaking at conferences, working on rockets, building robots in a team)

4) showcase computer science as the field that creates devices and programs that girls enjoy using and find useful and meaningful to use (e.g. iTunes, Google, apps, FitBit, computerized medical devices etc.)

5) redefine the field as one that, due to its pervasive nature, allows women to pursue any area of interest while also allowing them to have the family life and social time that leads to a balanced life.

6) invest in conferences that have females Computer Scientists as role models and presenters. A model for such a conference is the Women in Math and Science Conference at Millersville University [8].

7) invest in courses (at both the High School and College level) that give Freshman an example-based understanding of what Computer Science is and what Computer Scientists have done and are doing [9].

8 References

[1] "Computer and Information Technology Occupations." *Bureau of Labor Statistics.* Retrieved on 5/9/2016. http://www.bls.gov/ooh/computer-and-information-technology/home.htm *retrieved on 3/21/16*

[2] "Why Do Women Outnumber Men in College?" *the National Bureau of Economic Research*. http://www.nber.org/digest/jan07/w12139.html *retrieved on 5/9/2016.*

[3] Olson, Randal. "Percentage of Bachelors Degrees Conferred to Women, by Major (1970- 2012)." *Wordpress*. http://www.randalolson.com/2014/06/14/percentage-of-bachelors-degrees-conferred-to-women-by-major-1970-2012/ *retrieved on 3/10/2016*

[4] Women Who Choose Computer Science – What Really Matters: The Critical Role of Encouragement and Exposure, http://static.googleusercontent.com/media/www.wenca.cn/en/us/edu/pdf/women-who-choose-what-really.pdf *retrieved on 3/21/16*

[5]http://www.wsj.com/articles/SB10001424127887323717004578159433220839020 *retrieved on 3/21/16*

[6]https://dub.washington.edu/djangosite/media/papers/unequalrepresentation.pdf *retrieved on 3/21/16*

[7] http://www.gettyimages.com/collections/leanin *retrieved on 3/21/16*

[8] http://www.millersville.edu/wmsc/ *retrieved on 4/10/16*

[9]http://www.matematicalia.net/index.php?option=com_content&task=view&id=2302&Itemid=453 *retrieved on 4/10/16*

[10]http://jar.sagepub.com/content/early/2015/05/15/0743558415587025 *retrieved on 3/21/16*

A Survey of Recent Developments in Queue Wait Time Forecasting Methods

Ron Davis
Department of Electrical & Computing Engineering
Tennessee State University
Nashville, Tennessee 37209
rdavis8@my.tnstate.edu

Tamara Rogers
Department of Computer Science
Tennessee State University
Nashville, Tennessee 37209
trogers3@tnstate.edu

Yingping Huang
Florence, Alabama, 35630
yingping@gmail.com

***Abstract*—Having customers wait at a full-service restaurant before they are seated is a common sight at most full-service restaurants located in the United States. Yet, it presents a difficult situation for both the restaurant and the customer. Restaurants may lose customers if the wait is too long because customers will view that time as unproductive and many will chose to avoid that experience which costs the restaurant revenue. Additionally, the wait time estimates given to potential customers by the hostess or manager are notorious for being inexact and are probably better described as guesstimates. If the wait time estimate given to potential customers is too long, the potential customer may depart. If the given number is too small, the potential customer may become irate if the actual wait time exceeds it. Given the importance of providing customers with an accurate wait time estimate, little has been done in this industry to develop or implement a better method of doing so. Participants in the industry place so little value on wait time estimates or on the actual wait time numbers that the sheets used to track these during a shift are routinely discarded at the end of the day.**

All of the recently proposed solutions to this problem require either that additional hardware be added to each restaurant location or that the customer provide their own hardware (smartphone) for use by the planned system. This paper surveys these methods and discusses the benefits and drawbacks of these proposals. As an alternative, this paper suggests using a time series equation to forecasting future wait time values that can be implemented purely through software and through the existing hardware available in the restaurants. The initial results of the time series model are presented. Finally, the paper proposes the development of a wait time estimation algorithm that would be used to generate wait time forecasts using readily available Internet data. The development of this algorithm would be based on dynamic regression which allows forecasts to be developed with external variables.

Keywords—algorithm development; forecasting; queue; time series, restaurant

I. Introduction

Research has shown that the keys to success for full-service restaurants in the United States are specializing the menu, offering healthy options, maintaining a clean and well managed restaurant, and having quick service [1]. There are times when these restaurants are busy, and they must place their customers and their potential customers in a queue particularly during peak times. Forcing customers to wait before they are even seated at a table violates the industry's goal of quick service. For this industry, it is a wide spread problem. Ninety-three percent of full-service restaurants have wait time periods at some point during the week and the average wait time is thirty minutes [2]. Given that there are expected to be approximately 257,000 casual dining restaurants in America by the year 2019 [3] and that sixty-eight percent of Americans visit casual dine restaurants at least once each week [4], waiting at restaurants is and will remain a wide spread problem that negatively impacts both customers and the restaurant industry.

Customers are even adding to the problem by extending the amount of time they are in the restaurant due to their usage of smartphones and mobile devices [5]. One restaurant examined video surveillance from the years 2004 and 2014 and discovered that their increases in wait time were due to customers taking on average thirteen minutes longer to order, spending twenty minutes longer eating, and taking an extra fifteen minutes to pay the check. These delays were frequently caused by customers' preoccupation with their smartphones throughout the dining process. Examples of this include checking social media, email, and texts; asking wait staff to connect their mobile devices to the restaurant's Wi-Fi hot spot or to take a group photo; and taking photos of food and posting them to the Internet.

Restaurants are presented with several challenges when trying to manage this problem. How do restaurants accurately calculate the correct wait time estimate for customers? This is a critical number for the restaurant to get correct. If the wait time number quoted to the customer underestimates the real wait time, then a customer may become irate or think that the restaurant has not been honest when them when the wait time exceeds the actual amount of time waited. On the other hand, if the restaurant overstates the estimated wait time, customers may decide that the wait is too long and chose to leave and dine with a competitor. The restaurant may cost itself business and profits by simply being inaccurate with its wait time estimations. Surprisingly, given the importance of accurate wait time estimates, these numbers are usually not based on any real data or mathematical calculations. They are simply

based on the experience of the hostess or the manager who gives their best guesstimate at the time.

Having the ability to generate accurate wait time estimates would give restaurant managers the ability to use this data to make better business decisions such as proper staffing levels and marketing choices. Providing accurate wait time information to customers makes them more informed to make a better decision about restaurant selection. Customers select restaurants by considering several parameters including price, quality, convenience, and speed [6]. Without an accurate wait time number for each restaurant the customer is considering, the ability of the customer to make rational fully informed decisions is impaired. Providing customers with adequate wait time estimates allows them to accurately perceive and weigh the convenience and speed factors during their decision making process.

II. Related Work

Queueing theory is the study of wait times and was first developed and published by Agner Erlang in 1909 [7]. Erlang, an engineer, mathematician, and statistician who worked for a telephone company, was attempting to determine how many telephone circuits were necessary to process a given number of telephone calls for a local area [8]. As further research into queueing theory occurred, Little's Law was developed and implemented into multiple fields. The equation for Little's Law consists of the long term average number of customers in a system (L), the average time a customer spends in the system (W), and the long term average arrival rate of new customers (λ) and is shown as Equation 2.1.

$$L = \lambda W \tag{2.1}$$

Given the mean customer arrival rate (λ), the mean service rate (μ), and the utilization factor ($\rho=\lambda/\mu$) in an M/M/1 queuing model, a restaurant can determine queuing factors such as:

- The probability that n customers are in the restaurant.

$$P_n = (1-\rho)\rho^n \tag{2.2}$$

- The average number of customers in the restaurant.

$$L = \frac{\lambda}{\mu - \lambda} \tag{2.3}$$

- The average number of customers in the queue.

$$L_q = \frac{\rho\lambda}{\mu - \lambda} \tag{2.4}$$

- The average amount of time customers spend in the restaurant. (This includes wait time and service time.)

$$W = \frac{1}{\mu - \lambda} \tag{2.5}$$

- The average amount of time a customer spends waiting in the queue.

$$W_q = \frac{\rho}{\mu - \lambda} \tag{2.6}$$

Gupta, Dharmadhikari, Bector, and Chow stated, "…if both the arrival and service time distributions are completely specified, we can, in principle, find all the performance distributions. However problems arise when only partial information is available about these distributions…[7]." In regards to Little's Law, Little and Graves stated that, "we are observing and measuring not forecasting" because while the relationship $L=\lambda W$ remains true, it is being conducted after the fact [9]. Therefore, other techniques must be used to overcome these issues.

The most obvious method to measure human queue parameters is to use trained human observers to do so. They can observe and measure wait times in queues, the number of people in a queue, and service times at the service point among other factors. However, it is not practical to do this for an extended period. Human observers are expensive to implement, would be costly to use on a wide-scale basis, and may not be generate consistent results among different individuals [10]. A potential solution is to replace visual observation and counting techniques used by a human with some piece of technology that can perform the same functions. Recent research exists detailing how this has been done with several different technological solutions.

2.1 Using Wi-Fi Signal Strength to Determine Queue Factors

One option used to determine wait times in human queues is to analyze and measure the signal strength of the smartphones of the people in the queue. The queue is categorized into three zones: waiting period, service period, and leaving period [11]. This monitoring station passively monitors packets sent from the smartphones of the individuals in the queue. It is expected that the signal strength of the packets will increase as the phone gets closer to both the service point and to the monitoring station. Once the person is at the service point, the signal strength is expected to stay steady as he is serviced and then experience a drastic drop off as service is finished and the individual leaves the service area [11]. This approach allows organizations to determine the beginning of the service point, the leaving point, and the end of leaving point in real time of a single user. These values are then used to calculate the amount of time that individual was in the waiting period (or the wait time the individual experienced in the queue), the service time for the individual, and the time when the individual left the queue. In calculating the beginning of service and leaving point times, this approach was determined to be accurate within four seconds [11].

Challenges using this technique include determining the exact point in time when a user transitions between the waiting period, the service period, and the leaving period. Determining exactly when these transitions occur is complicated by wireless signal propagation issues such as multipath interference and attenuation. Also, implementation in a full-service restaurant

may be problematic since the user may stay in the vicinity of the monitoring station during their entire dining experience.

2.2 Using Bluetooth to Determine Wait Times

Another option explored in literature is the possibility of using Bluetooth signals to determine passenger wait times in a security queue in an U.S. airport. The system was used to measure the transition times as passengers progressed through the pre-security area of the airport and through the security screening queue and checkpoint. The measurements concluded when the walk to a particular concourse was completed [12]. The idea for this came from a method used to take passenger vehicle traffic counts on roadways using the media access control (MAC) addresses of Bluetooth devices embedded in modern automobiles. The system developed in this case was composed of a pair of low powered Bluetooth receivers. As any passenger with a Bluetooth device approached the first receiver in the pre-screening area, his MAC address and a timestamp of the transaction were recorded. Once the passenger progressed through the queue for security and then completed the security checkpoint process, he entered the concourse where a second Bluetooth recording device again recorded the MAC address of his Bluetooth device and the transactional timestamp data [12]. An example of this would be a passenger carrying a Bluetooth device through this process with the unique MAC address of "00:21:06:8C:7A", the pre-security timestamp of 08:49, and concourse timestamp of 08:59 [12]. With these three pieces of data, it can be seen that the passenger entered security at 8:49 AM on a particular date and arrived in the concourse ten minutes later at 8:59 AM. The advantages of this approach are that it is inexpensive and simple to implement. Even though only five percent of passengers had their Bluetooth feature engaged, meaningful data was collected [12]. Disadvantages include concerns over personal privacy issues through the capture of the unique MAC address of passengers' Bluetooth devices. However, the MAC address may be discarded after the passenger has been detected by both Bluetooth receivers and the transit time has been calculated. Another disadvantage is that some passengers may not take a direct trip through the screening process. They may be diverted by take a side trip to a restaurant or restroom or an interaction with family members or friends.

2.3 Using Light and Switching Mats to Calculate Service Times

Another system implemented in an airport involved using light sensors and switching mats to measure passenger queue parameters. A light sensor is a piece of equipment that is low cost and has been in use for decades [10]. The transmitter transmits a continuous beam that was aimed across the queue and is redirected back by the reflector. Anyone passing though the beam breaks the connection and is counted as a single passenger. In addition, a timestamp of the event is created and stored. This method has limitations since two people walking side-by-side would still be counted as only one passenger and the transmitter and reflector need to be located where passengers cannot reach and tamper with them. A switching mat is made with two conductive plates. When the top plate does not have weight on it, it will not touch the bottom plate and no connection is made. However, when someone stands on the mat, both plates touch and a connection is made. When this occurs, the passenger count is increased by one and again a timestamp is taken [10]. This technology also has some limitations. During the data collection phase of the research, the switching mat failed for four days due to a defective power supply. With these devices, it is possible to count the number of passengers given that their movement is in one direction only with a reasonably high degree of accuracy. In one test, the light sensor counted 3,735 people compared to a manual count of 3,745 [10]. Results were not as good in a situation where people could walk bi-directionally in the queue, the sensors overstated the count at 205 versus the manual count of 182 [10]. Overall, the sensors were able to count the number of people in the queue to an acceptable level of accuracy. Given the timestamp data created by a crossing event, service and wait times in the queue may be calculated using these technologies. However, forcing all of the customers waiting at a restaurant to walk in only one direction in a single file line is not realistic.

2.4 Using Video Images to Estimate Queue Parameters

Another approach is to determine queue factors such as wait time, service time, and queue length is to use surveillance videos of the queue area. To accomplish this, numerous parameters must be addressed such as the type of queue, the type of service, lighting levels, camera angles, and the appearance of people (e.g. carrying a bag versus not carrying one) for this type of system to function properly [13]. To analyze an image, the system has a queue module and a counter module. The queue module estimates the number of individuals located in the queue area while the counter module detects an individual at a counter and estimates how long it takes each person to be serviced at the counters. These numbers are used to determine the service time of the counters.

Given the probability density function of the number of people waiting in the queue, $\rho_N(x)$, and the service time density, ρ_s, then the average waiting time, T_{avg}, for individuals in the queue is determined by Equation 2.7.

$$T_{avg} \approx \left(\int_0^\infty x p_N(x) dx \right) \left(\int_0^\infty s p_s(s) ds \right) \quad (2.7)$$

Proper camera placement is a key to success using this method. A top down installation versus a side view offers better performance and accuracy. Additionally, multiple cameras improve the performance but may not be practical due to economic concerns.

2.5 Using Smartphones to Determine Queue Wait Times

Another technique to determine customer wait time only (the service time and number of customers in the queue was not a goal of the system) involved installing an app on customers' smartphones to estimate line wait time through crowdsensing at a coffee shop located on a college campus. This coffee shop only handled pedestrians (no drive-thru for automobile traffic), had a First-In First-Out (FIFO) customer queue, and had a wireless access point (WAP) available for customer use with Internet connectivity.

An app was created and approximately 1000 students installed it on their smartphones. Having users manually enter wait time data was determined to be inefficient, so the app was modified to calculate queue wait times without user involvement through the localization features of the phone itself. The app uses the Basic Service Set Identification (BSSID) of the nearby WAPs as the basis for wait time estimation. When the app is activated, it scans for WAPs within range. Since the beacons of the WAP located in the café are unique, the arrival and exit of the customer from the shop may be detected. However, the system has no way of determining if the customer is waiting in the queue, being serviced, or has been serviced and is now sitting at a table drinking his coffee. Having a customer linger in the area generates a false positive for the system and returns an excessive value for his wait time. Also, having a potential customer visit the café but quickly exit without ordering generates another false positive value that is too low. The lack of ability to distinguish between customer states required the app to utilize different techniques that are implemented in the back end of the system when the wait time is calculated. For example, observational data showed that most wait times ranged between two and twenty minutes. Therefore, any wait times in excess of twenty minutes were viewed as a lingering customer and discarded as noise [14]. Those under two minutes were viewed as a quick entrance and exit by a person and were thrown out as well. The architecture of the app created by Bulut, Yilmaz, Demirbas, Ferhatosmanoglu, and Ferhatosmanoglu was divided into subsystems and organized as shown in Figure 1 [14].

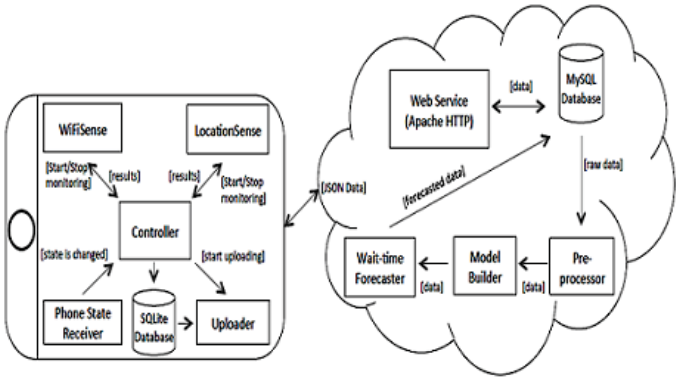


Fig. 1. System Architecture (Bulut M., et al.)

The characteristics of the system and its subsystems are as follows:

- A client side of the app shown on the left of Figure 1 resides and runs on customers' smartphones. The server side application shown on the right of Figure 1 is stored and runs on the cloud.
- The Phone State Receiver subsystem listens for specific changes in the state of the phone. If an event is triggered such as Wi-Fi status change, then a special object is fired. This can quickly drain the battery of the smartphone.
- The Wi-Fi Sense subsystem uses Wi-Fi beacons generated by the coffee shop's WAP which are cheap for the smartphone electrically. The beacons are used to calculate the smartphone's movements in the coffee shop without having to log into the WAP. This data is then sent to the cloud to calculate the current wait time for the customer. The app also operates under the assumption that if a user starts the app then they are going to travel to and enter the coffee shop.
- The Location Sense subsystem seeks to determine if the user is in close proximity to the coffee shop. If it is determined that the distance is less than 100 meters, the subsystem will set an alert to discover the timestamp at the point the customer enters the coffee shop and the moment when he exits the café.
- The Uploader subsystem collects the client side data from the application and transmits it to the cloud as input into the cloud side wait time estimation system.
- The Web Service subsystem is an interface between the client side app and the server side of the program. It accepts wait time data from smartphones and it provides wait time estimates to smartphones requesting this information.
- The Pre-Processor subsystem receives wait time data from the Web Services subsystem and removes any outliers and smooths the data.
- The Model Builder subsystem periodically makes the model based on the wait time data provided to it by the Pre-Processor.
- The Wait Time Forecaster subsystem uses the model created by the Model Builder subsystem to forecast future wait times.

There are several complications that arise as the wait time data is processed on the server side. First, it was determined that customer wait times in the coffee shop queue were based mostly on the time of the day, the day of the week, and the seasonality of the date [15]. For example, an on-campus coffee shop is going to have little or no wait times during spring break when most of the students and faculty are not present on campus. Time series analysis uses a data set that was attained by taking measurements sequentially over time and is a mathematical method used to extract information from the shape of data that reflect the trends and patterns contained in the data [15]. This technique was used to forecast the future wait times for the app. However, it was discovered that even with hundreds of users' smartphones automatically reporting the wait time data of the café on a regular basis, there were not enough readings available to use time series estimation without accounting for the sparseness of the data. Therefore, statistical techniques such as exponential smoothing, heuristic regression, and the Holt-Winters (HW) forecasting method were needed to supplement the time series method by smoothing out and filling in the missing data [14]. Also, observation data was collected by having someone sit in the coffee shop on a periodic basis with a stopwatch and manually record wait time data [15]. This was done in ten-minute intervals and was

referred to as ground truth data (GD). The GD data was not used for forecasting purposes; instead, it was used to measure error only.

The data were applied to two different estimation approaches. The first approach was to use Nearest Neighbor Estimation (NNE) method, which works well with sparse data sets and the algorithm requires *O(n)* computation time. NNE is a technique that categorizes new data into a known set or class. It is assigned based on what is most common among its nearest neighbors. For example, k = 1, would mean that the new data point would be assigned to the class of its nearest neighbor.

For this system, each data point was classified by three dimensions ([*w, d, i*]) where *w* is the week of the year (1-52), *d* is the day of the week (1-7), and *i* is the interval of the day (1-54). (There are 54 ten-minute intervals between 8:00 AM – 5:00 PM.) The next step was to determine the similarity between the vectors using a weighted Euclidean distance (L_{ij}) formula as shown in in Equation 2.8 [14]. Then, linear regression determined the relationship between the data ([*w, d, i*]) and the wait time (v_i) and optimized the weights α, β, and γ of Equation 2.9 [13].

$$L_{ij} = \sqrt{|\alpha(w_i - w_j)^2| + |\beta(d_i - d_j)^2| + |\gamma(i_i - i_j)^2|} \quad (2.8)$$

$$v_i = \alpha w_i + \beta d_i + \gamma i_i \quad (2.9)$$

Using eight weeks of collected data that has the outliers removed, it was determined that the weight for the week (α) was .991, the day (β) was .130, and the ten minute interval of the day (γ) was .032. This means that the day and interval of the day have a high similarity to the values collected in previous weeks. However, the weight for the week indicates that the importance of the weekly data decreases as time goes by. In other words, the data from the closest week is the most valuable in forecasting. The final step in this process is to find the data points in the historical data k distance and take the average of their wait times. This result is then presented at the new estimated wait time [15].

The second estimation approach attempted to fill in the sparseness of the data by using exponential smoothing and the Holt-Winters forecasting model. The Holt-Winters is popular because it is easy to automate and has low data storage requirements [16]. The Holt-Winters method does have some potential problems. It is susceptible to data outliers that can misrepresent forecasts and it can only accommodate a single seasonal pattern [16]. The equation for Holt-Winters is shown as Equation 2.10.

[Current Level + Trend] * Seasonal Index (2.10)

Exponential smoothing is a method that considers previous values of the time series data and assigns weights to those data. The weights decrease as the data get older which gives fresh or newer data greater weight. Given a forecasted value of s_t, a smoothing value of α, and a current value of x_t, the formula for exponential smoothing is shown as Equation 2.11.

$$s_t = \alpha x_t + (1 - \alpha) s_{t-1} \quad (2.11)$$

Overall, the system can estimate the wait time of each customer with a twenty seconds accuracy rate [15]. Modeling error was determined by calculating the Mean Absolute Error (MAE) with a set of n wait times: y_1, y_2, …, y_n and their estimated wait time values f_1, f_2, …, f_n MAE is given as Equation 2.12 [12].

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |f_i - y_i \quad (2.12)$$

Given this equation, modeling error was determined for the NNE, exponential smoothing, and Holt-Winters techniques over a two-week period. The forecasting errors for the three models are 227 seconds for NNE, 156 seconds for exponential smoothing, and 155 seconds for Holt-Winters. The forecasting error for the three models developed by Bulut, Yilmaz, Demirbas, Ferhatosmanoglu, and Ferhatosmanoglu covering the entire eight week period of collected data is shown as Figure 2. [14].

Finally, the accuracy of the forecasting ability of the app developed by Bulut, Yilmaz, Demirbas, Ferhatosmanoglu, and Ferhatosmanoglu is shown in Figure 3 [14]. It covers a two day period during the last week of data collection and uses the Holt-Winters method. The forecasted data is in blue while the collected data is in red.

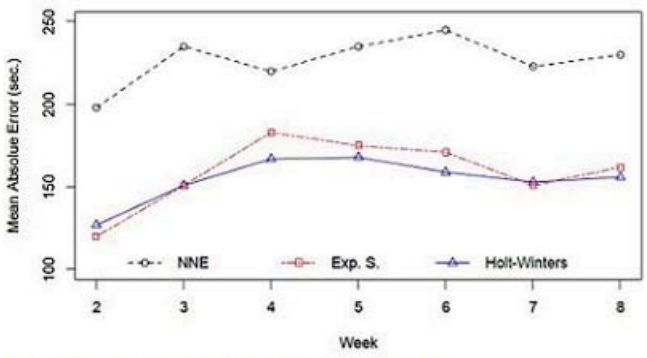


Fig. 2. Weekly Error of Each Model (Bulut, M. et al.)

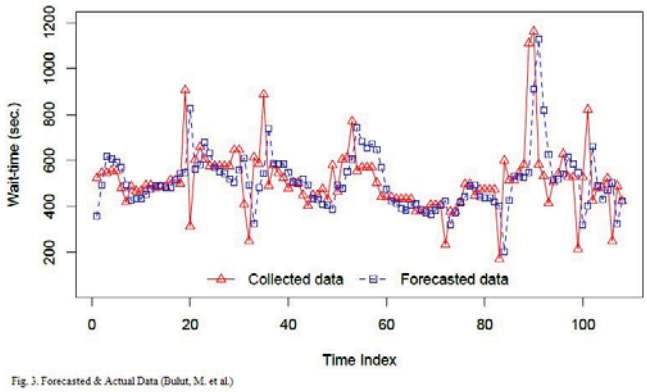


Fig. 3. Forecasted & Actual Data (Bulut, M. et al.)

The accuracy of the wait time estimation does not have to be exact since most customers have a tendency to misjudge

time intervals (overestimating short periods and underestimating longer ones) [17]. The majority of customers will not be upset if the estimated wait time is accurate within an acceptable range and will feel relief if the actual wait time is shorter than it was predicted to be [17]. Given all of this, the performance of the system as shown in Figure 3 is quite good.

The developers of the app state that they want to add additional features to their system such as determining the service time and being able to calculate and broadcast the length of the queue in addition to the forecasted wait times. Integrating with social networks would enable the app to be able to service other locations and other types of businesses that experience customer wait times such as banks or post office buildings. Expanding to other businesses would require the longitude and latitude of the location and the BSSID of the WAP located at the site. The business could be added to the app and wait time data could be collected as users visit the new business. Once enough data has been acquired, a new model could be developed to start forecasting wait times for the new location [15].

2.5 Research Objective

With varying degrees of success, each solution reviewed here presents a technological solution to overcome the problem of incomplete queuing parameter data. All of the techniques surveyed offer a solution that requires the placement of on-site infrastructure to determine different queueing parameters for that specific location. This infrastructure is provided either by the restaurant or by the customer. However, each method has its drawbacks and limitations. For example, using sensor mats and light barriers to count people and to estimate other queuing factors works well in a situation where passengers are required to walk in a limited amount of space all in the same direction. If passengers can walk in a bi-directional manner, then the results using these methods are overstated and misleading. Obviously, potential customers located in most restaurant waiting areas cannot be asked to physically line up and walk in a single direction so these technologies would not be appropriate for implementation in a full-service restaurant.

All of these methods start with the same assumption: it is necessary to have some device(s) in place at a specific location in order to determine queue parameters for that area. Overcoming this assumption is a research area with potential for advancement and progress. The ideal solution to forecasting future wait times for a full-service restaurant queue would be to do this in some mathematical fashion based on commonly and readily available data from the Internet. To reach this goal, a good starting point lies with historical wait time data and historical wait time estimates provided by the restaurants themselves. While this eliminates the need for additional technology on site, it has not been practical in the past on a large scale because either restaurants are not measuring or they are not maintaining their wait time numbers. This research seeks to determine if it is possible to simply this approach by avoiding adding any technology on site. Yet, the authors wish to generate similar or better results compared to what is currently available. This next section will show the initial steps of the process for developing an algorithm that uses freely available data from the Internet to infer or calculate the wait times of a queue of a full-service restaurant at specific U.S. locations. Furthermore, it is hoped that it will be possible to apply this algorithm to different full-service restaurants from various chains and brands to locations throughout the United States. If this approach is successful, applying it to numerous restaurants within an urban area, a region, or the entire country will be much simplified because this research is proposing a software solution rather than a hardware one.

III. Implementation

To develop this algorithm, it was decided that duplicating the work shown in section 2.5 by using a similar time series approach was a logical place to start. However, historical wait time data was needed from multiple restaurant locations to begin. The limitation of adding no new technology to collect this data from the restaurants was a significant limiting factor to obtaining this data. It was learned that very few people in the full-service restaurant industry view their wait time data and wait time estimates as a resource. Almost all of it is discarded at the end of the business day. For months, all of the data that was acquired for this research were data that were destined for the trash but simply had not been discarded yet.

Fortunately, a relationship was established with a full-service restaurant chain that covers most of the U.S. with over 500 locations. This chain agreed to provide its wait time estimates in fifteen minute increments for all of its U.S. stores to the authors electronically. At the last count, over 3 million data points of wait time data have been made available to support this research. One week of this data for this chain has been plotted and is shown as Figure 4.

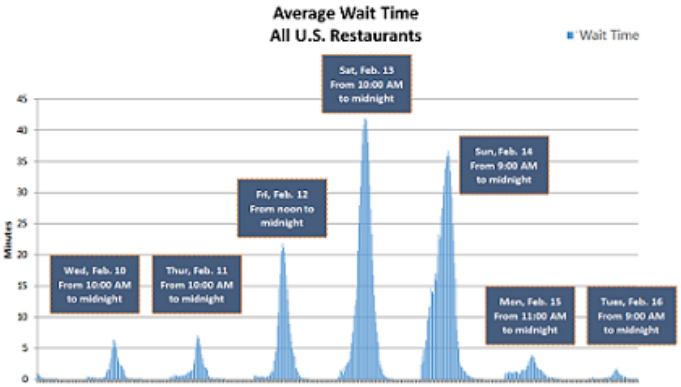


Fig. 4. One Week of Electronically Collected Wait Time Estimates

Using the R programming language, four weeks of this historical data was used to train forecasting models using the Holt-Winters, the Seasonal Trend Decomposition procedure using Loess (STL), exponential smoothing, and ARIMA methods to determine the trend, seasonality, and remainder components of the data. The mean absolute error calculation was performed for each model using equation 2.12 to determine which model gave the best fit for the data.

I V. Results & Future Work

The initial results of this work are encouraging. The best results came from the STL model which is shown in Figure 5.

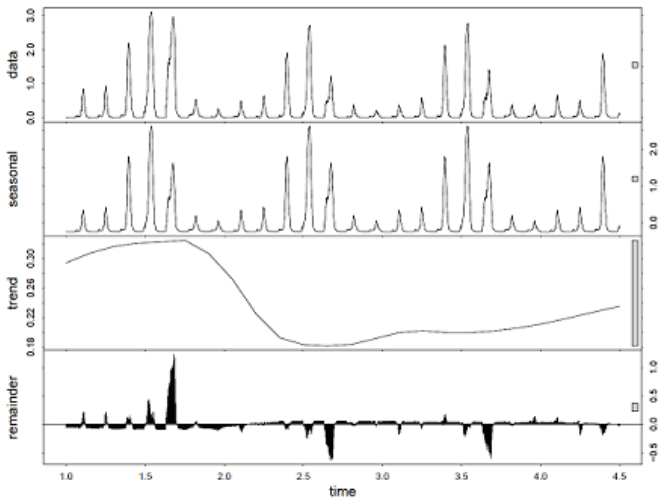


Fig. 5. Decomposition Plot of Wait Time Data

A weekly forecast was made using the STL approach and the forecast and the actual wait times are shown in Figure 6.

Fig. 6. Forecasted Versus Actual Wait Times

The next step in the process is to continue collecting wait time data and to refine the model currently being used. When a satisfactory level of performance of the model has been reached, it will be used to predict the wait times for the restaurant chain providing the data. In this case, satisfactory performance will be based on several different methods used to measure error such as the Mean Absolute Error as shown in Equation 2.12, the Mean Absolute Percentage Error (MAPE), and the Root Mean Square Deviation (RMSD).

All of this shows that if a restaurant or restaurant chain provides historical wait time information about its locations, then highly accurate wait time forecasts may be made by simply using a time series approach. After this has been completed, the next goal is to determine if this same collected data can be used to calculate the customer wait times for other restaurant chains that have similar characteristics such as size, location, and menu similarities. If this is possible, then a model will be developed and implemented. The final step will be to determine if there is a statistical correlation between restaurant wait times and commonly available data on the Internet such as traffic, weather, etc. The ultimate goal is to be able to develop an algorithm that will forecast restaurant wait times using only this Internet data as inputs into the model. An anticipated method of developing this algorithm will be through the use of a dynamic regression model which is a time series model that takes into account exogenous predictor variables which are totally independent from the other variables in a standard time series model. These external variables add the dynamic effects of causal factors to the model. For example, when modeling farming or crop output levels, the amount of rainfall would be an exogenous variable of the model.

REFERENCES

[1] Webstaurantstore.com, "Casual Dining vs. Fine Dining", 2016. [Online]. Available: http://www.webstaurantstore.com/article/2/casual-dining-vs-fine-dining.html. [Accessed: 01- Feb- 2016].

[2] FSR magazine, "Study Released on Average Restaurant Wait Times", 2013. [Online]. Available: https://www.fsrmagazine.com/new-restaurant-concepts/study-released-average-restaurant-wait-times. [Accessed: 11- Feb- 2016].

[3] Euromonitor.com, "Full-Service Restaurants in the US", 2016. [Online]. Available: http://www.euromonitor.com/full-service-restaurants-in-the-us/report. [Accessed: 10- Feb- 2016].

[4] Deloitte Development, "Second helpings: Building consumer loyalty in the fast service and casual dining restaurant sector", 2014. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/Tax/us-tax-thl-restaurant-loyalty-survey-032814.pdf. [Accessed: 15- Feb- 2016].

[5] J. Dodge, "Restaurant Wait Times Skyrocket, And Customers May Be To Blame", Chicago.cbslocal.com, 2016. [Online]. Available: http://chicago.cbslocal.com/2014/07/23/restaurant-wait-times-skyrocket-and-customers-may-be-to-blame/. [Accessed: 07- Feb- 2016].

[6] E. Duecy, "NPD: Diners look for menu variety, 'buzz' in full-service eateries", Nation's Restaurant News, 2006.

[7] Y. Gupta, A. Dharmadhikari, C. Bector and Wing Sing Chow, "Information theoretic approximations for single server queueing systems", Computers & Industrial Engineering, vol. 12, no. 1, pp. 23-38, 1987.

[8] R. Cooper, Introduction to Queueing Theory, 2nd ed. New York: North Holland, 1981.

[9] S. Graves and J. Little, Building Intuition: Insights from Basic Operations Management Models and Principles. New York: Springer Science+Business Media, LLC, 2008, pp. 81-100.

[10] D. Bauer, M. Ray and S. Seer, "Simple Sensors Used for Measuring Service Times and Counting Pedestrians", Transportation Research Record: Journal of the Transportation Research Board, vol. 2214, pp. 77-84, 2011.

[11] Y. Wang, Y. Chen and R. Martin, "Leveraging Wi-Fi Signals to Monitor Human Queues", IEEE Pervasive Computing, vol. 13, no. 2, pp. 14-17, 2014.

[12] D. Bullock, R. Haseman, J. Wasson and R. Spitler, "Automated Measurement of Wait Times at Airport Security", Transportation Research Record: Journal of the Transportation Research Board, vol. 2177, pp. 60-68, 2010.

[13] V. Parameswaran, V. Shet and V. Ramesh, "Design and Validation of a System for People Queue Statistics Estimation", Studies in Computational Intelligence, pp. 355-373, 2012.

[14] M. Bulut, Y. Yilmaz, M. Demirbas, N. Ferhatosmanoglu and H. Ferhatosmanoglu, "LineKing: Crowdsourced Line Wait-Time Estimation Using Smartphones", Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pp. 205-224, 2012.

[15] M. Bulut, M. Demirbas and H. Ferhatosmanoglu, "LineKing: Coffee Shop Wait-Time Monitoring Using Smartphones", IEEE Transactions on Mobile Computing, vol. 14, no. 10, pp. 2045-2058, 2015.

[16] P. Goodwin, "The Holt-Winters Approach to Exponential Smoothing: 50 Years Old and Going Strong", Foresight: The International Journal of Applied Forecasting, no. 19, 2010.

[17] A. Pruyn and A. Smidts, "Customers' reaction to waiting: Effects of the presence of 'fellow sufferers' in the waiting room", Advances in Consumer Research, vol. 26, pp. 211-216, 1999.

SESSION

NOVEL ALGORITHMS

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

Super Symmetric Boolean Functions

Peter M. Maurer
Dept. of Computer Science
Baylor University
Waco, Texas 76798-7356
Waco, Texas 76798

Abstract – Super symmetry is a type of matrix-based symmetry that extends the concept of total symmetry. Super symmetric functions are "even more symmetric" than totally symmetric functions. Even if a function is not super symmetric, the super symmetric transpose matrices can be used to detect partial super symmetries. These partial symmetries can be mixed arbitrarily with ordinary symmetric variable pairs to create large sets of mutually symmetric variables. In addition, one can detect subsets of super symmetric inputs, which are distinct from partial super symmetries. Super symmetry allows many new types of Boolean function symmetry to be detected and exploited.

1 Introduction

Symmetric Boolean functions have many applications in the field of Electrical Computer Aided Design (ECAD) [ref]. A symmetric Boolean function is a function of n variables, whose input variables can be rearranged in some fashion without changing the output of the function. An example is $x_1x_2x_3 + x_4$, (multiplication is AND, and addition is OR) in which the variables x_1, x_2 and x_3 can be rearranged arbitrarily.

This concept can be made more precise using permutations [1, 2]. Let f be an n-input Boolean function and $X = \{x_1, x_2, ..., x_n\}$ be its set of input variables. If p is a permutation on the set X that leaves f unchanged, then f is symmetric and is said to be *invariant* with respect to p. Also, f and p are said to be *compatible*. The set of all permutations of X is called the *symmetric group* of X, and is designated S_X. The *symmetry group,* G_f, of an n-input Boolean function, f, is the set of all permutations $p \in S_X$ that are compatible with f. Because the identity permutation, which leaves X unchanged, is compatible with every function, G_f is always non-empty. A function, f, is said to be *symmetric* if G_f contains more than one element.

The only thing that affects the structure of S_X is the size of X. If X and Y are two different sets such that $|X| = |Y|$, then S_X is isomorphic to S_Y. For simplicity, we will usually assume that $X = \{1, 2, 3, ..., n\}$, and will designate S_X as S_n. There is a natural mapping between $\{1, 2, 3, ..., n\}$ and sets of variables such as $\{x_1, x_2, ..., x_n\}$ or elements of vectors such as $(v_1, v_2, v_3, ..., v_n)$. When applying members of S_n to these sets, we will assume that the natural mapping between $\{1, 2, 3, ..., n\}$ and the set of indices is being used.

Symmetric Boolean functions were first studied by Shannon [3], who gave us Shannon's theorem, the basis of most symmetry detection algorithms. Shannon's theorem is based on the cofactors of a Boolean function, f, which are obtained by setting one or more input variables of f to constant values. For example, $x_2x_3 + x_4$ is the cofactor obtained by setting x_1 to 1 in the function $x_1x_2x_3 + x_4$.

Cofactors can be designated in several different ways. One can specify the variable and the value in a subscript, as in $f_{a=1}$. If there is a natural ordering to the variables, one can specify a list of variable values such as f_{xx10x}, where the x represents a variable that has not been replaced. Most often, when the variables in question are understood, we simply use lists of values as in f_0, f_1 or f_{101}.

Shannon's theorem states that two input variables, x_1 and x_2, of a function f are symmetric variable pairs if and only if $f_{01} = f_{10}$, where the cofactors are taken with respect to x_1 and x_2. The variables of a symmetric pair can be exchanged in arbitrary fashion without altering the output of the function. Symmetric variable pairs are transitive in the sense that if (x_1, x_2) is a symmetric variable pair, and (x_2, x_3), is a symmetric variable pair, then so is (x_1, x_3).

Since [3], there have been much work on detecting and exploiting symmetric functions.[4-24]. Symmetries can be broken into three broad categories, total symmetry which allows the inputs of a function to be permuted arbitrarily, partial symmetry, which allows one or more subsets of inputs to be permuted arbitrarily, and strong symmetry, which includes everything else. Some subclasses of strong symmetry, such as hierarchical symmetry [16], and rotational symmetry [17] have been identified and studied. The Universal Symmetry Detection Algorithm [25] is capable of detecting any type of strong symmetry.

2 Super Symmetry

As pointed out in [26], permutation-based symmetry can be recast in terms of matrices over GF(2). If one views an n-input function as a function of a single n-element vector, then traditional symmetry can be defined in terms of permutation matrices on these vectors. Permutation matrices are matrices that have a single 1 in each row and in each column. A permutation matrix is so called because it permutes the elements of a vector without changing them. One can obtain any permutation matrix p by permuting the rows of the identity matrix, I.

There is a one-to-one correspondence between permutations and permutation matrices. The set of all permutations on a set of n elements, S_n, and the set of all $n \times n$ permutation matrices, SR_n, are mathematical groups that are isomorphic to one another. Since the class of $n \times n$ non-singular matrices is much larger than the class of permutations on n input variables, matrices can be used to define a much larger class of symmetries than permutations.

For example, matrices can be used to define conjugate symmetry. Let SR_n be the set of all $n \times n$ permutation matrices, and let M be an arbitrary non-singular $n \times n$ matrix. Then the matrices in the set $G = \{M^{-1}NM \mid M \in SR_n\}$ define a new type of symmetry

ISBN: 1-60132-434-0, CSREA Press ©

called *conjugate symmetry*. Conjugate symmetry cannot be defined directly in terms of permutations and is a type of matrix-based symmetry.

Super symmetry is another type of matrix-based symmetry that extends the concept of total symmetry and the concept of permutation matrices. We start with SR_n, the $n \times n$ permutation matrices. Every matrix $M \in SR_n$ is both a row-permutation and a column-permutation of the identity matrix. For example, if $n = 4$, then every element of SR_4 can be constructed by arranging the rows (or columns) 0001, 0010, 0100, and 1000 in some order. We can expand SR_n by adding an $n+1^{st}$ row containing all ones to the existing set of *n* rows. Let HR_n be the set of all matrices that can be formed from these $n+1$ rows, without choosing duplicates. HR_n is closed under matrix multiplication, and is isomorphic to the symmetric group S_{n+1}. Figure 1 shows an example with $n = 3$. By the same token, we can start with the columns that contain a single 1, and add a column of all 1's. The set of all matrices that can be formed from these columns, without choosing duplicate columns, is VR_n. VR_n is also closed under matrix multiplication, and is isomorphic to S_{n+1}. If $n > 2$ then $HR_n \neq VR_n$. We call HR_n and VR_n the *super symmetric groups* of degree n.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\0&0&1\end{pmatrix}\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}\begin{pmatrix}1&0&0\\0&0&1\\0&1&0\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix}\begin{pmatrix}0&0&1\\1&0&0\\0&1&0\end{pmatrix}$$
$$\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&1&1\\1&0&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&1&1\\0&1&0\\1&0&0\end{pmatrix}\begin{pmatrix}1&1&1\\0&0&1\\0&1&0\end{pmatrix}\begin{pmatrix}1&1&1\\0&0&1\\1&0&0\end{pmatrix}\begin{pmatrix}1&1&1\\1&0&0\\0&1&0\end{pmatrix}$$
$$\begin{pmatrix}1&0&0\\1&1&1\\0&0&1\end{pmatrix}\begin{pmatrix}0&1&0\\1&1&1\\0&0&1\end{pmatrix}\begin{pmatrix}0&0&1\\1&1&1\\1&0&0\end{pmatrix}\begin{pmatrix}1&0&0\\1&1&1\\0&1&0\end{pmatrix}\begin{pmatrix}0&1&0\\1&1&1\\1&0&0\end{pmatrix}\begin{pmatrix}0&0&1\\1&1&1\\0&1&0\end{pmatrix}$$
$$\begin{pmatrix}1&0&0\\0&1&0\\1&1&1\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\1&1&1\end{pmatrix}\begin{pmatrix}0&0&1\\0&1&0\\1&1&1\end{pmatrix}\begin{pmatrix}1&0&0\\0&0&1\\1&1&1\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&1\\1&1&1\end{pmatrix}\begin{pmatrix}0&0&1\\1&0&0\\1&1&1\end{pmatrix}$$

Figure 1. The Super Symmetric Group HR_3.

To prove that HR_n and VR_n are groups isomorphic to S_{n+1} we start with the following theorem which proves that the matrices of HR_n and VR_n are non-singular.

Theorem 1. *Every element of HR_n and VR_n is non-singular.*

Proof. Let $M \in HR_n$. If M is singular then some subset of the rows of M must sum to zero. If there is no row of all ones, then M is a permutation matrix and non-singular. Let us assume that row i of M is all ones. Other than row i, there are $n-1$ rows of M, each containing a single 1. These rows are part of some permutation matrix, and therefore, no subset of rows that does not include row i can sum to zero. Other than the 1's in row i, the matrix M contains exactly $n-1$ 1's. Therefore there must be at least one column that contains no 1's, except the 1 in row i. No sum of rows that includes row i can have a zero in column i, because every other row has a zero in this column. Therefore no subset of the rows of M sums to zero, and M is non-singular. Now consider $N \in VR_n$. By a similar argument, we can show that no subset of the columns of N can sum to zero, therefore N is nonsingular.■

Now we can prove that HR_n and VR_n are groups isomorphic to S_{n+1}.

Theorem 2: *HR_n and VR_n are closed under matrix multiplication, and are isomorphic to S_{n+1}.*

Proof: Let $M, N \in HR_n$ and consider the form of $K = M \times N$. Because M and N are nonsingular, K must be nonsingular. If no row of M is all ones, then M is a permutation matrix. In this case, K is a row-permutation of N, and $K \in HR_n$. So let us assume that row i of M is all 1's. Now, suppose N is a permutation matrix. Because every row of N has a single 1, every row, except row i, of K has a single 1. Row i of K is the sum of all rows of N, which is a row of all 1's. Therefore $K \in HR_n$. If N is not a permutation matrix, then it must have a row, j of all 1's. In this case, the rows of K, except for row i must be a permutation of the rows of N, not including row i. Row i of K must be the sum of the rows of N. Every column of N, except one, must have exactly 2 ones. The remaining column must have a single one. Therefore the sum of the rows of N must contain a single one in some position, and zeroes elsewhere. Because the product is non-singular, row i cannot duplicate any other row of K. Therefore, K must either be a permutation matrix, or a permutation matrix with one row replaced by a row of all ones. Thus $K \in HR_n$, and HR_n is closed under multiplication. A similar argument shows that VR_n is also closed under multiplication. To show that HR_n is isomorphic to S_{n+1}, it suffices to show that HR_n is the set of permutations of a set of size $n+1$. This follows from the fact that every matrix in HR_n is a permutation of the $n+1$ rows used to form the elements of HR_n, each element, M, of HR_n has n rows from the set of $n+1$ rows. The missing row is always unique, and we can imagine it as being appended as the $n+1^{st}$ row of M. Thus HR_n is isomorphic to S_{n+1}. A similar argument on the columns of the elements of VR_n shows that VR_n is also isomorphic to S_{n+1}.■

Any finite set of non-singular matrices that is closed under multiplication is a group. Because HR_n and VR_n are groups, they can serve as the symmetry group of certain functions. We say that a function f is *super symmetric* if either HR_n or VR_n leaves f invariant. If we wish to be more specific, we will call f *H-super symmetric* or *V-super symmetric*.

3 Boolean Orbits

Let G be a group of $n \times n$ matrices. Two *n*-element vectors v and w are said to be in the same Boolean orbit of G if there is a matrix $M \in G$ such that $v \times M = w$. Being in the same Boolean orbit is an equivalence relation that breaks the set of all *n*-element vectors into a collection of disjoint subsets. The Boolean orbits of a group can be used to determine whether a group G is the compatible with

a function f. The function f is compatible with G if and only if f maps every element of each Boolean orbit of G to the same value. For example, the symmetric group S_3 has the Boolean orbits $\{(0,0,0)\}$, $\{(0,0,1),(0,1,0),(1,0,0)\}$, $\{(0,1,1),(1,0,1),(1,1,0)\}$ and $\{(1,1,1)\}$. A 3-input function f is totally symmetric if and only if f maps the three vectors $\{(0,0,1),(0,1,0),(1,0,0)\}$ to the same value, and the three vectors $\{(0,1,1),(1,0,1),(1,1,0)\}$ to the same value.

The Universal Symmetry Detection algorithm can detect any type of symmetry as long as the Boolean orbits of that symmetry are known. The Boolean orbits of V and H super symmetry are relatively easy to compute. Since every super symmetric function is also totally symmetric, all vectors of the same weight must be contained in a single orbit. The Boolean orbits of V and H symmetry can be obtained by combining the Boolean orbits of total symmetry.

Let us first derive the Boolean orbits of H super symmetry. We will designate the set of all vectors of weight k as W_k. The sets W_k are just the Boolean orbits of total symmetry. We will designate the Boolean orbits of H super symmetry as O_k, where k is the weight of the lightest vector in O_k. Note that if an orbit O_i contains any vector of weight k, then $W_k \subseteq O_i$. In particular, $W_k \subseteq O_k$. Consider the orbit O_0. This orbit must contain a single vector, since every linear transformation maps the zero vector onto itself. The orbit O_1, contains all vectors of weight 1 and must also contain the vector of all 1's. Let v be an n-element vector of weight 1, and let $M \in HR_n$. The vector $v \times M$ must be equal to some row of M, and must either be a vector of weight 1 or a vector of all 1's. If v is a vector of all 1's, and M is a permutation matrix, then $v \times M$ is a vector of all 1's. If M contains a row of all 1's then $v \times M$ is the sum of the rows of M. Every column except one of M contains exactly two ones. The other column contains exactly one 1. Thus the sum of the rows of M is a vector of weight 1, and $O_1 = W_1 \bigcup W_n$. Now consider the orbit O_2 containing all vectors of weight 2. Let M be any element of HR_n. Any vector that can be formed by adding two rows i and j of M must be an element of O_2. If rows i and j of M are both of weight 1, then their sum is of weight 2 and is already contained in O_2. Let us assume that one of the rows is all ones. Then the sum of rows i and j is of weight $n-1$ and $W_{n-1} \subseteq O_2$. Now suppose that v is of weight $n-1$. If M is a permutation matrix or if M contains a row of all 1's and this row corresponds to the zero element of v, then $v \times M$ is of weight $n-1$. If M contains a row of all 1's and this row *does not* correspond to the zero element of v, then $v \times M$ is the sum of a vector of all 1's and $n-2$ distinct vectors of weight 1. Thus $v \times M$ is a vector of weight 2, and $O_2 = W_2 \bigcup W_{n-1}$. Continuing in this vein, we can show that any H super symmetry Boolean orbit, O_k, is equal to $W_k \bigcup W_{n-k+1}$, where k runs from 1 through $\left\lceil \frac{n}{2} \right\rceil$.

Now let us derive the Boolean orbits of V super symmetry. We will designate each orbit as Q_i, where i is the smallest weight of any element of Q_i. Note that if $W_j \bigcap Q_i \neq \phi$ then $W_j \subseteq Q_i$. In particular, $W_i \subseteq Q_i$. As before, Q_0 contains only the zero vector.

When a vector $v = (a_1,...,a_n)$ is multiplied by a matrix $V_i \in VR_n$, the result is $v' = (a_1,...a_{i-1}, p, a_{i+1}...,a_n)$, where p is the parity of v. (i.e., p is 1 if the number of bits in v is odd.) If $a_i = 0$ and $p = 0$, or if $a_i = 1$ and $p = 1$, then $v = v'$. If $a_i = 0$ and $p = 1$ then the weight of v' is one larger than that of v. If $a_i = 1$ and $p = 0$ then the weight of v' is one smaller than that of v. Note that the weight of v can increase only if it is odd, and can decrease only if it is even. Thus $Q_i = W_i \bigcup W_{i+1}$, where i is odd, i running from 1 to m where m is the largest odd number less than or equal to n. The other matrices of VR_n will not affect these orbits because they are either permutation matrices that do not change the weight of a vector, or they are permutation matrices with a single column set to ones. Such matrices combine a permutation of v with parity insertion, and do not change the orbits described above.

We have created a super symmetry detection module to the universal symmetry detector using the Boolean orbits describe above.

4 Symmetric Variable Pairs

Although the universal symmetry detection algorithm can detect super symmetry, super symmetric functions are comparatively rare. The same is true, of course, for totally symmetric functions. However, when a function is not totally symmetric, it may be partially symmetric, and using symmetric variable pairs, we can detect such partial symmetries. By the same token, we can detect super symmetric variable pairs and partial super symmetries. The super symmetric variable pairs can be mixed arbitrarily with ordinary symmetric variable pairs.

Ordinary symmetric variable pairs correspond to a type of a permutation called a *transposition*. A transposition of a set, X, is a permutation that swaps two elements of X, leaving everything else fixed. In the matrix domain, a transposition corresponds to a *transpose matrix*. A transpose matrix swaps two elements of an input vector, leaving all other elements fixed. We designate a transpose matrix that swaps elements i and j of a vector as $T_{i,j}$. Every row, k, of $T_{i,j}$ except rows i and j, is identical to row k of the identity matrix. Row i of $T_{i,j}$ has a 1 in column j and zeros elsewhere. Row j has a 1 in column i and zeros elsewhere. Figure 2 has several examples of transpose matrices.

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Figure 2. Some transpose matrices.

Super symmetry introduces $2n$ new transpose matrices known as the *super symmetric transpose matrices*. Half of these matrices are taken from HR_n and half are taken from VR_n.

In permutation matrices, we consider a row containing a 1 in position i and zeroes elsewhere to represent the i^{th} input variable. Alternatively, we could consider a column containing a 1 in the i^{th} position to represent the i^{th} input variable. In the super symmetric matrices, we consider the row of all 1's or a column of all 1's to represent an $n+1^{st}$ "invisible" variable. In HR_n a super symmetric transpose matrix is a matrix that is identical to the identity matrix except for row i, which is a row of all 1's. In VR_n a super

symmetric transpose matrix is identical to the identity matrix except for column i which is a column of all 1's. We designate these matrices as H_i and V_i respectively. Figure 3 gives some examples of such matrices.

$$\begin{pmatrix}1&0&0&0\\0&1&0&0\\1&1&1&1\\0&0&0&1\end{pmatrix}\quad\begin{pmatrix}1&1&1\\0&1&0\\0&0&1\end{pmatrix}\quad\begin{pmatrix}1&0&0&1\\0&1&0&1\\0&0&1&1\\0&0&0&1\end{pmatrix}\quad\begin{pmatrix}1&1&0\\0&1&0\\0&1&1\end{pmatrix}\quad\begin{pmatrix}1&1\\0&1\end{pmatrix}$$

Figure 3. Super Symmetric Transpose Matrices.

For any ordinary transpose matrix $T_{i,j}$, the matrix is self-inverting. That is, $T_{i,j}T_{i,j} = I$. As the Theorem 3 shows, the same is true for the matrices V_i and H_i.

Theorem 3. $H_iH_i = I$ *and* $V_iV_i = I$ *for all* $1 \le i \le n$.

Proof: Since H_i is identical to the identity matrix, except for row i, every row of H_iH_i is identical to the identity matrix, except for row i. Because row i of H_i is all ones, row i of H_iH_i is the sum of the rows of H_i. Every column of H_i contains exactly two 1's, except for column i which contains exactly one 1. Thus the sum of the rows of H_i has a 1 in column i and zeroes elsewhere, and is equal to row i of the identity matrix.

Similarly, since every column k of V_i, except for column i, is identical to column k of the identity matrix, every column k of V_iV_i is identical to column k of the identity matrix. Because column i of V_i is all ones, column i of V_iV_i is the sum of the columns of V_i. Every row of V_i, except row i has exactly two 1's. Row i has exactly one 1. Therefore the sum of the columns of V_i has a 1 in row i and zeroes elsewhere, and column i of V_iV_i is identical to column i of the identity matrix.■

It is convenient to think of the matrices H_i and V_i as being transpose matrices between x_i and the "invisible" $n+1^{st}$ variable, x_{n+1}. This makes the transitivity of the new matrices more obvious. For example, because of transitivity, a function is H super symmetric if it is compatible with $T_{1,2}$, $T_{1,3}$, …, $T_{1,n}$ and H_1. For V super symmetry, we substitute V_1 for H_1.

Another important and useful property of the super symmetric transpose matrices is that the conjugate of any matrix H_i with another matrix H_j $(i \ne j)$ is an ordinary transpose matrix. The same is true for matrices V_i and V_j, as the following theorem shows.

Theorem 4. *Suppose* $i \ne j$. *Then* $H_j^{-1}H_iH_j = T_{i,j}$ *and* $V_j^{-1}V_iV_j = T_{i,j}$.

Proof: By Theorem 3, $H_j^{-1} = H_j$ and $V_j^{-1} = V_j$, so $H_j^{-1}H_iH_j = H_jH_iH_j$ and $V_j^{-1}V_iV_j = V_jV_iV_j$. H_jH_i has the following form. Since every row of H_j, except row, is identical to the corresponding row of the identity matrix, every row, except row j of H_jH_i is identical to the corresponding row of H_i. Because row j of H_j is all ones, row j of H_jH_i is the sum of the rows of H_i. Every column of H_i has exactly two 1's, except for column i, which has exactly one 1. Thus row j of H_jH_i has a one in column i and zeroes elsewhere. Row i of H_jH_i contains all 1's. We can use the structure of H_jH_i to deduce the structure of $H_jH_iH_j$. Because every row of H_jH_i except rows i and j is identical to the corresponding row of the identity matrix, every row of $H_jH_iH_j$, except rows i and j is identical to the corresponding row of the identity matrix. Because row j has a 1 in column i and zeroes elsewhere, row j of $H_jH_iH_j$ is identical to row i of H_j, and has a 1 in column i and zeroes elsewhere. Because row i of H_jH_i contains all 1's, row i of $H_jH_iH_j$ is the sum of the rows of H_j. Every column of H_j has exactly two ones, except for column j which has exactly one 1. Therefore row i of $H_jH_iH_j$ has a 1 in column j and zeroes elsewhere. Therefore $H_jH_iH_j$ is the transpose matrix $T_{i,j}$.

Now consider the structure of V_iV_j. Because every column of V_j is identical to the corresponding column of the identity matrix, except for column j, every column of V_iV_j is identical to the corresponding column of V_i, except for column j. Because column j of V_i is all ones, column j of V_iV_j is equal to the sum of the columns of V_i. Every column of V_i contains exactly two 1's, except for column i which has exactly one 1. Thus column j of V_iV_j has a 1 in row i and zeroes elsewhere. Column i of V_iV_j contains all ones. We can now deduce the structure of $V_jV_iV_j$. Since every column of V_iV_j except for columns i and j, is identical to the corresponding column of the identity matrix, every column of $V_jV_iV_j$, except for columns i and j, is identical to the corresponding column of V_j. But these columns are identical to the corresponding columns of the identity matrix, so every column of $V_jV_iV_j$, except for columns i and j, is identical to the corresponding column of the identity matrix. Column j of $V_jV_iV_j$ is equal to column i of V_j, which has a 1 in row i and zeroes elsewhere. Because column i of V_iV_j is all ones, column i of $V_jV_iV_j$ is the sum of the columns of V_j. Every row of V_j has exactly two 1's, except for row j, which has exactly one 1. Thus the sum of the columns of V_j has a 1 in row j and zeroes elsewhere. Thus $V_jV_iV_j$ is the transpose matrix $T_{i,j}$.■

Let f be an n-input function with input variables $\{x_1, x_2, ..., x_n\}$. To determine whether f is compatible with H_i, we select some variable other than x_i, say x_j with $i \ne j$, and conditionally invert every variable except x_j itself with respect to x_j. These conditional inversions can be done simultaneously using the matrix H_j. We compute $f'(v) = f(H_j(v))$. The function f is compatible with H_i if and only if (x_i, x_j) is a symmetric variable

pair of f'. The correctness of this procedure stems from the fact that if $i \neq j$ then $H_j^{-1} H_i H_j = T_{i,j}$. Super symmetry can be viewed as a type of conjugate symmetry requiring multiple simultaneous conditional inversions.

Given the same function, f, we can determine whether f is compatible with V_i by selecting any input variable other than x_i, say x_j with $i \neq j$, and conditionally invert x_j with respect to every variable other than x_j itself. This gives us the new function, $f''(v) = f(V_i(v))$. The function f is compatible with V_i if and only if (x_i, x_j) is a symmetric variable pair of f''. Again, the correctness of this procedure depends on the fact that $V_j^{-1} V_i V_j = T_{i,j}$.

Because super symmetric transpose matrices can be equated with a type of conjugate symmetry, they can be detected and utilized by the hyperlinear algorithm for digital simulation [26, 27], and by other algorithms that detect symmetry using symmetric variable pairs.

5 Sub-Symmetries

For a Boolean function f to possess X symmetry in variables $\{x_1, x_2, ..., x_k\}$ every cofactor of the form $f_{xx...xa_{k+1}...a_n}$ must possess X symmetry. We usually do this by ensuring the symmetry relations exist between cofactors of the form $f_{a_1...a_k xx...x}$. It is possible for an *n*-input function to be super symmetric in any proper subset of its input variables, and it is possible for a function to have several subsets of variables in which it is super symmetric.

This is not the same as partial symmetry, because the super symmetric variable pairs involve all inputs of a function, while sub-super symmetries involve only a subset of variables. It is possible to test a subset of variables for super symmetry, and to test the same subset for compatibility with the super symmetric transpose matrices of the sub-symmetry. This gives us many more opportunities to detect symmetries in a Boolean function, because there are $2^n - 2$ proper subsets of variables, and $\sum_{i=2}^{n-1} 2i = 2\left(\frac{n(n-1)}{2} - 1\right) = n^2 - n - 2$ additional super symmetric transpose matrices.

6 Experimental Data

To determine the prevalence of super symmetry in real circuits, we tested the ISCAS 85 benchmarks for the presence of super symmetries. We tested for total super symmetry, for super symmetric variable pairs, and for sub symmetries. The results of our tests are given in Figure 4. These results show that super symmetries do indeed exist in real circuits, and are, in fact, quite numerous. The results for super symmetric variable pairs and for sub symmetries are especially encouraging. Because, in several cases, the number of symmetries exceeds the number of functions, it is clear that there are many functions that exhibit multiple sub-super symmetries and that there are functions that are compatible with many super symmetric variable pairs.

Circuit	Super Sym.	Var. Pairs	Sub-Sym.
c432	78	213	1097
c499	0	56	728
c880	122	33	902
c1355	288	44	704
c1908	158	59	5326
c2670	276	90	3145
c3540	710	1310	2093
c5315	830	2313	6206
c6288	512	528	928
c7552	582	1660	10093

Figure 4. Experimental Results.

7 Conclusion

The various aspects of super symmetry allow many different types of Boolean function symmetry to be detected and exploited. In addition to super symmetry itself we have partial super symmetries which are generated by the super symmetric transposition matrices. These partial symmetries can be mixed and matched in an arbitrary fashion with ordinary symmetric variable pairs. In addition, there are sub-super symmetries and partial sub-super symmetries which greatly expand the opportunity for detecting and exploiting symmetries in a Boolean function.

What is even more exciting, super symmetry allows us to exploit more of the full power of matrix-based symmetry. For example, for 4-input functions, there are 24 permutations of the inputs, but 20160 non-singular 4×4 matrices. There are obviously many more kinds of matrix-based symmetry than permutation-based symmetry, and super symmetry is only one of these.

We expect this work to be the basis of much more extended work in the future.

8 References

[1] D. S. Passman, *Permutation Groups.* New York: W. A. Benjamin, 1968.

[2] D. Robinson, *A Course in the Theory of Groups.* New York: Springer, 1995.

[3] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical Journal,* vol. 28, pp. 59-98, 1949.

[4] A. Abdollahi and M. Pedram, "Symmetry detection and Boolean matching utilizing a signature-based canonical form of Boolean functions," *IEEE Trans. on Computer-Aided Design,* vol. 27, pp. 1128-1137, June, 2008.

[5] N. N. Biswas, "On Identification of Totally Symmetric Boolean Functions," *Computers, IEEE Transactions On,* vol. 19, pp. 645-648, 1970.

[6] R. C. Born and A. K. Scidmore, "Transformation of switching functions to completely symmetric switching functions," *IEEE Transactions on Computers,* vol. 17, pp. 596-599, 1968.

[7] J. T. Butler, G. W. Dueck, V. P. Shmerko and S. Yanuskevich, "Comments on "Sympathy: fast exact minimization of fixed polarity Reed-Muller expansion for symmetric functions"," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 19, pp. 1386-1388, 2000.

[8] M. Chrzanowska-Jeske, "Generalized symmetric variables," in *The 8th IEEE International Conference on Electronics, Circuits and Systems,* 2001, pp. 1147-1150.

[9] K. S. Chung and C. L. Liu, "Local transformation techniques for multi-level logic circuits utilizing circuit symmetries for power reduction," in *Proceedings of the 1998 International Symposium on Low Power Electronics and Design,* 1998, pp. 215-220.

[10] P. T. Darga, K. A. Sakallah and I. L. Markov, "Faster symmetry discovery using sparsity of symmetries," in *Proceedings of the 45th Annual Design Automation Conference,* 2008, pp. 149-154.

[11] R. Drechsler and B. Becker, "Sympathy: Fast exact minimization of fixed polarity reed-muller expressions for symmetric functions," in *European Design and Test Conference,* 1995, pp. 91-97.

[12] Y. Hu, V. Shih, R. Majumdar and L. He, "Exploiting Symmetries to Speed Up SAT-Based Boolean Matching for

Logic Synthesis of FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, pp. 1751-1760, 2008.

[13] B. Hu and M. Marek-sadowska, "In-place delay constrained power optimization using functional symmetries," in *Design Automation and Test in Europe,* 2001, pp. 377-382.

[14] W. Ke and P. R. Menon, "Delay-testable implementations of symmetric functions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 14, pp. 772-775, 1995.

[15] N. Kettle and A. King, "An anytime algorithm for generalized symmetry detection in ROBDDs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 27, pp. 764-777, 2008.

[16] V. N. Kravets and K. A. Sakallah, "Generalized symmetries in boolean functions," Advanced Computer Architecture Laboratory Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, 2002.

[17] J. Mohnke, P. Molitor and S. Malik, "Limits of using signatures for permutation independent Boolean comparison," *Formal Methods Syst. Des.,* vol. 21, pp. 167-191, 2002.

[18] D. Moller, J. Mohnke and M. Weber, "Detection of symmetry of boolean functions represented by ROBDDs," in *IEEE International Conference on Computer-Aided Design,* 1993, pp. 680-684.

[19] J. C. Muzio, D. M. Miller and S. L. Hurst, "Multivariable symmetries and their detection," *IEE Proceedings on Computers and Digital Techniques,* vol. 130, pp. 141-148, 2008.

[20] S. Panda, F. Somenzi and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," in *IEEE International Conference on Computer-Aided Design,* 1994, pp. 628-631.

[21] J. Rice and J. Muzio, "Antisymmetries in the realization of boolean functions," in *IEEE International Symposium on Circuits and Systems,* 2002, pp. 69-72.

[22] C. Scholl, D. Moller, P. Molitor and R. Drechsler, "BDD minimization using symmetries," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 18, pp. 81-100, 1999.

[23] C. C. Tsai and M. Marek-Sadowska, "Generalized Reed-Muller forms as a tool to detect symmetries," *IEEE Transactions on Computers,* vol. 45, pp. 33-40, 1996.

[24] K. H. Wang and J. H. Chen, "Symmetry detection for incompletely specified functions," in *Proceedings of the 41st Annual Design Automation Conference,* 2004, pp. 434-437.

[25] P. Maurer, "A universal symmetry detection algorithm," Baylor University, 2013.

[26] P. M. Maurer, "Conjugate Symmetry," *Formal Methods Syst. Des.,* vol. 38, pp. 263-288, 2011.

[27] P. M. Maurer. Efficient event-driven simulation by exploiting the output observability of gate clusters. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions On 22(11),* pp. 1471-1486. 2003.

On r-gatherings on the Line

Yijie Han[1] **and Shin-ichi Nakano**[2]
[1]School of Computing and Engineering, University of Missouri at Kansas City, Kansas City, MO 64110, USA
[2]Department of Computer Science, Gunma University, City, Kiryu 376-8515, Japan

Abstract— *Given an integer r, a set C of customers, a set F of facilities, and a connecting cost $co(c, f)$ for each pair of $c \in C$ and $f \in F$, an r-gathering of customers C to facilities F is an assignment A of C to open facilities $F' \subset F$ such that r or more customers are assigned to each open facility. We wish to find an r-gathering with the minimum cost, where the cost is $\max_{c_i \in C}\{co(c_i, A(c_i))\}$. When all C and F are on a line an algorithm to find such an r-gathering is known. In this paper we give a faster algorithm with time complexity $O(|C| + |F| \log^2 r + |F| \log |F|)$.*

Keywords: algorithm, facility location, gathering

1. Introduction

The facility location problem and many of its variants are studied[7], [8]. In the basic facility location problem we are given (1) a set C of customers, (2) a set F of facilities, (3) an opening cost $op(f)$ for each $f \in F$, and (4) a connecting cost $co(c, f)$ for each pair of $c \in C$ and $f \in F$, then we open a subset $F' \subset F$ of facilities and find an assignment A of C to F' so that a designated cost is minimized.

An *r-gathering*[6] of customers C to facilities F is an assignment A of C to *open facilities* $F' \subset F$ such that r or more customers are assigned to each open facility. (This means each open facility has enough number of customers.) We assume $|C| >> r$ holds. Then we define the cost of (the max version of) a gathering as $\max_{c_i \in C}\{co(c_i, A(c_i))\}$. We assume $op(f_j) = 0$ for each $f_j \in F$ in this paper, as in [4]. The min-max version of the *r-gathering problem* finds an r-gathering having the minimum cost. For the min-sum version see the brief survey in [6].

Assume that F is a set of locations for emergency shelters, and $co(c, f)$ is the time needed for a person $c \in C$ to reach a shelter $f \in F$. Then an r-gathering corresponds to an evacuation assignment such that each opened shelter serves r or more people, and the r-gathering problem finds an evacuation plan minimizing the evacuation time span.

Armon[6] gave a 3-approximation algorithm for the r-gathering problem and proves that with assumption $P \neq NP$ the problem cannot be approximated within a factor less than 3 for any $r \geq 3$. Akagi and Nakano[4] gave an $O((|C| + |F|) \log(|C| + |F|))$ time algorithm to solve the r-gathering problem when all C and F are on a line. In this paper we give a faster $O(|C| + |F| \log^2 r + |F| \log |F|)$ time algorithm. Since we can assume in general $|F| << |C|$ and $r << |C|$ our algorithm is faster than the one in[4].

The remainder of this paper is organized as follows. Section 2 gives an algorithm to solve a decision version of the r-gathering problem, which is used as a subroutine in our main algorithm in Section 4. In Section 3 we describe the computation of left and right boundaries. Section 4 contains our main algorithm for the r-gathering problem. Section 5 analyze the running time of the algorithm tightly. Finally Section 6 is a conclusion.

2. (k,r)-gathering on the line

In this section we give an algorithm to solve a decision version of the r-gathering problem.

Given customers $C = \{c_0, c_1, \cdots, c_{|C|-1}\}$ and facilities $F = \{f_0, f_1, \cdots, f_{|F|-1}\}$ on a line (we assume they are distinct points and appear in those order from left to right) and two numbers k and r, (k, r)-gathering is an r-gathering such that $\max_{c_i \in C}\{co(c_i, A(c_i))\} \leq k$. Because there are $|C||F|$ possible $co(c_i, A(c_i))$ values we can do $\log(|C||F|)$ binary searches using (k, r)-gathering algorithms to find the $\min_A \max_{c_i \in C}\{co(c_i, A(c_i))\}$ (the min-max value). In [4] Akagi and Nakano observed that the number of binary searches can be reduced to $O(\log(|C| + |F|))$.

For a facility f, the index of its left boundary is $l(f) = \min\{i||f - c_i| \leq k\}$ and its left boundary is $c_{l(f)}$ and the index of its right boundary is $r(f) = \max\{i||f - c_i| \leq k\}$ and its right boundary is $c_{r(f)}$. Two facilities $f_a < f_b$ are intersecting if $r(f_a) \geq l(f_b) - 1$.

To find out whether a (k, r)-gathering exits we first compute the (indices of) left and right boundaries for every facility. The algorithm for computing these will be explained in the next section. In this section we just assume we can have them. For a facility f, if $r(f) - l(f) + 1 < r$ then we close it.

We can assume that the customers assigned to a facility is consecutive. A consecutive r' customers going to a facility are called a complete interval if $r' \geq r$. If $r' < r$ then they are called an incomplete interval.

We will use the Left-to-Right Maximal Scan and the Right-to-Left Minimal Scan. The Left-to-Right Maximal Scan is shown below:

Left-to-Right Maximal Scan

1. Find the rightmost non-closing facility f_a with $|c_0 - f_a| \leq k$. Set $i = a$. Set $border = 0$.
2. Find the rightmost non-closing intersecting facility f_b to the right of f_i.

if f_b does not exist then there is no solution; exit;

if $l(f_b) > border + r - 1$ then
begin
Mark $c_{border}, c_{border+1}, ..., c_{l(f_b)-1}$ as a complete interval of customers going to f_i;
Set $border = l(f_b)$;
end
else
begin
if $r(f_b) \geq border + 2r - 1$ then
begin
Mark $c_{border}, c_{border+1}, ..., c_{border+r-1}$ as a complete interval of customers going to f_i;
Set $border = border + r$;
end
else goto Step 3;
end
if $r(f_b) = |C| - 1$ then
begin
Mark $c_{border}, c_{border+1}, ..., c_{|C|-1}$ as a complete interval of customers going to f_b;
(k, r)-gathering found; exit;
end
else
begin
$i = b$; goto Step 2;
end
3 /*Here we reached a breakpoint because f_i and f_b cannot have $2r$ customers going to them.*/
if $r(f_b) = |C| - 1$ then
begin
Mark $c_{border}, c_{border+1}, ..., c_{border+r-1}$ as a complete interval of customers going to f_i and mark $c_{border+r}, c_{border+r+1}, ..., c_{|C|-1}$ as an incomplete interval of customers going to f_b;
exit;
end
else
begin
Let f_c be the immediate next non-closing facility right to f_b;
Mark $c_{border}, c_{border+1}, ..., c_{border+r-1}$ as a complete interval of customers going to f_i;
If f_b is not intersecting f_c then there is no solution for a (k, r)-gathering and we exit;
Mark $c_{border+r}, c_{border+r+1}, ..., c_{l(f_c)-1}$ as an incomplete interval of customers going to f_b;
Treat $c_{l(f_c)}$ through $c_{|C|-1}$ and $f_c, f_{c+1}, ..., f_{|F|-1}$ as a separate problem using divide-and-conquer;
/* Here we say that we break between $c_{l(f_c)-1}$ and $c_{l(f_c)}$ and between f_{c-1} and f_c.*/
exit;
end

Note that the Left-to-Right Maximal Scan for all facilities takes $O(|F|)$ time after the left and right boundaries are computed. If the Scan results in no breakpoints then we obtained a (k, r)-gathering. We will say that such a Scan is a successful Scan. If there is only one breakpoint then this breakpoint results in one incomplete interval and it is at the rightmost position among all formed intervals. In the case there is only one breakpoint we will say that the Scan is a complete Scan.

Now the Right-to-Left Minimal Scan:

Right-to-Left Minimal Scan
1. Find the rightmost non-closing facility f_a with $|f_a - c_{|C|-1}| \leq k$. Set $i = a$. Set $border = |C| - 1$.
2. Find the rightmost intersecting neighbor f_b to the left of f_i such that $border - l(f_b) + 1 \geq 2r$.
if f_b does not exist then goto Step 3.
if $r(f_b) \leq border - r$ then
begin
Mark $c_{r(f_b)+1}, c_{r(f_b)+2}, ..., c_{border}$ as a complete interval of customers going to f_i;
Set $border = r(f_b)$;
end
else
begin
Mark $c_{border-r+1}, c_{border-r+2}, ..., c_{border}$ as a complete interval of customers going to f_i;
Set $border = border - r$;
end
if $l(f_b) = 0$ then
begin
Mark $c_0, c_1, ..., c_{border}$ as a complete interval of customers going to f_b;
(k, r)-gathering found; exit;
end
else
begin
$i = b$; goto Step 2;
end
3 /*We reached a breakpoint because f_b cannot have r customers going to it.*/
Let f_b be the leftmost facility left to f_i and intersects with f_i.
if $l(f_b) = 0$ then
begin
Mark $c_{border-r+1}, c_{border-r+2}, ..., c_{border}$ as a complete interval of customers going to f_i and mark $c_0, c_1, ..., c_{border-r}$ as an incomplete interval of customers going to f_b;
exit;
end
else
begin
Let f_c be the immediate next non-closing facility left to f_b;
Mark $c_{border-r+1}, c_{border-r+2}, ..., c_{border}$ as a complete

interval of customers going to f_i;

If $|c_{r(f_c)+1} - f_b| > k$ then there is no solution for a (k, r)-gathering and we exit;

Mark $c_{r(f_c)+1}, c_{r(f_c)+2}, ..., c_{border-r}$ as an incomplete interval of customers going to f_b;

Treat c_0 through $c_{r(f_c)}$ and $f_0, f_1, ..., f_c$ as a separate problem using divide-and-conquer;

/* We say that we break between $c_{r(f_c)}$ and $c_{r(f_c)+1}$ and between f_c and f_{c+1}.*/

exit;

end

Note that the Right-to-Left Minimal Scan for all facilities takes $O(|F|)$ time after the left and right boundaries are computed. If the Scan results in no breakpoints then we obtained a (k, r)-gathering. We will say that such a Scan is a successful Scan. If there is only one breakpoint then this breakpoint results in one incomplete interval and it is at the leftmost position among all formed intervals. In the case there is only one breakpoint we will say that the Scan is a complete Scan.

Let i be an interval of customers going to f. The extended interval of i is $\{c_{l(f)}, c_{l(f)+1}, ...c_{r(f)}\}$.

Lemma 1: If a complete Left-to-Right Maximal Scan S results in a set $S(S)$ of I intervals then at least I facilities has to open for a (k, r)-gathering.

Proof: Let A be any set of intervals and we will use $E(A)$ to denote the set of extended intervals in A. Assume that a (k, r)-gathering G has a set $S(G)$ of $I' < I$ complete intervals. Then there is an extended interval i_1 in $E(G)$ that proper contains an extended interval i_2 in $E(S)$ and the rightmost customer in i_1 is to the right of the rightmost customer in i_2. This can be seen by starting from the left side and going to the right, comparing extended intervals one in $E(S)$ against one in $E(G)$. This says that S is not a maximal scan as in the Left-to-Right Maximal Scan we always find the rightmost intersecting neighbor in Step 2. □

Lemma 2: If a complete Right-to-Left Minimal Scan S results in a set $S(S)$ of I intervals then at most $I-1$ facilities can open for a (k, r)-gathering.

Proof: Assume a (k, r)-gathering G has a set $S(G)$ of $I' > I - 1$ complete intervals. Let $E(G)$ be the set of extended intervals of $S(G)$. Let $E(S)$ be the set of extended intervals of $S(S)$. Then there is an extended interval i_1 in $E(S)$ that proper contains an extended interval i_2 in $E(G)$ and the leftmost customer in i_1 is to the left of the leftmost customer in i_2. This can be seen by starting from the right side and going to the left, comparing extended intervals one in $E(S)$ against one in $E(G)$. This says that S is not a minimal scan as in the Right-to-Left Minimal Scan we always find the rightmost intersecting neighbor in Step 2. □

Lemmas 1 and 2 explains why the Left-to-Right Maximal Scan is called a maximal scan and why the Right-to-Left Minimal Scan is called a minimal scan.

Lemma 3: If a complete Left-to-Right Maximal Scan has I_{max} intervals and a complete Right-to-Left Minimal Scan has I_{min} intervals then $I_{min} \geq I_{max}$.

Proof: From Lemmas 1 and 2. □

Theorem 1: Assume we have a complete Left-to-Right Maximal Scan S_{max} with I_{max} intervals and a complete Right-to-Left Minimal Scan S_{min} with I_{min} intervals. If $I_{max} = I_{min}$ then there is no solution for a (k, r)-gathering. If $I_{max} < I_{min}$ then the two Scans can be combined into a solution for (k, r)-gathering.

Proof: If $I_{max} = I_{min}$ then Lemma 1 says that any (k, r)-gathering has $\geq I_{max}$ facilities open while Lemma 2 says that any (k, r)-gathering has $< I_{min}$ facilities open. Thus it is impossible to have a (k, r)-gathering.

If $I_{max} < I_{min}$ then there is a complete interval i_{min} created in S_{min} that is contained in a complete interval i_{max} created in S_{max}. Let $c_{min,l}$ be the leftmost customer in i_{min}, $c_{min,r}$ be the rightmost customer in i_{min}, $c_{max,l}$ be the leftmost customer in i_{max} and $c_{max,r}$ be the rightmost customer in i_{max}. Let i_{min} be the j_{min}-th interval counting from right to left created by S_{min} and i_{max} be the j_{max}-th interval counting from left to right created by S_{max}. We create a (k, r)-gathering by using the 0th through $(j_{max}-1)$-th intervals created by S_{max} and the 0th through $(j_{min}-1)$-th intervals created by S_{min}. We then add a complete interval for $c_{max,l}$ through $c_{min,r}$ and let them go to the facility opened in S_{max} for $c_{max,l}$ through $c_{max,r}$. This creates a (k, r)-gathering. We say that we combined S_{max} with S_{min} at i_{max} and i_{min}. □

Now we consider the situation where we have multiple breakpoints in the Scans. We use the following Fix procedure:

Fix

1. Start with the Right-to-Left Minimal Scan S_{min}.

2. if S_{min} is successful then we obtained a (k, r)-gathering and we exit;

else we stop when we reach the first breakpoint. This breakpoint partitions the customers into two sets $\{c_0, ..., c_{a-1}\}$ and $\{c_a, ..., c_{|C|-1}\}$ and partitions the facilities into two sets $\{f_0, ..., f_{b-1}\}$ and $\{f_b, ..., f_{|F|-1}\}$. $\{c_a, ..., c_{|C|-1}\}$ has been put into $I(S_{min})$ intervals (with one incomplete interval at the leftmost position and other $I(S_{min}) - 1$ complete intervals).

3. Now we start the Left-to-Right Maximal Scan S_{max} for $\{c_a, ..., c_{|C|-1}\}$.

4. /* If S_{max} is successful then S_{max} created $\leq I(S_{min})-1$ complete intervals by Lemma 2.*/

5. (Case 1) If S_{max} is successful or is complete with $\leq I(S_{min}) - 1$ intervals then we find the *leftmost* (complete) interval i_{max} created by S_{max} that contains a (complete) interval i_{min} created by S_{min} and combine the intervals created by S_{max} and S_{min} at i_{max} and i_{min} to get a (minimal) solution for the (k, r)-gathering for $\{c_a, ..., c_{|C|-1}\}$. If there are more than one complete

intervals created by I_{min} that are contained in i_{max} then we will pick the *leftmost* one to be combined with i_{max}. It is a minimal solution because we used "*leftmost*".

6. (Case 2) If S_{max} is complete with $I(S_{min})$ intervals then by Theorem 1 there is no solution for a (k,r)-gathering for $\{c_a, ..., c_{|C|-1}\}$. If there is a solution S for a (k,r)-gathering for $\{c_0, c_1, ..., c_{|C|-1}\}$ then let f be the facility c_a goes to in S. Let I be the number of open facilities to the right of and including f opened by S. Because c_a goes to f in S therefore f is to the right of f_c where $c_{r(f_c)} = c_{a-1}$ and f_c would be the first open facility if we would resume the Right-to-Left Minimal Scan after the first breakpoint. Thus the extended interval of f intersects with the extended interval of the last complete interval of S_{min}. However, S_{min} reached a breakpoint and thus $I \leq I(S_{min}) - 1$ by Lemma 2 (Note here that we may move the breakpoint to between $c_{l(f)-1}$ and $c_{l(f)}$). However, if we take the set F_o of open facilities right to and include f opened by S, then $|F_o| \leq I(S_{min}) - 1$. On the other hand S_{max} has $I(S_{min})$ intervals and thus the number of intervals in F_o has to be $\geq I(S_{min})$. This contradiction says that there is no solution for (k,r)-gathering for $\{c_0, c_1, ..., c_{|C|-1}\}$. Exit.

7. (Case 3) If S_{max} is not complete. Then stop at the first breakpoint of S_{max}. This breakpoint will partition $\{c_a, ..., c_{|C|-1}\}$ into $P = \{c_a, ...c_{a_1}\}$ and $\{c_{a_1+1}, ..., c_{|C|-1}\}$ and partition $\{f_b, ..., f_{|F|-1}\}$ into $\{f_b, ..., f_{b_1}\}$ and $\{f_{b_1+1}, ..., f_{|F|-1}\}$. Let c_{a_1} be a member of a complete interval i created by S_{min}. If c_{a_1} is not the rightmost customer in i then we add $c_{a_1+1}, c_{a_1+2}, ..., c_{a_2}$ to P, where c_{a_2} is the rightmost customer in i. The situation for customers $\{c_a, .., c_{a_2}\}$ can be analyzed in the same way as we analyzed in Steps 5 and 6.

Theorem 2: After the left and right boundaries have been computed we can find whether a solution for a (k,r)-gathering exists in $O(|F|)$ time.

Alternatively after computing the boundaries we can use the $O(|F|)$ time decision algorithm in [4].

3. Computing Left and Right Boundaries

For two neighboring facilities f_a and f_{a+1}, let $2r$ customers $F_{a,a+1} = \{c_b, c_{b+1}, ..., c_{b+2r-1}\}$ be such that $|f_a - c_{b+2r-2}| < |f_{a+1} - c_{b-1}|$ and $|f_a - c_{b+2r}| > |f_{a+1} - c_{b+1}|$. $F_{a,a+1}$ is called the boundary set of customers between f_a and f_{a+1}.

Lemma 4: Let $F_{a,a+1} = \{c_b, c_{b+1}, ..., c_{b+2r-1}\}$ be the boundary set of f_a and f_{a+1}, then in an optimal r-gathering or an (k,r)-gathering c_d, $d > b + 2r - 1$, will not go to facility f_a and c_e, $e < b$, will not go to facility f_{a+1}.

Proof: Suppose in an optimal r-gathering or a (k,r)-gathering c_{b+2r} goes to facility f_a. Let the leftmost customer going to f_a be c_t. If $t \geq b + 1$ then we can delete f_a and let all customers going to f_a now go to f_{a+1}. If $t \leq b$ then we can let $c_{b+r}, c_{b+r+1}, ..., c_l$ go to f_{a+1}, where c_l was the rightmost customer of f_a.

The other situation can be proved similarly. □

In order to use Lemma 4 we need place a dummy customer d_l at the left of f_0 and a dummy customer d_r at the right of $f_{|F|-1}$ and let $|f_0 - d_l|$ and $|f_{|F|-1} - d_r|$ larger than $\max\{|f_0 - c_{|C|-1}|, |f_{|F|-1} - c_0|\}$.

We will let $ll(f_{a+1}) = rl(f_a) = c_b$ and $lr(f_{a+1}) = rr(f_a) = c_{b+2r-1}$.

Lemma 4 says that for computing an optimal r-gathering or a (k,r)-gathering we need consider no more than $4r$ distances corresponding to customers in $[ll(f_a), lr(f_a)]$ and $[rl(f_a), rr(f_a)]$ for each facility. Thus the total number of distances to be considered is $4|F|r$. We may collect all these $4|F|r$ distances and then do binary search $\log(4|F|r)$ times to find the minimum k value for an optimal r-gathering. This will result in $O(|C|+|F|r\log(|F|r)+|F|(\log r)(\log(|F|r)))$ time for r-gathering by (1) preprocess them in $O(|C| + |F|)$ time to compute the boundary sets $F_{a,a+1}$, (2) sort $4|F|r$ distances in $O(|F|r\log(|F|r))$ time, (3) binary search $\log(4|F|r)$ rounds among the $4|F|r$ possible minimum distances where each round consists of computing the left and right boundaries in $O(|F|\log r)$ time and Right-to-Left Minimal Scan and Left-to-Right Maximal Scan in $O(F)$ time.

We maintain the set M of possible minimum costs, then repeatedly compute the median k of possible minimum costs, then compute left and right boundaries and call Right-to-Left Minimal Scan and Left-to-Right Maximal Scan for value k to find whether a (k,r)-gathering exits. If it returns YES then the minimum cost is less than or equal to k and we can remove the larger half of costs from M. If it returns NO then the minimum cost is larger than k and we can remove the smaller half of costs from M. After $\log(4|F|r)$ rounds we can find the minimum cost k^*. In later sections we show we can do better than this.

4. r-gathering on the line

If all C and F are on the line, an $O((|C|+|F|)\log(|C|+|F|))$ time algorithm to solve the r-gathering problem is known[4]. In this section we give a faster algorithm. Our algorithm runs in $O(|C|+|F|\log^3 r+|F|\log|F|\log r)$ time. Since $C >> F$ and $C >> r$ holds in general, or if we can assume r as a constant, our algorithm is faster.

We can observe that the minimum cost k^* of a solution of an r-gathering problem is $co(c,f)$ for some $c \in C$ and some $f \in F$. Since the number of possible minimum cost, say some $co(c,f)$, is at most $4|F|r$ by Lemma 4, one can find the minimum cost in $O(|C| + |F|r\log(|F|r) + |F|(\log r)(\log(|F|r))$ time as we explained before.

However we can design a faster algorithm which runs in $O(|C| + |F|\log^3 r + |F|\log|F|\log r)$ time. Our algorithm maintains a set M of possible minimum costs, then repeatedly computes the "median of medians" k, defined below,

then call Right-to-Left Minimal Scan and Left-to-Right Maximal Scan for k. Depending whether a (k,r)-gathering exists the algorithm removes some subset of possible minimum costs from M. After $O(\log^2 r)$ rounds M has at most $2|F|$ distances remaining, then we can find the minimum cost k^* by an ordinary binary search. Now we explain the detail.

Set initially $M\ell(f_j) = \{co(c_i, f_j)|c_i \in [ll(f_j), lr(f_j)]\}$, and $Mr(f_j) = \{co(c_i, f_j)|c_i \in [rl(f_j), rr(f_j)]\}$. We are going to repeatedly remove the half of distances from some $M\ell(f_j)$ and/or $Mr(f_j)$. M is the set of all $M\ell(f_j)$ and $Mr(f_j)$, $j = 1, 2, \cdots, |F|$, however if $M\ell(f_j)$ has exactly one distance then $M\ell(f_j)$ is removed from M. Similar for $Mr(f_j)$.

We will use a weighing scheme similar to the one used in [5]. If $M\ell(f_j)$ has $2r/2^x$ customers we define the weight $w\ell(f_j)$ of $M\ell(f_j)$ as $(1+\log r - x)$. The weight $wr(f_j)$ of $Mr(f_j)$ is defined similarly. The weight of M is the sum of the weights of $M\ell(f_j)$ and $Mr(f_j)$ in M.

Initially each $M\ell(f_j)$ has exactly $2r$ customers, so $x = 0$, and its weight is $1 + \log r$. So initially the weight of M is $2|F|(1 + \log r)$.

Say there are $N \le 2|F|$ $M\ell(f_j)$'s and $Mr(f_j)$'s with more than one distance remaining and the total weights in them is T. In each round we find the median of $M\ell(f_j)$ and the median of $Mr(f_j)$ and this gives us N medians. This takes constant time for each facility. We then find the median k of these N medians and this takes $O(|F|)$ time. Say that k is the median of $M\ell(f_i)$ then we place all $M\ell(f_j)$'s and $Mr(f_j)$'s whose median is $< k$ above $M\ell(f_i)$ and all $M\ell(f_j)$'s and $Mr(f_j)$'s whose median is $> k$ below $M\ell(f_i)$. Because k is the median of the medians we have put half $(N/2)$ of $M\ell(f_j)$'s and $Mr(f_j)$'s above $M\ell(f_i)$ and the other half $(N/2)$ of $M\ell(f_j)$'s and $Mr(f_j)$'s below $M\ell(f_i)$. If a $(k.r)$-gathering exists then we remove half of the distances in each of the $M\ell(f_j)$'s and $Mr(f_j)$'s below $M\ell(f_i)$. If a (k,r)-gathering does not exists then we remove half of the distances in each of the $M\ell(f_j)$'s and $Mr(f_j)$'s above $M\ell(f_i)$'s. Thus in any case we remove half of the distances from half of the $M\ell(f_j)$'s and $Mr(f_j)$'s. Thus we remove total $N/2$ weights with one weight from each of the $M\ell(f_j)$'s or $Mr(f_j)$'s from which we removed half of the distances. Let us say that $M\ell(f_j)$ has $2r/2^x$ distances remaining and thus has weight $1+\log r - x$ and we removed half of distances in it and thus removed one weight. Then we removed $(1/(1+\log r - x))$-th $\ge (1/(1+\log r))$-th weight from it. If we pair one $M\ell(f_j)$ from which we removed half of the distances and one $M\ell(f_t)$ from which we did not remove half of the distances and say that $M\ell(f_j)$ has $2r/2^x$ distances and $\log r + 1 - x$ weights and $M\ell(f_t)$ has $2r/2^y$ distances and $\log r+1-y$ weights then the one weight we removed from $M\ell(f_j)$ is at least $1/(2(\log r + 1))$-th of the sum of the weights of $M\ell(f_j)$ and $M\ell(f_t)$. This says that in one round we reduce weights from T to at most $T(1 - 1/(2(\log r + 1))$. Initially we have $2|F|(1 + \log r)$ weights. So after $4(1 + \log r)\log r$ rounds the weights are at most

$$2|F|(1+\log r)(1-1/(2(1+\log r)))^{(2(1+\log r))2\log r} = 2|F|(1+\log r)(1/e$$

$$\le 2|F|(1+\log r)(1/2)^{(2\log r)} = 2|F|(1+\log r)/r^2 \le |F|/r.$$

After $4(1 + \log r)\log r$ rounds, as explained above, the weight T is at most $|F|/r$. Since each weight accounts for $2r/2^x$ customers for some $1 \le x \le \log r + 1$, one weight always account for at most r customers. Thus the number of remaining distances is at most $|F|$ because weights $T \le |F|/r$. Note that we have to place back the the last remaining distance in $M\ell(f_j)$'s and $Mr(f_j)$'s where all distances except one have been removed.There are iat most $2|F|$ of them. Thus we have at most $3|F|$ distances remaining.

Finally sort the remaining $3|F|$ remaining distances in $O(|F|\log|F|)$ time, then binary search them $\log(3|F|)$ rounds each of which takes $O(|F|\log r)$ time for computing the left and right boundaries and $O(|F|)$ time for Right-to-Left Minimal Scan and Left-to-Right Maximal Scan. Then we find the minimum cost.

Theorem 3: One can solve the r-gathering problem in $O(|C| + |F|\log^3 r + |F|\log|F|\log r)$ time when all C and F are on the real line. □

5. Tighter Analysis

In this section we analyze the running time of our algorithm in the preceding section more tightly.

We analyze again the running time to compute the boundaries in Section 3, in which we find some indices from $[ll(f_j), lr(f_j)]$ and $[rl(f_j), rr(f_j)]$ for each $f_j \in F$ by binary search. We repeat this in $O(\log^2 r)$ rounds.

For the first round we find the boundaries by binary search from the $2r$ distances. However for later round the number of distances from which we find the boundary is smaller.

Assume that for the first round the number of computation to compute the boundaries is at most $c|F|\log r$ for some constant c. For the second round the number of computation for the boundaries is at most

$$c|F|\log r/2 + c|F|(\log r - 1)/2) \quad (1)$$

$$= c|F|\log r(1/2 + 1/2 - 1/(2\log r)) \quad (2)$$

$$= c|F|\log r(1 - 1/(2\log r)). \quad (3)$$

So for the x-th round the number of computation for the boundaries is at most $c|F|\log r(1 - 1/(2\log r))^{x-1}$ Thus the total number of computation for the boundaries for all round is at most

$$c|F|\log r + c|F|\log r(1-1/(2\log r)) + \cdots + c|F|\log r(1-1/(2\log r))^{x-1}$$

Except for the computation for the boundaries above and the computation for the weighted median, which runs in $O(|F|)$ time for each round and $O(|F|\log^2 r)$ time in total, the algorithm consists of $O(\log^2 r)$ rounds, in which

each round call Right-to-Left Minimal Scan and Left-to-Right Maximal Scan, which runs in $O(|F|)$ time. This will accounts for $O(|C| + |F| \log^2 r)$ time. After that there are $3|F|$ distances remaining, and we use $O(\log |F|)$ rounds in which each x-th round computes the median of $3|F|/2^{x-1}$ distances, computes the left and right boundaries and call Right-to-Left Minimal Scan and Left-to-Right Maximal Scan, which runs in $O(|F| \log r)$ time.

Thus the running time of the algorithm is $O(|C| + |F| \log^2 r + |F| \log |F| \log r)$.

Note that after $M \leq 3|F|$ distances remaining, each round consists of finding the median (value k) in $O(|M|)$ time, compute left and right boundaries and this takes $O(|M|)$ time as follows. Assume that m_i distances are from f_i, that is $co(c, f_i)$ for some c. We have $\sum_i \log m_i = O(M)$ since $\sum_i m_i = M$. Thus we need $O(|F|)$ time for each round and $O(|F| \log |F|)$ time over all rounds.

Theorem 4: Optimal r-gathering of $|C|$ customers and $|F|$ facilities can be found in $O(|C| + |F| \log^2 r + |F| \log |F|)$ time. □

6. Conclusion

In this paper we have given an algorithm to solve the r-gathering problem when all C and F are on the real line. The running time of the algorithm is $O(|C| + |F| \log^2 r + |F| \log |F|)$ and faster than the known algorithm in [4].

References

[1] G. Aggarwal, T. Feder, K. G. Aggarwal, T. ederS, K. Kenthapadi, S. Khuller, R. Panigrahy, D. Thomas and A. Zhu, Achieving anonymity via clustering, Transactions on Algorithms, 6, Article No.49 (2010).

[2] P. Agarwal and M. Sharir, Efficient Algorithms for Geometric Optimization, Computing Surveys, 30, pp.412-458 (1998).

[3] T. Akagi and S. Nakano, On (k, r)-gatherings on a Road, Proc. of Forum on Information Technology, FIT 2013, RA-001 (2013).

[4] T. Akagi and S. Nakano, On r-gatherings on the Line, Proc.of FAW 2015, Guilin, Guangxi, China LNCS 9130, pp.25-32 (2015).

[5] R. J. Anderson, G. L. Miller. Deterministic parallel list ranking. Algorithmica, Vol. 6, pp. 859-868 (1991).

[6] A Armon, On min-max r-gatherings, Theoretical Computer Science, 412, pp.573-582 (2011).

[7] Z. Drezner, Facility Location: A Survey of Applications and Methods, Springer (1995).

[8] Z. Drezner and H.W. Hamacher, Facility Location: Applications and Theory, Springer (2004).

[9] G. Frederickson and D. Johnson, Generalized Selection and Ranking: Sorted Matrices, SIAM Journal on Computing, 13, pp.14-30 (1984).

[10] H. Fournier, and A. Vigneron, Fitting a Step Function to a Point Set, Proc of ESA 2008, Lecture Notes in Computer Science, 5193, pp.442-453 (2008).

[11] J. Y. Liu, A Randomized Algorithm for Weighted Approximation of Points by a Step Function, Proc. of COCOA 2010, Lecture Notes in Computer Science, 6508, pp.300-308 (2010).

A segment partitioning heuristic for scheduling jobs with release times and due-dates

Nodari Vakhania[a],
Federico Alonso-Pecina[b], Crispin Zavala[b]
(a) Centro de Investigación en Ciencias, UAEMor, Mexico.
(b) Facultad de Contaduría, Administración e Informática UAEMor, Mexico.

***Abstract*—In a standard strongly NP-hard single-machine scheduling problem the jobs are characterized by release times and due-dates and the objective to minimize the maximum job lateness. We develop a heuristic method for solving this problem based on the partition of the schedule horizon into two types of time intervals containing urgent and non-urgent jobs, respectively. We report the results of the preliminary computational experiments testing the practical performance of the proposed algorithm.**

Key words: scheduling single-machine; release-time; due-date; lateness; bin packing; polynomial-time algorithm

I. Introduction

In scheduling problems have a finite set of *resources* (machines or processors) that may perform *orders* (jobs or tasks) from another finite set. The objective is to arrange the assignment of the orders to the resources to minimize some overall (usually time) criteria.

In this paper we address a single-machine scheduling problem when every job j is characterized by its *release time* r_j and *due-date* d_j; r_j is the time moment when job j arrives to the system hence becomes available for processing on the machine, and d_j is the desired completion time for job j. The problem of scheduling jobs with release times and due-dates on a single machine with the objective to minimize the maximum job lateness, with the common abbreviation $1/r_j/L_{\max}$ (Graham et al. [4]), can be stated as follows. We are given n jobs in $\{1, 2, \ldots, n\}$. Each job j has (non-interruptible) processing time p_j, release time r_j and due-date d_j. The n jobs are to be scheduled on a single machine that can process at most one job at a time. A *feasible schedule* S is a mapping that assigns to each job j a starting time $t_j(S)$, such that $t_j(S) \geq r_j$ and $t_j(S) \geq t_k(S) + p_k$, for any job k included earlier in S (for notational simplicity, we use S also for the corresponding job-set); the first inequality says that a job cannot be started before its release time, and the second one reflects the restriction that the machine can handle only one job at any time. $c_j(S) = t_j(S) + p_j$ is the completion time of job j. We aim to find out if there is a schedule which meets all job due-dates, i.e., every j is completed by time d_j. If there is no such schedule then we look for an optimal schedule, i.e., one minimizing the maximum job *lateness* $L_{\max} = \max\{j|c_j - d_j\}$. We denote by $L(S)$ ($L_j(S)$, respectively) the maximum lateness in S (the lateness of job j in S, respectively).

The problem is known to be strongly NP-hard (Garey & Johnson [2]). Hence, the development of efficient heuristics with a good practical behavior is of a primary interest. The earliest proposed and the most widely used heuristics for an approximate solution of problem $1/r_j/L_{\max}$ is the ED (Earliest Due-date) heuristic, suggested by Jackson [6]. This heuristic, iteratively, at each scheduling time t (given by job release or completion time), among the jobs released by time t schedules one with the largest delivery time or the smallest due-date (breaking ties by selecting a longest one).

In the worst-case, Jackson's heuristic delivers a solution which is twice worse than an optimal one, i.e., ED-heuristic is a 2-approximation algorithm. Potts [8] has proposed an alternative approximation algorithm with an improved approximation ratio of $3/2$, in which Jackson's heuristic is repeatedly applied $O(n)$ times. Hall and Shmoys [5] have proposed polynomial approximation schemes for the same problem, and also an 4/3-approximation an algorithm for its version with the precedence relations with the same time complexity of $O(n^2 \log n)$ as the above algorithm from [8].

Implicit enumerative algorithms have also been developed for problem $1/r_j/L_{\max}$. Among the most efficient such algorithms are ones proposed by McMahon & Florian [7] and Carlier [1].

The problem can naturally be simplified by imposing some restrictions on job processing times. Two such versions are known to be polynomially solvable. Garey et al. [3] have developed a sophisticated $O(n \log n)$ algorithm for the case when all jobs have equal integer length p (abbreviated $1/p_j = p, r_j/L_{\max}$). Later in [10] was proposed an $O(n^2 \log n \log p)$ algorithm solving a more general setting when a job processing time can be either p or $2p$ (abbreviated $1/p_j \in \{p, 2p\}, r_j/L_{\max}$).

Recently in [11] certain conditions which satisfaction guarantees the obtainment of an optimal solution to problem $1/r_j/L_{\max}$ were presented. These conditions take an advantage of a close relationship between the scheduling problem and a version of the bin packing problem with different bin capacities. The heuristic method that we build here also takes an advantage of this relationship. The schedules that we create are partitioned into two types of intervals, containing, roughly classifying, urgent and non-urgent jobs. We call the intervals containing urgent jobs kernel intervals, and the intervals containing non-urgent jobs bin intervals. In every optimal schedule, kernel jobs form a tight sequence in the sense that

the *delay* of its earliest scheduled job (*i.e.*, the difference between the starting and release times of that job) cannot exceed some precalculable magnitude $\delta \in [0, p_{\max}]$, where $p_{\max}$ is the maximal job processing time.

Because of a little degree of the flexibility, it is easier to arrange kernel intervals. Our heuristic method uses ED-heuristic to schedule these intervals. The rest of the scheduling horizon consists of the bin intervals, within which all the non-urgent jobs are to be distributed. Our task is then to find a proper such job distribution. We use a variation of LPT (Longest Processing Time) heuristic to find such distribution of the non-urgent jobs. The LPT-heuristic, iteratively, at each scheduling time t (given by job release or completion time), among the jobs released by time t schedules one with the largest processing time (breaking ties by selecting a most urgent one).

The practical behavior of our algorithm was tested for a number of randomly generated problem instances, described in the concluding section.

II. Preliminary concepts and notions

From here on, let S be an ED-schedule, one created by ED-heuristic.

Schedule S may contain a *gap*, that is its maximal consecutive time interval in which the machine is idle. We assume that there occurs a 0-length gap (c_j, t_i) whenever job i starts at its release time immediately after the completion of job j.

A *block* in S is its consecutive part consisting of the successively scheduled jobs in without any gap in between, which is preceded and succeeded by a (possibly a 0-length) gap.

Given schedule S, let i be a job that realizes the maximum job lateness in S, i.e., $L_i(S) = \max_j\{L_j(S)\}$. Let, further, B be the block in S that contains job i. Among all the jobs in B with this property, the latest scheduled one is called an *overflow job* in S (not necessarily it ends block B).

Note that if schedule S contains two or more overflow jobs then they belong to different blocks in S.

A *kernel* in S is a maximal (consecutive) job sequence ending with an overflow job o such that no job from this sequence has a due-date more than d_o. For a kernel K, we let $r(K) = \min_{i\in K}\{r_i\}$, and will denote by $L(K)$ the maximum lateness of a job in K.

It follows that every kernel is contained in some block in S, and the number of kernels in S equals to the number of the overflow jobs in it. Furthermore, since any kernel belongs to a single block, it may contain no gap.

In schedule S, the *delay* of kernel K is the difference between the starting time of its earliest scheduled job and $r(K)$.

Observation 1: The maximum job lateness in a kernel K cannot be reduced if it has no delay (i.e., the earliest scheduled job in K starts at time $r(K)$). Hence, if an ED-schedule S contains a kernel with this property, then it is optimal.

Proof. Recall that all jobs in K are no less urgent than the overflow job o, and that jobs in K form a tight sequence (i.e., without any gap). Then since the earliest job in K starts at its release time, no reordering of jobs in K can reduce the current maximum lateness, which is $L_o(S)$. Hence, there is no feasible schedule S' with $L(S') < L_o(S)$, i.e., S is optimal. □

Due to the above observation, assume, without loss of generality, that the condition in Observation 1 does not hold. Then there exists a job, less urgent than o, scheduled before all jobs in K that delays the starting of jobs in K. By rescheduling such a job to a later time moment behind K, the jobs in K can be restarted earlier. We define now this operation formally.

Suppose i precedes j in S. We will say that i *pushes* j in S if ED-heuristic will reschedule j earlier if i is discarded.

It follows that the earliest scheduled job of every kernel is immediately preceded and pushed by a job e with $d_e > d_o$. In general, we may have more than one such a job scheduled before kernel K in block B (one containing K). We call such a job an *emerging job* for K, and we call the latest scheduled one (job e above) the *delaying* emerging job.

Aiming in restarting the kernel jobs earlier, we may *activate* an emerging job e for K; *that is*, we force e and all passive emerging jobs to be rescheduled after K (the latter jobs are also said to be in the state of activation for K). This we achieve by increasing the release times of all these jobs to a sufficiently large magnitude, say $r(K)$, so that when ED-heuristic is newly applied, neither job e nor any passive emerging job will surpass any kernel job, and hence the earliest job in K will start at time $r(K)$. We note that more than one emerging job can be activated for K and the same emerging job may be activated for two or more successive kernels.

III. The heuristic

As we have mentioned in the introduction, our heuristic is based on the idea of partitioning the scheduling horizon into the urgent (kernel) and non-urgent (bin) intervals. It consists of the two basic stages. First, at the *partitioning stage*, all the kernel and bin intervals are determined. At the *construction stage*, kernel and bin intervals are filled in by urgent and the non-urgent jobs, respectively.

A. The partitioning stage

We have implemented two versions for extracting the kernel intervals at the partitioning stage. In both of these versions, the initial ED-schedule σ is created; σ is obtained by ED-heuristic, which is applied to the originally given problem instance. In schedule σ, one or more kernels in different blocks (with the same value of the maximum job lateness, may arise). Note that the corresponding overflow jobs have the same lateness and they pertain to different block in σ. This set of kernels in schedule σ form the initial set of kernels.

If we will have a deeper look into the structure of the ED-schedules we may see that extra potential kernels may be "hidden" within schedule σ. Consider a simple instance with three jobs with the parameters: (job 1) $r_1 = 0, p_1 =$

$10, d_1 = 100$, (job 2) $r_2 = 1$, $p_2 = 3$, $d_2 = 4$ and (job 3) $r_3 = 5, p_3 = 3, d_3 = 9$.

Since at time 0 only job 1 is released, the initial schedule σ assigns job 1 at time 0, then it assigns job 2 right at the completion time 10 of job 1, and finally it assigns job 3 at time $10 + 3 = 13$. There is a single kernel in σ consisting of job 2, which is the overflow job with the lateness $10+3-4 = 9$, whereas job 1 is the delaying emerging job. Note that the lateness of job 3 in σ is $13 + 3 - 9 = 7$.

If we activate the delaying emerging job 1 for the above kernel, we obtain another ED-schedule σ_1, in which job 1 starts at time 1 and completes at time 4 with 0 lateness; at that completion time, only job 1 is released, hence it is assigned at time 4 and is completed at time 14; at that time, job 3 is assigned. The lateness of job 3 in schedule σ_1 is $14+3-9 = 8$. Hence, there arises a new kernel consisting of a single job 3 in schedule σ_1 (the former kernel of schedule σ consisting of job 2 disappears in σ_1).

During the partitioning stage of our heuristic, the augmentation of the initial set of kernels in schedule σ by the above kind of the "hidden" kennels yields an improved, more accurate, performance results.

The kernel augmentation procedure has two versions. In the first one, whenever a new kernel arises, the delaying emerging job is temporally omitted, the corresponding kernel is rescheduled by ED-heuristic (being correspondingly left-shifted), and the construction proceeds similarly by ED-heuristic (with the rescheduled kernel though) until another kernel is encountered or schedule σ^*, consisting of all the jobs except the omitted delaying emerging jobs, is constructed. Note that the earliest scheduled job of every arisen during the procedure kernel K will start at its release time $r(K)$ in σ^*.

Let L_i^* be the (reduced) lateness of a kernel job i in σ^*, and let $\delta(K) = L^* - L(K)$. Since every kernel K is restarted at time $r(K)$ in σ^*, $L^*(K) = \max_{i \in K}\{L_i^*\}$, and hence $L^* = \max_\kappa\{L^*(K_\kappa)\}$ are lower bounds on the objective value:

Observation 2: The maximum lateness in schedule σ^* obtained on the partitioning stage is a lower bound on the optimal objective value.

Proof. By the definition of schedule σ^*, every kernel K arisen during the partitioning stage starts at its earliest possible starting time $r(K)$ in σ^*. Then our claim immediately follows from Observation 1. □

The kernels intervals can be defined with some degree of the flexibility, due to the observation.

Observation 3: Every kernel K can be delayed by $\delta(K)$ without increasing the maximum lateness.

Proof. Let K' be a kernel that realizes $\max_\kappa\{L^*(K_\kappa)\}$. By definition of $\delta(K)$, the completion time of every job in $K \neq K'$ can be increased by $\delta(K)$ so that none of the jobs in K will be completed later than a job realizing $\max_{i \in K'}\{L_i^*\}$. This clearly proves the observation. □

From Observation 3, we may assert that in an optimal schedule S_{opt} every kernel K starts either no later than at time $r(K) + \delta(K)$ or it is delayed by some $\delta \geq 0$ (the latter delay, as we will see later, may be unavoidable for a proper accommodation of the non-kernel jobs). Let $\Delta = L_o(\sigma) - L^*$, where o is an overflow job in σ. Then note that the maximum lateness in any feasible ED-schedule in which the delay of some kernel is more than Δ is no less than that in σ, i.e. Hence, no such schedule will be created by our heuristic.

We shall refer to the magnitude $L^* + \delta$ ($0 \leq \delta \leq \Delta$) as the *$\delta$-boundary*.

Recall that the first version of the kernel augmentation procedure, in schedule σ^*, each delaying job is omitted. In the second version of the procedure, every delaying emerging job is activated for the corresponding kernel. Thus the second version of the kernel augmentation procedure is similar to the first one, with the difference that, for every arisen kernel, the corresponding delaying emerging job is activated for that kernel (instead of being omitted).

B. The construction stage

At the construction stage, the heuristic schedules kernel jobs so that none of them surpasses δ-boundary, for any given choice of δ. The kernel intervals are given some degree of flexibility, depending on the value of δ according to Observation 3. The value of δ can be taken arbitrarily from the interval $[0, \Delta]$. In general, we have a bin between two adjacent kernel intervals, and a bin before the first and after the last kernel interval. Because of the allowable right-shift (Observation 3) the starting and completion times of the corresponding kernel and bin intervals are defined with the allowable flexibility, determined by the current value of the parameter δ (note that, since there may exist no gap within any kernel segment, the length of every kernel interval and hence the corresponding bin intervals are fixed).

Bin intervals are scheduled by LPT-heuristic so that the bin interval before every kernel K is extended up to the time moment $r(K) + \delta(K) + \delta$. If the next job selected by LPT-heuristic completes by time $r(K) + \delta(K) + \delta$, it is scheduled the next; otherwise, among the available jobs, the next shortest job is similarly selected, until none of the released jobs fits into the bin (within still available interval before time moment $r(K) + \delta(K) + \delta$). Then the next bin is similarly scheduled until all bins are scheduled.

IV. Preliminary computational experiments and final remarks

We have implemented our heuristic (with both versions of the kernel augmentation procedure) in Java using the development environment Eclipse IDE for Java Developers (version Luna Service Release 1 (4.4.1)) under Windows 8.1 operative system for 64 bits, and have used a laptop with Intel Core i7 (2.4 GHz) and 8GB of RAM DDR3 to run the code. The inputs in our main program are plain texts with job data that we have generated randomly, as we briefly describe below. The program for the generation of our instances was

constructed under the same development environment as our main program.

The computational experiments are at an early stage of development, still an ongoing research. So far, job release times and due dates were generated with the $rdn()$ function in Java, with an open range $(0, 50n)$, where n is the number of jobs in a corresponding instance. The processing times were generated from the interval $[1, 50]$ and also from the interval $[1, 100]$.

For a majority of the created problem instances the heuristic with the second version of the kernel augmentation procedure gave a solution with the objective value equal to the corresponding lower bound (as in Observation 2), whereas about 60% of the solutions with the first version of the kernel augmentation procedure achieved this lower bound. Since the heuristic runs in time $n \log n$, all the instances were solved instantly.

In the instances that were not solved optimally, the activated delaying emerging jobs have converted to the overflow jobs, hence the objective value could have been improved. We intend to extend the heuristic with an additional subroutine dealing with that kind of scenario. This, we believe, will improve its performance. Besides, we plan to test the heuristic for larger amount of problem instances, also generated randomly but in several different ways. For instance, the set of jobs can be divided into two or more subsets and job parameters for each subset can be derived independently, from different time intervals.

REFERENCES

[1] J. Carlier (1982). The one–machine sequencing problem. *European J. of Operational Research*. 11, 42–47.

[2] Garey M.R. and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP–completeness (Freeman, San Francisco, 1979)

[3] M.R. Garey, D.S. Johnson, B.B. Simons and R.E. Tarjan. Scheduling unit–time tasks with arbitrary release times and deadlines. *SIAM J. Comput.* 10, 256–269 (1981)

[4] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5: 287 – 328, 1979.

[5] L.A. Hall and D.B. Shmoys. Jackson's rule for single-machine scheduling: Making a good heuristic better, Mathematics of Operations Research 17 22–35 (1992)

[6] Jackson J.R. Scheduling a production line to minimize the maximum lateness. *Management Science Research Report 43, University of California, Los Angeles* (1955)

[7] G. McMahon and M. Florian. On scheduling with ready times and due-dates to minimize maximum lateness. *Operations Research.* 23, 475–482 (1975)

[8] C.N. Potts. Analysis of a heuristic for one machine sequencing with release dates and delivery times. *Operations Research* 28, p.1436-1441 (1980)

[9] Vakhania N. A better algorithm for sequencing with release and delivery times on identical processors. *Journal of Algorithms* 48, p.273-293 (2003)

[10] Vakhania N. "Single-Machine Scheduling with Release Times and Tails". *Annals of Operations Research*, 129, p.253-271 (2004)

[11] N. Vakhania, J.A. Hernandez, C. Zavala. A single-machine scheduling problem to minimize the maximum lateness is tightly related with a variation of bin packing problem with different bin capacities. Ciencia e Tecnica Vitivinicola Journal Vol. 30 (2015)

SESSION

LATE BREAKING AND POSITION PAPERS

Chair(s)

TBA

ISBN: 1-60132-434-0, CSREA Press ©

ISBN: 1-60132-434-0, CSREA Press ©

Algorithms, Integer Sequences, and Closed Formulas in the Enumeration of Integer Matrices

Shanzhen Gao, Keh-Hsun Chen
Department of Computer Science, College of Computing and Informatics
University of North Carolina at Charlotte, Charlotte, NC 28223, USA
Email: sgao3@uncc.edu, chen@uncc.edu

***Abstract*—We will discuss the following three problems: (1) the number of $m \times n$ matrices over $\{0,1\}$ with each row summing to s and each column summing to t; (2) the number of nonnegative integer matrices of size $m \times n$ with each row sum equal to s and each column sum equal to t; (3) the number of $(0,1)$ - matrices of size $n \times n$ such that each row has exactly s 1's and each column has exactly s 1's and with the restriction that no 1 stands on the main diagonal. We will present many conjectures and three algorithms. Integer sequences which arise from many areas are widely used in many disciplines. We can get many integer sequences based on our conjectures which could be verified by our computation.**

Keywords: Algorithm, integer matrix, closed formula, integer sequence

I. Introduction

Let m, n, s, t be positive integers such that $sm = tn$. Let $f(m,n,s,t)$ be the number of $m \times n$ Matrices over $\{0,1\}$ with each row summing to s and each column summing to t. Equivalently, $f(m,n,s,t)$ is the number of semiregular bipartite graphs with m vertices of degree s and n vertices of degree t. This problem has been the subject of considerable study, and it is unlikely that a simple formula exists. The asymptotic value of $f(m,n,s,t)$ has been much studied but the results are incomplete. Historically, the first significant result was that of Read, who obtained the asymptotic behavior for $s = t = 3$ [1]. McKay and Wang (2003) solved the sparse case $\lambda(1-\lambda) = o((mn)^{-1/2})$ using combinatorial methods [3] . Canfield and McKay used analytic methods to solve the problem for two additional ranges. In one range the matrix is relatively square and the density is not too close to 0 or 1. In the other range, the matrix is far from square and the density is arbitrary. Interestingly, the asymptotic value of $f(m,n,s,t)$ can be expressed by the same formula in all cases where it is known. Based on computation of the exact values for all m; n<30, they got the conjecture that the same formula holds whenever $m+n \to \infty$ regardless of the density (they defined the density $\lambda = s/m = t/m$).

We are concerned in this paper with the closed formulas of $f(m,n,s,t)$. The number in question can be related in various ways to the representation theory of the symmetric group or of the complex general linear group, but this does not make their computation any easier. The case $s = t = 2$ is solved by Anand, Dumir, and Gupta [4]. A formula for the case $s = t = 3$ appears in L. Comtet's Advanced Combinatorics [5], without proof.

Let $t(m,n,s,t)$ be the number of nonnegative integer matrices of size $m \times n$ with each row sum equal to s and each column sum equal to t ($sm = nt$). The enumeration of nonnegative integer matrices has been the subject of considerable study, The determination of $t(m,n,s,t)$ is an unsolved problem and it is unlikely that a simple formula exists except for very small s, t. Equivalently, $t(m,n,s,t)$ counts $2-$way contingency tables of order $m \times n$ such that the row marginal sums are all s and the column marginal sums are all t. Another equivalent description is that $t(m,n,s,t)$ is the number of semiregular labelled bipartite multigraphs with m vertices of degree s and n vertices of degree t. The matrices counted by $t(m,n,s,t)$ arise frequently in many areas of mathematics, for example enumeration of permutations with respect to descents and statistics. The last field in particular has an extensive literature in which such matrices are studied as contingency tables or frequency table.

An integer sequence is a sequence (i.e., an ordered list) of integers. An integer sequence may be specified explicitly by giving a formula for its nth term, or implicitly by giving a relationship between its terms. For example, the sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ (the Fibonacci sequence) is formed by starting with 0 and 1 and then adding any two consecutive terms to obtain the next one: an implicit description. The sequence 0, 3, 8, 15, $\ldots$ is formed according to the formula n^2-1 for the nth term: an explicit definition. Alternatively, an integer sequence may be defined by a property which members of the sequence possess and other integers do not possess [6], [7]. An integer sequence is a computable sequence, if there exists an algorithm which given n, calculates a_n, for all $n > 0$. An integer sequence is a definable sequence, if there exists some statement $P(x)$ which is true for that integer sequence x and false for all other integer sequences. The set of computable integer sequences and definable integer sequences are both countable, with the computable sequences a proper subset of the definable sequences (in other words, some sequences are definable but not computable). The set of all integer sequences is uncountable (with cardinality equal to that of the continuum); thus, almost all integer sequences are incomputable and cannot be defined.[6]

Why does one integer follow another? What is the pattern? What rule or formula dictates the position of each integer? Most people think deeply about sequences only when confronted by one on a test, but for mathematicians, computer scientists, and others, sequences are part and parcel

of their work. Today sequences are especially important in number theory, combinatorics, and discrete mathematics, but sequences have been known and wondered about even before the time of Pythagoras, who discovered an infinite sequence of integers such that $a^2 + b^2 = c^2$. In medieval times, bell ringers relied on sequences to cycle through all possible combinations of bells. But no one in the intervening millennia had thought to compile sequences into a collection that could be referenced by others. Neil Sloane started collecting integer sequences as a graduate student in 1965 to support his work in combinatorics. The database was at first stored on punch cards. He published selections from the database in book form twice: [12] containing 2372 sequences in lexicographic order and assigned numbers from 1 to 2372. [13] containing 5488 sequences. These books were well received and, especially after the second publication, mathematicians supplied Sloane with a steady flow of new sequences. The collection became unmanageable in book form, and when the database had reached $16,000$ entries Sloane decided to go online—first as an e-mail service (August 1994), and soon after as a web site (1996). As a spin-off from the database work, Sloane founded the Journal of Integer Sequences in 1998. The database continues to grow at a rate of some $10,000$ entries a year. Sloane has personally managed 'his' sequences for almost 40 years, but starting in 2002, a board of associate editors and volunteers has helped maintain the database. The On-Line Encyclopedia of Integer Sequences (OEIS), also cited simply as Sloane's, is an online database of integer sequences, created and maintained by N. J. A. Sloane, a researcher at AT&T Labs. OEIS records information on integer sequences of interest to both professional mathematicians and amateurs, and is widely cited. As of 25 September 2015 it contains over $260,000$ sequences, making it the largest database of its kind. And $15,000$ new entries are added each year. Each entry contains the leading terms of the sequence, keywords, mathematical motivations, literature links, and more, including the option to generate a graph or play a musical representation of the sequence. The database is searchable by keyword and by subsequence. [9], [10], [11], [12], [13]

Sequences can come from anywhere. Computational fields not surprisingly generate a lot of sequences. Computer science, to a large extent based on discrete math, also makes use of sequences (number of steps to sort n things). While it makes sense that sequences appear in mathematics, they are all around. The Fibonacci sequence in particular appears in nature: the growth of branches, pinecone rows, sandollar, and the number petals in many flowers all relate to the Fibonacci sequence. The sequence appears in art and literature too. Sloane originally started the sequence collection as an aid to research so that anyone coming upon a sequence in their calculations could immediately get additional terms and maybe a formula. This use of the OEIS is more important than ever today, since many computer-related tasks can be stated in terms of a sequence: minimizing the number of steps needed to count a set of items, ranking a list of unsorted numbers from lowest to highest, even characterizing the behavior of a program or algorithm. As more applications today depend on ideas and concepts taken from pure mathematics—cryptography, the use of graphs to study social networks, the ranking of search engine listings—sequences increasingly play a more direct role in solving real-world problems. [14]

Sequence data is pervasive in our lives, and understanding sequence data is of grand importance. Much research has been conducted on sequence data mining in the last dozen years. Hundreds if not thousands of research papers have been published in forums of various disciplines, such as data mining, database systems, information retrieval, biology and bioinformatics, industrial engineering, etc. The area of sequence data mining has developed rapidly, producing a diversified array of concepts, techniques and algorithmic tools. [15]

There are many research topics on integer sequence. For example: (1) How to find a good formula for a sequence with a bad formula or no formula at all? Sometimes it is not very hard to find the first several terms of a sequence by hand computation. It is might be very tough to find a formula. (2) How to find a good algorithm to compute more times for a sequence if you could not get a formula? People have been working on some sequences for more than one hundred years. However, they still could not get the first one hundred terms, or even not the first thirty terms. (3) The applications and data structure of some sequences. (4) Find some new sequences.

You can obtain many integer sequence from this paper.

II. Conjectures on Zero-One Matrices

"Let $f(n)$ be the number of $n \times n$ matrices M of zeros and ones such that every row and column of M has exactly three ones, $f(0) = 1$, $f(1) = f(2) = 0$, $f(3) = 1$. The most explicit formula known at present for $f(n)$ is

$$f(n) = 6^{-n} \sum \frac{(-1)^{\beta} n!^2 (\beta + 3\gamma)! 2^{\alpha} 3^{\beta}}{\alpha! \beta! \gamma!^2 6^{\gamma}} \qquad \text{((ii))}$$

where the sum is over all $(n + 2)(n + 1)/2$ solutions to $\alpha+\beta+\gamma = n$ in nonnegative integers. This formula gives very little insight into the behavior of $f(n)$, but it does allow one to compute $f(n)$ faster than if only the combinatorial definition of $f(n)$ were used. Hence with some reluctance we accept (ii) as a "determination" of $f(n)$. Of course if someone were later to prove $f(n) = (n - 1)(n - 2)/2$ (rather unlikely), then our enthusiasm for (ii) would be considerably diminished." [16]

The enumeration of Integer-matrices has been the subject of considerable study. It has been the subject of considerable study, and it is unlikely that a simple formula exists. The number in question can be related in various ways to the representation theory of the symmetric group or of the complex general linear group, but this does not make their computation any easier.

Let $f(m, n, s, t)$ be the number of $(0, 1)$ - matrices of size $m \times n$ such that each row has exactly s ones and each column has exactly t ones ($sm = nt$). The determination of $f(m, n, s, t)$ is an unsolved problem, except for very small s, t.

In some row, let $x_{i_1}x_{i_2}\cdots x_{i_k}$denote the i_1-th column, the i_2-th column, $\cdots$, the i_k-th column entries are 1 in some row and other entries are all 0,where $i_1, i_2, \cdots, i_k \in \{1, 2, \cdots, n\}$.

Example: Let $m = n = 4, s = t = 3$, then $x_1x_2x_3|x_1x_2x_4|x_1x_3x_4|x_2x_3x_4$ denotes the matrix as follows:

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Obviously, $f(m, n, s, t)$ equals the coefficient of $x_1^t x_2^t \cdots x_n^t$ in the symmetric polynomial

$$\left(\sum_{i_1<i_2<\cdots<i_s} x_{i_1}x_{i_2}\cdots x_{i_s} \right)^m$$

where $i_1, i_2, \cdots, i_s \in \{1, 2, \cdots, n\}$, and the sum is over all the possible of $s-$ combinations from $\{1, 2, \cdots, n\}$ with $i_1 < i_2 < \cdots < i_s$. It is easy to get,

$$f(m, n, s, t) = f(n, m, t, s), \quad (sm = tn)$$
$$f(m, n, s, t) = f(n, m, n - s, m - t), \quad (sm = tn)$$
$$f(m, n, 1, t) = \frac{m!}{(t!)^n} \quad (m = tn)$$
$$f(m, n, s, 1) = \frac{n!}{(s!)^m} \quad (sm = n)$$

Conjecture 1. $f(n, n, 2, 2) = \frac{n!}{2^n} \sum_{r_0=0}^{n} \binom{n}{r_0} \frac{(-1)^{n-r_0}(2r_0)!}{2^{r_0} r_0!}$

Conjecture 2. $f(m, n, 3, 2) = \frac{n!}{2^n} \sum_{r_0=0}^{m} \binom{m}{r_0} \frac{(-1)^{m-r_0}(2r_0+m)!}{(n-m+r_0)!6^{r_0}}$

Conjecture 3. $f(m, n, 4, 2) = \frac{n!}{2^n} \sum_{r_0=0}^{m} \sum_{r_1=0}^{m-r_0}$

$$\frac{m!}{r_0!r_1!(m-r_0-r_1)!} \frac{(-1)^{2(m-r_0)-r_1}}{(n-2m+2r_0+r_1)!} \frac{(4r_0+2r_1)!}{24^{r_0} 2^{(m-r_0)}}$$

Conjecture 4. $f(m, n, 5, 2) = \frac{n!}{2^n} \sum_{r_0=0}^{m} \sum_{r_1=0}^{m-r_0}$

$$\frac{m!}{r_0!r_1!(m-r_0-r_1)!} \frac{(-1)^{r_1+2(m-r_0-r_1)}(4r_0+2r_1+m)!}{(n+r_1-2m+2r_0)!120^{r_0} 6^{r_1} 2^{(m-r_0-r_1)}}$$

Conjecture 5. $f(m, n, 6, 2) = \frac{n!}{2^n} \sum_{r_0=0}^{m} \sum_{r_1=0}^{m-r_0} \sum_{r_2=0}^{m-r_0-r_1}$

$$\frac{m!}{r_0!r_1!r_2!(m-r_0-r_1-r_2)!} \frac{(-1)^{3m-3r_0-2r_1-r_2}}{(n+2r_1+r_2-3m+3r_0)!}$$

III. Algorithm One

The algorithm used to verify the equations presented counts all the possible matrices, but does not construct them.It is best described with an example. Suppose we wanted to compute $f(12, 9, 3, 4)$. We first create a state vector of length 9, filled with $4s$:

#(4 4 4 4 4 4 4 4 4)

Each state vector can be thought of as a container to inform us how many ones need to go into each column. The '#' symbol reminds us that we must count the number of possibilities that we can put the indicated number of ones into each column. We assign where the ones will go in the first row. Clearly, 3 ones need to go in the first row somewhere, and there are (9 take 3) = 84 possibilities for this placement. Hence, we simply assign them to go in the leftmost positions. Then, our state vector drops to #(3 3 3 4 4 4 4 4 4) noting that however many possibilities there are to fill in the remaining 11 rows, we multiply this by (9 take 3). Thus, we have

#(4 4 4 4 4 4 4 4 4) = 84 * #(3 3 3 4 4 4 4 4 4).

Eventually, we would like to drop the state vector to #(0 0 0 0 0 0 0 0 0) after (exactly) all 12 rows have been assigned, reflecting a properly filled-in matrix. Now, for the second row, there are again 3 ones to place. Some of them can go in columns where ones are above, and some of them can go in columns where ones haven't been placed yet. The possibilities are as follows: 3/0, 2/1, 1/2, and 0/3, where x/y denotes putting x ones in the "left part" (where ones have been placed before) and y ones in the "right part" (where ones haven't been placed yet). We calculate each in turn.

For 3/0, there is only (3 take 3) = 1 way to place all 3 ones in the left part, and (6 take 0) = 1 way to place 0 ones in the right part. Hence, in this case we drop our state vector to #(2 2 2 4 4 4 4 4 4), since 2 ones will need to be placed in the leftmost three columns during subsequent row assignments, and we note that we'll multiply the ways to fill in a matrix this way by (3 take 3) * (6 take 0) = 1 * 1.

We also consider 2/1. There are (3 take 2) = 3 ways to place 2 ones in the left part, and (6 take 1) = 1 way to place a one in the right part. Now, as before, we will elect to place these ones in the leftmost area of each part.

Since 2 ones will be placed in the leftmost area of the left part, and 1 one will be placed in the leftmost area of the right part, our state vector in this case drops to

#(2 2 3 3 4 4 4 4 4).

We also consider 1/2. There are (3 take 1) = 3 ways to place 1 one in the left part, and (6 take 2) = 15 ways to place a one in the right part. Hence, our state vector in this case drops to

#(2 3 3 3 3 4 4 4 4).

We also consider 0/3. There is (3 take 0) = 1 way to place 0 ones in the left part, and (6 take 3) = 20 ways to place a one in the right part. Hence, our state vector in this case drops to

#(3 3 3 3 3 3 4 4 4).

Thus, in total, we have

#(3 3 3 4 4 4 4 4 4) = (1 * 1 * #(2 2 2 4 4 4 4 4 4)) + (3 * 1 * #(2 2 3 3 4 4 4 4 4)) + (3 * 15 * #(2 3 3 3 3 4 4 4 4)) + (1 * 20 * #(3 3 3 3 3 3 4 4 4)).

We would then proceed to work on each sub-state vector in turn. One final example: to compute #(2 2 3 3 4 4 4 4 4), we see that we have three parts: the left part (consisting of two columns) where 2 ones have already been placed, the middle part (consisting of two columns) where 1 one has already been placed, and the right part (consisting of five columns) where 0 ones have been placed. To assign our third row, we (again) need to place 3 ones, so we consider all the possibilities.

We see that 3/0/0 is not possible since there are only two columns in the left part. Similarly, 0/3/0 is not possible. We

then compute the remaining possibilities: 2/1/0, 2/0/1, 1/2/0, 1/1/1, 1/0/2, and 0/0/3, and continue on.

After 11 of the 12 row assignments, we will either get state vectors like #(0 0 0 0 0 0 1 1 1) in which case we can terminate with a 1, or vectors like

#(0 0 0 0 0 0 0 0 1) or #(0 0 0 0 0 0 0 1 2)

in which case we can terminate with a 0, since it is impossible to fill in 3 ones in the last row in the prescribed manners.

This is the backbone of the algorithm. We remark that it is very possible to take different paths to get the same state vector later on, so we only compute its count once, storing it for later use if it shows up again. In its current implementation, the calculation engine is completely separated from the storage object, so improvements to reading/writing from/to the storage object can be explored independently. We've found that in Scheme, a tree with ten branches at each node seems to optimize reading and writing, once the state vector is hashed (uniquely) into a whole number. Other node widths are certainly possible.

IV. Algorithm Two

Conjecture 6. *The number of* $(0,1)$ *- matrices of size* $n \times n$ *such that each row has exactly* s 1*'s and each column has exactly* s 1*'s and with the restriction that no* 1 *stands on the main diagonal is* $\sum_{k=0}^{n}\sum_{s=0}^{k}\sum_{j=0}^{n-k}\frac{(-1)^{k+j-s}n!(n-k)!(2n-k-2j-s)!}{s!(k-s)!((n-k-j)!)^2 j!2^{2n-2k-j}}$.

Enclosed is a walkthrough for the Lefty algorithm which computes the number of nxn 0-1 matrices with t ones in each row and column, but none on the main diagonal.

The algorithm used to verify the equations presented counts all the possible matrices, but does not construct them.

It is called "Lefty", it is reasonably simple, and is best described with an example.

Suppose we wanted to compute the number of 6x6 0-1 matrices with 2 ones in each row and column, but no ones on the main diagonal. We first create a state vector of length 6, filled with 2s:

#(2 2 2 2 2 2)

This state vector symbolizes the number of ones we must yet place in each column. We accompany it with an integer which we call the "puck", which is initialized to 1. This puck will increase by one each time we perform a ones placement in a row of the matrix (a "round"), and we will think of the puck as "covering up" the column that we won't be able to place ones in for that round.

Since we are starting with the first row (and hence the first round), we place two ones in any column, but since the puck is 1, we cannot place ones in the first column. This corresponds to the forced zero that we must place in the first column, since the 1,1 entry is part of the matrix's main diagonal.

The algorithm will iterate over all possible choices, but to show each round, we shall make a choice, say the 2nd and 6th columns. We then drop the state vector by subtracting 1 from the 2nd and 6th values, and advance the puck:

#(2 1 2 2 2 1); 2

For the second round, the puck is 2, so we cannot place a one in that column. We choose to place ones in the 4th and 6th columns instead and advance the puck:

#(2 1 2 1 2 0); 3

Now at this point, we can place two ones anywhere but the 3rd and 6th columns. At this stage the algorithm treats the possibilities differently: We can place some ones before the puck (in the column indexes less than the puck value), and/or some ones after the puck (in the column indexes greater than the puck value). Before the puck, we can place a one where there is a 1, or where there is a 2; after the puck, we can place a one in the 4th or 5th columns. Suppose we place ones in the 4th and 5th columns. We drop the state vector and advance the puck once more:

#(2 1 2 0 1 0); 4

For the 4th round, we once again notice we can place some ones before the puck, and/or some ones after.

Before the puck, we can place:

(a) two ones in columns of value 2 (1 choice)

(b) one one in the column of value 2 (2 choices)

(c) one one in the column of value 1 (1 choice)

(d) one one in a column of value 2 and one one in a column of value 1 (2 choices).

After we choose one of the options (a)-(d), we must multiply the listed number of choices by one for each way to place any remaining ones to the right of the puck.

So, for option (a), there is only one way to place the ones.

For option (b), there are two possible ways for each possible placement of the remaining one to the right of the puck. Since there is only one nonzero value remaining to the right of the puck, there are two ways total.

For option (c), there is one possible way for each possible placement of the remaining one to the right of the puck. Again, since there is only one nonzero value remaining, there is one way total.

For option (d), there are two possible ways to place the ones.

We choose option (a). We drop the state vector and advance the puck:

#(1 1 1 0 1 0); 5

Since the puck is "covering" the 1 in the 5th column, we can only place ones before the puck. There are (3 take 2) ways to place two ones in the three columns of value 1, so we multiply 3 by the number of ways to get remaining possibilities. After choosing the 1st and 3rd columns (though it doesn't matter since we're left of the puck; any two of the three will do), we drop the state vector and advance the puck one final time:

#(0 1 0 0 1 0); 6

There is only one way to place the ones in this situation, so we terminate with a count of 1. But we must take into account all the multiplications along the way: 1*1*1*1*3*1 = 3. So, this string of rounds counts the following three matrices:

0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1

0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1
0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0
1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0
1 1 0 0 0 0 1 0 1 0 0 0 0 1 1 0 0 0 <– only variation
0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 1 0

Another way of thinking of the varying row is to start with the first matrix, focus on the lower-left 2x3 submatrix, and note how many ways there were to permute the columns of that submatrix. Since there are only 3 such ways, we get 3 matrices.

We cannot optimize by permuting submatrices that contain an entry of the main diagonal, since that is a 'fixed' position that must contain a zero.

We note that, in the actual implementation, after each round, the state vector values to the left of the puck are sorted (but the values to the right of the puck maintain their exact positions) to make counting possibilities easier. Hence, we would have in the third and fourth rounds, respectively,

#(1 2 2 1 2 0); 3

#(1 2 2 0 1 0); 4

In a larger example (13x13 matrix with 3 ones in each row/column), we might come across the following state:

#(0 1 1 1 2 2 3 3 0 1 0 0 1); 9

To place three ones in this case, the algorithm would branch depending on how many ones it wishes to place to the right of the puck, make that choice, and then multiply by the possibilities for placing the remaining ones to the left of the puck. Hence,

Case 1: Right of the puck gets 3 ones.

Not possible since there are only two nonzero columns there.

Case 2: Right of the puck gets 2 ones.

Only one way to do this, but there are three different ways to place the third one to the left of the puck:

(a) under a column with a 1 value (3 ways), with resultant state #(0 0 1 1 2 2 3 3 0 0 0 0 0); 10

(b) under a column with a 2 value (2 ways), with resultant state #(0 1 1 1 1 2 3 3 0 0 0 0 0); 10

(c) under a column with a 3 value (2 ways), with resultant state #(0 1 1 1 2 2 2 3 0 0 0 0 0); 10.

Case 3: Right of the puck gets 1 one.

There are two ways to do this, so we have to branch depending on if it's going in the 10th column or 13th column.

Subcase 1: 10th column.

To place the other two ones to the left of the puck, we have choices:

(d) both ones under a 1-value ((3 take 2) ways),
with resultant state #(0 0 0 1 2 2 3 3 0 0 0 0 1); 10

(e) one one under 1-value, one under 2-value ((3 take 1)*(2 take 1) ways),
with resultant state #(0 0 1 1 1 2 3 3 0 0 0 0 1); 10

(f) one one under 1-value, one under 3-value ((3 take 1)*(2 take 1) ways),
with resultant state #(0 0 1 1 2 2 2 3 0 0 0 0 1); 10

(g) both ones under 2-value ((2 take 2) ways),
with resultant state #(0 1 1 1 1 1 3 3 0 0 0 0 1); 10

(h) one one under 2-value, one under 3-value ((2 take 1)*(2 take 1) ways),
with resultant state #(0 1 1 1 1 2 2 3 0 0 0 0 1); 10

(i) both ones under 3-value ((2 take 2) ways),
with resultant state #(0 1 1 1 2 2 2 2 0 0 0 0 1); 10.

Subcase 2: 13th column.

The options (j)-(o) are the same as (d)-(i) in the above subcase, but the resultant states have #(... 0 1 0 0 0) at the end instead.

Case 4: Right of the puck gets 0 ones.

So all three ones go to the left of the puck. We have choices:

(p) all ones under 1-value ((3 take 3) ways),
with resultant state #(0 0 0 0 2 2 3 3 0 1 0 0 1); 10

(q) two ones under 1-value, one under 2-value ((3 take 2)*(2 take 1) ways),
with resultant state #(0 0 0 1 1 2 3 3 0 1 0 0 1); 10

(r) two ones under 1-value, one under 3-value ((3 take 2)*(2 take 1) ways),
with resultant state #(0 0 0 1 2 2 2 3 0 1 0 0 1); 10

(s) two ones under 2-value, one under 3-value ((2 take 2)*(2 take 1) ways),
with resultant state #(0 1 1 1 1 1 2 3 0 1 0 0 1); 10

(t) one one under 2-value, two under 3-value ((2 take 1)*(2 take 2) ways),
with resultant state #(0 1 1 1 1 2 2 2 0 1 0 0 1); 10

In all options (a)-(t), the state would be resorted: since the puck moved from the 9th column to the 10th column, it will reveal a 0 in the 9th column, which will then get moved to the front of the state vector.

In general, Lefty will iterate over all possible choices (optimizing for permutations below the main diagonal by multiplying by the indicated cofactors), add up the values, and produce the result. To provide a further speedup, a storage object is used to store each state vector for which a count has been acquired, so that if that state vector is seen again, the count can be produced from memory instead of recalculated. This speedup is necessary, and without it the algorithm will take too long.

V. Algorithm Three

Let $t(m, n, s, t)$ be the number of nonnegative integer matrices of size $m \times n$ with each row sum equal to s and each column sum equal to t ($sm = nt$).

Conjecture 7. $t(n, n, 2, 2) = 4^{-n} \sum_{i=0}^{n} \frac{2^i (n!)^2 (2n-2i)!}{i!((n-i)!)^2}$

Conjecture 8. $t(n, m, 3, 2) = 2^{-m} \sum_{i=0}^{n} \frac{m!n!(2m-2i)!}{i!(m-i)!(n-i)!6^{n-i}}$

Conjecture 9. $t(m, n, 4, 2) = 24^{-m} \sum_{\alpha+\beta+\gamma=m} \frac{3^\alpha 6^\gamma m!n!(4\beta+2\gamma)!}{\alpha!\beta!\gamma!(2\beta+\gamma)!2^{2\beta+\gamma}}$ *where the sum is over all* $\binom{m+2}{2}$ *solutions of* $\alpha + \beta + \gamma = m$ *in nonnegative integers.*

Conjecture 10. $t(m, n, 5, 2) = 120^{-m} \sum_{\alpha+\beta+\gamma=m} \frac{10^\beta 15^\gamma m!n!(5\alpha+3\beta+\gamma)!}{\alpha!\beta!\gamma!(n-\beta-2\gamma)!2^{n-\beta-2\gamma}}$,

Algorithm Description For t(m, n, s, t)

The algorithm used to verify the equations presented counts all possible matrices, but does not construct them.

It is a bit involved, so it is best described with an example.

Suppose we wanted to compute the number of 4x6 matrices over nonnegative integers with row sum 12 and column 8. We first create a list of all nonincreasing partitions of 12: 12, 11 1, 10 2, 10 1 1, 9 3, etc., and store this in memory. We make sure that each partition stored is not of length greater than the number of columns of the matrix. We then create a state vector of length 6 filled with 8s:

#(8 8 8 8 8 8)

This state vector symbolizes the sum of integers we must place in each column, and each time the state changes, it is sorted in nondecreasing order.

An additional vector, called the cap vector, is created when we deal with a new state. It records the length of the contiguous blocks of numbers found in the state. Here, it is

#(6).

Next, we iterate over each of the (valid) partitions of 12 that we could possibly use for the choice of the first row of the matrix. Here, our first partition is 8 4. We then create a partition block (pb) vector, which is exactly a "cap vector" of the partition, instead of the state. Here, it is

#(1 1).

Finally, we create all the assignment vectors that are valid for this partition and this cap vector. An assignment vector dictates where the indicated element of the partition will be placed in the row. Assignment vectors always have the same length as the partition we are planning to use. The entries of the assignment vector refer to the (zero-based) indices of the cap vector. Since the cap vector in this case only has one index (namely, 0) and both 8 and 4 can be elements in the matrix row, we assign 8 and 4 to the 0th index:

#(0 0)

In other words, both the 8 and the 4 will appear in block 0 of the state. Now, there are (6 take 1)*(5 take 1) ways of placing the 8 and 4, so we note that when we drop the state vector. We pretend that the first row of the matrix will be (8 4 0 0 0 0), and so, dropping the state vector, the remaining three rows must sum to

#(0 4 8 8 8 8)

and we record that the number of ways of obtaining a matrix of state #(8 8 8 8 8 8) is 30 times the number of ways we can obtain a matrix of state #(0 4 8 8 8 8).

Of course, we must add to our count the other ways to assign the 8 and 4. Since there are no other ways, no more assignment vectors can be constructed. We then add to our count the ways in which we can use the partition 8 3 1 (with all applicable assignment vectors), and then 8 2 2 (with all applicable assignment vectors), and so forth.

To get a better feel for how the assignment vectors are created, let's say that, in the middle of our counting, we achieve the state

#(1 1 4 6 6 6)

with two rows left to fill. Our cap vector is then

#(2 1 3)

and suppose we are considering the partition 4 4 3 1. Its pb is #(2 1 1). Since the cap vector has length 3, the indices for it are 0, 1, and 2, so the entries of each assignment vector can be comprised only of 0, 1, and/or 2.

To create the first assignment vector, we note that the first element of the partition, 4, cannot be placed in block 0 of the state (the block of two 1s), since 4 > 1. A single 4 can be placed in block 1 of the state (the block consisting of the single 4), so the first 4 in the partition can be assigned to block 1:

#(1 ? ? ?)

But block 1 is only length 1 (as noted by the cap vector's entry of 1 at index 1), so no more 4s can go in that block. The second 4 in the partition can also be placed in block 2 of the state (the block of three 6s), since 4 <= 6. Thus, our assignment vector changes to

#(1 2 ? ?).

Next in the partition, we have a 3, which is also greater than 1, so it too cannot go into block 0. Block 1 has already been taken by the 4. Hence the only remaining place for it is in block 2:

#(1 2 2 ?)

Finally, the last element of the partition is a 1, which can go anywhere in the state. We begin by assigning it to block 0, giving the resulting assignment vector as

#(1 2 2 0).

How many ways could these assignments be carried out? The first 4 has only one way. The second 4 and the 3 are both in block 2, but they are different numbers, so they can be inserted in (3 take 1)*(2 take 1) ways. Finally, the 1 has (2 take 1) ways to be inserted into block 0. Hence we multiply to get 12 ways for this assignment vector, and dropping the state, we get #(0 1 0 2 3 6). Sorting it, it becomes #(0 0 1 2 3 6), which we will process after we deal with the remaining assignment vectors possible for 4 4 3 1.

To get the next assignment vector, we note that we can keep everything the same, but the 1 in the partition can be put in block 2. This gives

#(1 2 2 2)

and to compute the number of ways, we have 1*(3 take 1)*(2 take 1)*(1 take 1) = 6.

To get the next assignment vector, we note we've exhausted all possibilities for #(1 2 ? ?), so we then find the 'next' way to assign the two 4s in the partition. The only remaining option is to put them both in block 2, so we start with

#(2 2 ? ?).

Now, the 3 can go in block 1 and the 1 can go in block 0, giving

#(2 2 1 0)

and total number of ways (3 take 2)*(1 take 1)*(2 take 1) = 6.

Now, we think of a "block" of the assignment vector as the entries that correspond to an equal number in the partition; here, the first two entries correspond to the partition entry 4, so they form a block. The pb tells us the length of each block of the assignment vector. For example, recall that here, pb

is #(2 1 1), so each assignment vector corresponding to this partition has three blocks, the first of which has length two, and the remaining two have length one. We construct assignment vectors that are nondecreasing in each block, though we can have a decrease when we move to a new block from an old one. The remaining three assignment vectors and the number of ways to make the assignment are then

#(2 2 1 2) with ways (3 take 2)*(1 take 1)*(1 take 1) = 6

#(2 2 2 0) with ways (3 take 2)*(1 take 1)*(2 take 1) = 12

#(2 2 2 1) with ways (3 take 2)*(1 take 1)*(1 take 1) = 6.

Let's consider a larger example. Suppose the state was

#(0 1 1 1 1 2 2 2 3 3 3 3 4 5 5)

with row sum 18. This state will produce a cap vector of #(4 3 4 1 2) (since zeroes in the state are ignored). Let's suppose we were considering the partition

3 3 3 2 2 2 1 1 1,

which gives a pb of #(3 3 3). There are 433 total assignment vectors for this partition. The first one we could construct is

#(2 2 2 1 1 1 0 0 0) with ways (4 take 3)*(3 take 3)*(4 take 3) = 16,

an intermediate one we could construct is

#(2 3 4 1 1 2 0 1 2) with ways (4 take 1)*(1 take 1)*(2 take 1) for placing the three 3s

(3 take 2)(3 take 1) for placing the three 2s

(4 take 1)(1 take 1)*(2 take 1) for placing the three 1s (total 576),

and the last one we could construct is

#(3 4 4 2 2 2 1 1 2) with ways (1 take 1)*(2 take 2) for placing the three 3s

*(4 take 3) for placing the three 2s

(3 take 2)(1 take 1) for placing the three 1s (total 12).

Notice that each block of each assignment vector has its entries in nondecreasing order, but often there is a decrease when we move from block to block. Since the state vectors are nondecreasing, this is to be expected.

In general, for each state vector that is achieved, this algorithm will iterate over all assignment vectors for each valid partition, multiplying cofactors and adding the results. When fitting the last row, though, the calculation is surprisingly easy: continuing the example we had above, if we examine the state #(0 0 1 2 3 6), we see that there is only one possible partition of 12 that fits it (namely 6 3 2 1) and there is only one way to fit it in. Hence, there is only one way to achieve this state. The situation is the same for every state with one row left to be filled.

For further speedup, a fast storage object must be used, so that if a given state is seen again, we can recall from memory how many partially-filled matrices can produce it. This speedup is necessary, for without it, the algorithm will take too long. Other approaches are certainly possible.

References

[1] R.C. Read, Some enumeration problems in graph theory, Doctoral Thesis, University of London, (1958).

[2] B. D. McKay and X. Wang, Asymptotic enumeration of 0-1 matrices with equal row sums and equal column sums, Linear Alg. Appl., 373 (2003) 273-288.

[3] E. Rodney Canfield and Brendan D. McKay, Asymptotic enumeration of dense $0-1$ matrices with equal row sums and equal column sums. Electron. J. Combin. 12 (2005), Research Paper 29, 31 pp.

[4] Anand, Dumir, and Gupta in Duke Math J., 33 (1966) 757-769.

[5] L. Comtet, Advanced Combinatorics (page 236),Kluwer Academic Publishers,1974(page 236).

[6] http://en.wikipedia.org/wiki/Integer_sequence

[7] The On-Line Encyclopedia of Integer Sequences, http://oeis.org/

[8] http://en.wikipedia.org/wiki/On-Line_Encyclopedia_of_Integer_Sequences

[9] N. J. A. Sloane, The On-Line Encyclopedia of Integer Sequences, Proceedings Fifteenth Conference on Fibonacci Numbers, 2012, to appear.

[10] N. J. A. Sloane, My Favorite Integer Sequences, arXiv:math/0207175v1 [math.CO].

[11] D. Applegate, O. E. Pol, N. J. A. Sloane, The Toothpick Sequence and Other Sequences from Cellular Automata, Congress. Numer., 206 (2010), 157-191.

[12] N. J. A. Sloane, A Handbook of Integer Sequences, Academic Press, NY, 1973.

[13] N. J. A. Sloane and S. Plouffe, The Encyclopedia of Integer Sequences, Academic Press, 1995.

[14] AT&T Labs Research, The Achievement of The Online Encylopedia of Integer Sequences, March 6, 2012.

[15] G. Dong, J.Pei, Sequence Data Mining, Series: Advances in Database Systems, Springer, 2007.

[16] R. P. Stanely, Enumerative Combinatorics, Volumes 1, Cambridge University Press (first edition 1986, second edition 2011).

Concerns related to Sustainable Transportation Systems for Urban Freight

Regis Z. Stinson[1], Peter Keiller[2]
[1]*Department of Civil Engineering,* [2]*Department of Computer Science*
Howard University
Washington, DC 20059

FCS'16- POSITION PAPER

Abstract

The transportation sector has proven to be a particularly difficult area to overcome for the advancement of sustainable development. The economic and demographic development of urban agglomerations heavily depends on a reliable supply of goods and material thus making urban freight transportation a major concern for the sustainable development of cities. A number of strategies ranging from implemented policies to localized technology and modal shifts options in different countries have been researched to reduce the urban freight transport impacts. Modern technology has also made it easier for everyone to accurately measure and control the impacts of urban freight transportation. This paper addresses some of the concerns related to sustainable transport systems for urban freight.

Keywords: Sustainable transport system, Freight transportation

1 Introduction

For the past few years, there has been a worldwide concern to set up sustainable development strategies in order to achieve a continuous improvement in quality of life. Since the 1987 Brundtland Commission report (Oxford University Press 1987) brought global attention to the concept of sustainable development, scholars and policy professionals have worked to apply its principles in the urban and metropolitan context. However, not every aspect of a city's function has been studied in depth. According to Hicks [1], "any urban area depends for its existence on a massive flow of commodities into, out of, and within its boundaries. Yet the transport of goods remains a forgotten aspect of urban transportation study". The transportation sector has proven to be particularly difficult for the advancement of sustainable development.

The UN-HABITAT report [2], states that "transportation alone is responsible for approximately 23 percent of total energy related greenhouse (GHG) emissions and 13 percent of global GHG emissions". Furthermore, transportation has a direct relationship with the growth of urban areas, which continues to grow at a rapid rate. Even the most efficient cities' transportation systems are facing escalating motorization and mobility demands. Travels throughout all regions of the world have increased at the rate equal to or greater than most countries economic growth and development.

The economic and demographic development of urban agglomerations depends heavily on a reliable supply of goods and materials. Nevertheless, freight transportation in urban centers contributes considerably to the GHG emissions, noise and traffic congestion. In fact, in most cities around the world, these negative effects have reached levels at which the quality of urban life has been significantly affected.

2 Background Review

Freight transportation is primarily a business to business industry, and for firms established within city limits it forms a vital link with suppliers and customers. Figure 1 presents the role of freight transportation in an effective production and distribution system [14].

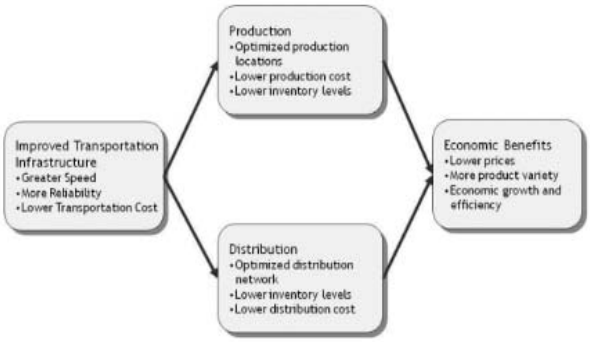


Fig.1 Efficient Freight Transportation System

Freight is carried by vehicles that move on the same streets and arteries used by private and public vehicles. Urban freight traffic also contributes to the belief that cities are not safe, which causes numerous citizens to move out of the city limits. The already significant volume of freight vehicles moving within city limits is

growing, and is expected to continue to grow at a rate faster than expected. Major contributing factors to this phenomenon are the current production and distribution practices based on low inventories and timely deliveries as well as the explosive growth of business-to-customer electronic business activities that generates significant volumes of personal deliveries. [3]

Urban freight transportation is considered a grave problem for the sustainable development of cities. Apart from a comprehensive solution to the freight transportation problem, a specific study on urban freight transportation is needed.

Different types of freight flow in, out, and throughout the urban areas including consumer goods, waste products, construction materials, mail, etc. [4]. Changes in modern society influence the continuing growth of urban freight transportation such as movement towards a post-industrial society, urbanization, aging, individualization and also the increasing awareness of sustainable development [5].

Concurrently, businesses are also moving towards the just-in-time (JIT) operations which require timely delivery with small shipments and less storage space. The rapid growth of online shopping has also generated significant increase in the volumes of home deliveries as well as the level of freight traffic. According to Dablanc [4], "urban freight represents ten to fifteen percent of vehicle equivalent miles travelled in city streets and two to five percent of the employed urban workforce. Also, three to five percent of urban land is devoted to freight transportation and logistics".

Last kilometer freight distribution emphasizes the last link of the supply chain that delivers goods to retailers in urban areas. Traditionally, the retailer as the last party in the supply chain is the one who finally sells the product to the consumer. The key characteristics of last kilometer freight distribution are: (1) a wide variety of goods being delivered over relatively short distances in a congested urban setting and small shipment size with high frequency of delivery and (2) Freight carriers will serve a number of locations in one delivery round with less capacity utilization in comparison with long-distance freight transport. Allen [8] states that the degree of centralization in the supply of goods to retail outlets also influences the level of freight transport.

The growth in freight is a major contributor to congestion in urban areas and on intercity routes producing congestion that affects the timeliness and reliability of freight transportation. A long-distance freight movement also plays a significant contributor to local congestion which typically delays the freight from getting to its destination and as such affects the local economy.

The current growing urban freight demand increases recurring congestion at "freight bottlenecks"- places where freight and passenger service conflict with one another, and where there is not enough room for local pickup and delivery.

Congestion could also be caused by restrictions on urban freight movement, such as the lack of space for trucks to load and unload as well as limitations on delivery and pick-up times. One estimate of urban congestion attributes 947,000 hours of vehicle delay to delivery trucks parked at curbside in dense urban areas where office buildings and stores lack off-street loading facilities. [6]

In addition, the environmental impacts caused by urban freight are imposing a large toll on urban centers. Urban freight pollutes more than long distance freight transportation, due to the average age of the vehicles and the high number of short trips and stops. Freight transportation generates between 20% and 60% (according to the pollutants considered) of local transport-based pollution.

3 Addressing the Concerns

Usually, freight transportation is considered to be a private industry on both the supplier and user sides and it is driven by economic parameters. A large majority of cities have not yet found adequate solutions to help optimize the movement of goods in urban areas. It is proposed that sustainable urban freight transportation systems could play a major role in this optimization process. According to Behrends [9], a sustainable urban freight transportation system should fulfill four main objectives. First, it should ensure the accessibility offered by the transportation system of a city to all categories of freight transportation. Second, it should have reduced levels of air pollution, greenhouse gas emissions, waste and noise and no negative impacts on health of the citizens or nature. Third, it has to improve the energy efficiency and cost-effectiveness of the transportation of goods, taking into account the external costs. Lastly, it has to contribute to the enhancement of the attractiveness and quality of the urban environment, by avoiding accidents, minimizing the use of land and without compromising the mobility of citizens.

To achieve these objectives related to sustainable urban freight transportation systems, different strategies are at present being implemented including "demand management, operations management, pricing policies, vehicle technology improvements, clean fuels, and integrated land use and transportation planning" [11]. Currently, there are several cities around the world that have implemented programs and regulations to pursue

sustainability in their urban freight transportation system [12].
Most of these plans, such as reducing GHG emissions, are not directly targeted in most government policies or regulations. There is therefore still a chance to make a difference by implementing these goals and generating more benefits for the community.

Land use patterns, urban design, or the built form of a city, can have an impact on reducing the GHG emissions and traffic congestion from urban goods movement by supporting the efficient delivery of goods. It is clear that innovation and new technologies are critical to achieve a more sustainable way to move goods among cities. A survey by the Supply Chain and Logistics Association of Canada and Industry Canada [10] after examining green supply chain management strategies for logistics and transportation services from a Canadian perspective, concluded that for green technology initiatives to be successful, environmental benefits and a positive financial result for the service provider must both be achieved at the same time. While the study was not strictly focused on urban goods movement, it still provides some background on the drivers behind technology adoption and the types of tools that are being used to achieve a sustainable urban freight system.
There are numerous tests projects on the applicability of alternative fuels namely hybrid-electric, complete electric, compressed natural gas (CNG), biodiesel and ethanol. In most cases, fleet based transportation companies are usually in the best position to test these alternative fuels by using some of their vehicles as test cases and switching to others while still keeping the vehicles operating on comparable routes. Fleets with a large variety of vehicle types on set routes are good candidates to be in the test and in the end benefit from new fuels. A large percentage of their operating costs are fuel, so reducing these costs is critical to their long-term success. Both UPS and FedEx, two of the largest courier companies, have several on-going tests in various markets. Also, they are constantly testing routing technology to improve their operations.
In the United States, California has started a statewide effort to better the efficiency of freight transportation and to transition their freight transportation system to zero-emission technologies [13]. In 2015, Governor Brown requested that the California Department of Transportation, the California Energy Commission, the California Air Resources Board, and the Governor's Office of Business and Economic Development take part in this initiative. The goal of this action plan is to come up with strategies for a sustainable freight transportation freight system.

Technology is making it easier for everyone to accurately measure and control the impacts of urban freight transportation by constantly monitoring the outcomes. Researching for innovative strategies and technologies to reduce the impacts of urban freight transport in both big and small companies is ongoing.
The field of Operation Research plays an important part within most studies to determine the optimal selection of routes to transport goods.

4 Minimizing Freight Impact

The implementation of city based programs promote and direct research over successful best practices across the world on reducing impacts of urban freight transportation [7,13].
The implementation of policies and regulations by the city or state governments are critical for the reduction of pollution and congestion.
The implementation of land use and urban design strategies to create a freight transportation system is being considered. It will contribute to the city dynamics making it a better place to live. In fact, if well planned, some of these strategies may not only have one outcome but several. For example, designing small self-service booths around the city could not only reduce trips and therefore emissions, but also alleviate the congestion problems that urban freight generates. Moreover, land patterns and urban design can successfully be influenced by policy which could regulate the need of the presence of centralized loading docks, or the implementation of ideas like the self-service booths in most commercial buildings.
The implementation of mode shifting alternatives among the cities is also addressed. Currently, there are more resources to take advantage of one's transportation knowledge and all that is needed is to know how to efficiently implement these assets. For example, by creating a similar system to the self- serving booth for bicycles, couriers can find ways to avoid congestion in the urban core as well as reducing emissions and create a more pedestrian friendly environment for city dwellers.
Finally, we deal with the implementation of innovative technologies to help the accurate measure and control of the urban freight transportation impacts. These approaches would make it possible to improve the way people handle new technologies by constantly monitoring the outcomes from their own devices. It is a matter of taking chances and looking for the implementation of innovative technologies in both big and small companies so that the benefits of the implementation of a sustainable urban freight system can be accountable in every sector of the city.

5 Conclusion

In conclusion, this paper reviewed a number of strategies ranging from implemented policies to localized technology and modal shifts options in different countries to reduce the urban freight transport impacts. From this assessment we realize that there are cracks along the different strategies and programs that have not been tested. Also, we observe that we still need effective methods to collect data and develop tools to support the evaluation of different measures in terms of the short and long term potential and that there is an important gap relating to information about key stakeholders able to tackle the impacts of the transportation of goods in urban areas.

6 References

[1] Hicks, S. (1977) Urban freight. In: Hensher, D. (Ed.) Urban Transport Economics. Cambridge University Press.

[2] United Nations HABITAT (UNHABITAT) Cities and Climate Change (2011) [unhabitat.org/pmss/listitemDetails.aspx?publicationID=3086]

[3] Crainic, T. G., Ricciardi, N. and Storchi, G. (2004), 'Advanced freight transportation systems for congested urban areas', *Transportation Research Part C: Emerging Technologies*, Vol. 12 No. 2, pp. 119-137.

[4] Dablanc, L (2007) 'Goods transport in large European cities: Difficult to organize, difficult to modernize', Transportation Research Part A: Policy and Practice, vol. 41, no. 3, pp. 280-285.

[5] OECD/ENV Report (1998) Scenarios for Environmentally Sustainable Transport Organization for Economic Co-operation and Development 2003, Delivering the Goods -21st Century Challenges to Urban Goods Transport, OECD working group on urban freight logistics, Paris.

[6] Freight Transportation Planning for Urban Areas Chatterjee, Arun Institute of Transportation Engineers. ITE Journal; Dec 2004; 74, 12; ProQuest Guidance on measuring and reporting GHG emissions from freight transport operations (PDF guide online)

[7] BESTUFS (2007). Good Practice Guide on Urban Freight Transport. http://www.bestufs.net

[8] Allen, J, Anderson, S, Browne, M & Jones, P 2000, A framework for considering policies to encourage sustainable urban freight traffic and goods/service flows Report 1: Approach taken to the project, Transport Studies Group, University of Wesminster, London.

[9] Behrends, S. (2007), Novel rail transport services, Work Package 2 - Deliverable, FastRCargo, Department of Technology Management and Economics, Chalmers University of Technology, Gothenburg.

[10] Industry Canada and Supply Chain & Logistics Association of Canada. (2008). Green Supply Chain Management: Logistics and Transportation Services, A Canadian Perspective. www.ic.gc.ca/logistics.

[11] Deakin, E. (2001),'Sustainable Development and Sustainable transportation ' , Working paper, University of Berkeley, Institute of Urban and Regional Development.

[12] Russo, F., and Comi, A. (2011) 'Measures for sustainable Freight transportation at Urban Expected Goals and Tested results in Europe'. Journal Urban Planning and Development, pp., 142-152.

[13] California Sustainable Freight Action Plan Retrieved July 20, 2016, from http://www.casustainablefreight.org/app_pages/view/154

[14] Envision Freight http://www.envisionfreight.com/value/index.html%3Fid=introduction.html

CPU and GPU DVFS via analyzing of workload characteristic

Kyoungsu Jun[1], **Hyunmin Yoon**[2], **Yoonsik Choi**[1] **and Minsoo Ryu**[1]
[1]Department of Computer Science and Engineering, Hanyang University, Seoul, Korea
[2]Department of Electronics Computer Engineering, Hanyang University, Seoul, Korea
{ksjun, hmyoon, yschoi}@rtcc.hanyang.ac.kr, msryu@hanyang.ac.kr

Abstract – *I-EAS, which is a low power management using process scheduler, scales frequency by calculating workload for purely core usage. On the basis of this, we determine the energy-efficient frequency. In the low power policy of CPU, I-EAS calculates workload that the CPU imposes load on the memory using the performance monitoring unit (PMU), In the low power policy of GPU, I-EAS finds a job that uses a lot of memory to analyze the characteristics of the job of GPU to run and calculates workload that the GPU imposes load on the memory using a memory-intensive job. It is difficult to determine the energy-efficient frequency with the environment without PMU, we can solve the problem through the platform performance monitoring unit (PPMU). And analysis of I-EAS's GPU job did not take into account the characteristics of job excepting tiler job, our system considers characteristics of all job. In our system, we more specifically calculate workload to use memory in order to determine the energy-efficient frequency.*

Keywords: Dynamic Voltage Frequency Scaling (DVFS), Performance Monitoring Unit (PMU), Platform Performance Monitoring Unit (PPMU), Governor, Memory decomposition

1 Introduction

Current mobile devices despite their small size provide functionalities like a desktop computer, such as 3D gaming, drawing, editing documents, web browsing. However, the increasing performance of these mobile devices comes with the cost of high energy consumption. Charging these mobile devices frequently is not possible, therefore minimizing the energy consumption of mobile devices is a hot research topic. In order to reduce the energy consumption while the processor is actively running, dynamic voltage frequency scaling (DVFS) is used which decreases the operating voltage and frequency of the processor. However, by reducing the operating frequency of the processor the execution time of a task is increased. Therefore, to reduce energy consumption while meeting the real-time deadlines of tasks, energy aware Scheduling (EAS) schemes have been proposed [1, 2]. Another approach in reducing the power consumption of computing devices is advanced configuration and power interface (ACPI) which specifies how a computer's I/O system, operating system and peripheral devices communicate to each other about their power consumption requirements [3]. However, these existing techniques have the limit to optimize the energy consumption.

In DVFS, when a CPU core is running its optimum operating frequency is determined by the workload. The low energy consumption policy is implemented by measuring the workload during a given period and then corresponding frequency is determined using this workload. In Linux based systems, CPU, GPU and memory are all targets to reduce the power consumption. In previous research [4, 5], the implemented policy measures the memory workload through performance measuring unit (PMU) and the workload for the CPU core is calculated by excluding the memory workload. The low power policy of GPU used the information of GPU core provided by Linux system and calculated the workload. The low power policy of memory measured the workload using platform performance monitoring unit (PPMU). A performance analysis unit, PPMU, provided by Exynos SoC can measure performance data and can be used to analyze the system performance. The low power policy of memory measures the workload and determines the frequency using this workload measured by PPPMU. This workload is a rate of the CPU cycle count that of the memory bus used by CPU and devices over the entire cycle count used by memory bus.

However, if there is no PMU so low power policy of CPU cannot calculate workload using CPU core, CPU will not achieve efficient power management. In low power policy of GPU, it only considers tiler job used to memory decomposition. But when CPU gives jobs to GPU, only vertex job but also tiler job is transferred to GPU as a job. [6]. Therefore, both execution time are always equal when Linux system measures the time of vertex job and tiler job. Tiler job's phase of increase and decrease is similar to memory's workload phase, also vertex job's phase is equal to tiler job. So it is important to consider vertex job and tiler job for memory decomposition.

In this paper, we suggest two portions that are complemented with low power policy of CPU and low power policy of GPU. Low power policy of CPU calculates redefined CPU workload, and it entire memory bus cycle count from PPMU. And this rate is used in memory workload to calculate the workload to purely use the CPU core. Low power policy of CPU calculates memory workload of vertex job and tiler job

using the memory and workload using GPU core by utilizing memory workload.

2 Effective workload

Tasks that use CPU, GPU and memory intricately impose certain workloads on each of these components. The workload of a task in a given period is defined as the amount of time for which the task was using a CPU/GPU core or memory during that particular period. Two types of operations can be defined for CPU and GPU, those currently running on the core and those waiting for results of memory operations. If the effective workload that is workload using the core except workload is calculated, waiting the memory is possible, and if a decision of frequency using this effective workload is done, system can achieve effective power management with nearly equivalent performance.

3 Low power policy using effective workload

3.1 Low power policy of CPU

When the CPU is operating, it is possible to divide up into workload using the CPU and workload using the memory. Then, it is easy to calculate purely effective workload. As calculating purely core usage, I-EAS system obtains bus access cycle count for memory access and the executed instruction count through PMU.

$$W_{mem} = \frac{CNT_{BusAccess}}{CNT_{Instruction}} \quad (1)$$

$$Util^{CPU}_{new} = Util^{CPU}_{capacity} \times \frac{(100 - W_{mem})}{100} \quad (2)$$

CPU DVFS of I-EAS measures workload and determines the frequency of CPU using Linaro GPU governor when the scheduling events occur such as task wake up, enqueuer and dequeue. The workload that is measured by the kernel is calculated as effective workload purely using the CPU core except workload using the memory. Workload that the CPU imposes load on the memory is bus access cycle count portion of the executed instruction count. It represents equation (1). After that, we considered that effective workload is CPU utilization multiplied by the core usage rate (2).

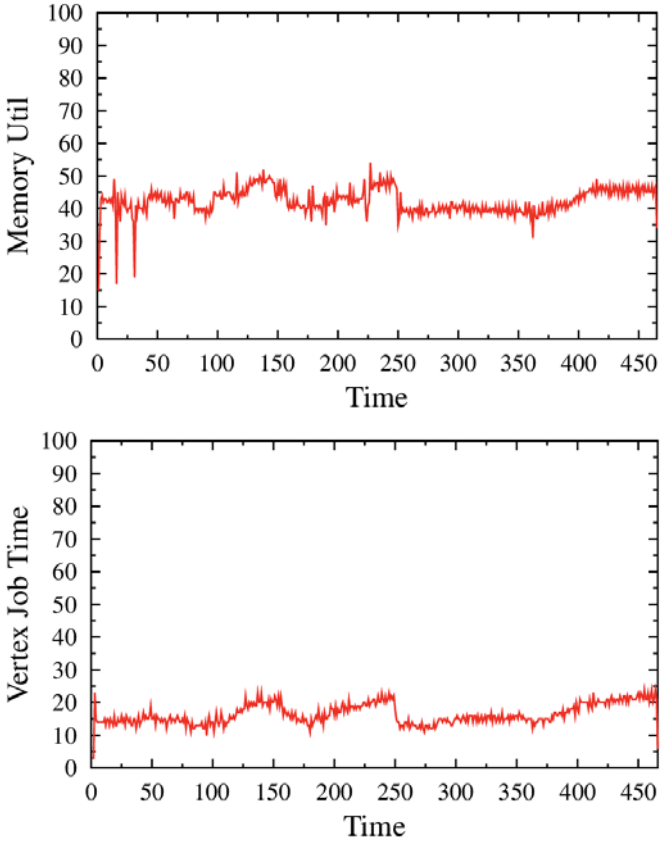
$$T^{Mem}_{Peripherals\_use} = \frac{CNT^{Mem}_{Peripherals\_R/W\_cycle}}{CNT^{Mem}_{Total\_cycle}} \times period \quad (3)$$

$$Util^{CPU}_{new} = Util^{CPU}_{current} \times \frac{T^{CPU}_{total} - \alpha \times T^{Mem}_{Peripherals\_use}}{T^{CPU}_{total}} \quad (4)$$

In the environment there is no PMU, we can measure effective workload. The PPMU measures both cycle count that peripherals use memory bus and total cycle count. We obtain the time that peripherals use memory bus during memory execution period. In Equation (3), The result must multiple a value of α. α means that each system is different and is tunable. After excluding the time using the memory from the entire time that CPU measure, we figure a rate of the time using the CPU core over the entire time. This rate is used to calculate the new effective workload of CPU in Equation (4).

3.2 Low power policy of GPU

GPU jobs measured in Linux system are vertex job, tiler job and fragment job. Vertex job transforms 3D geometry and projects onto 2D render target. Tiler job calculates updated region of render target and update region to frame buffer. Fragment job generates the final color for the render target for each pixel covered by a primitive. I-EAS found the result of the GPU job characteristics analysis that tiler job used a lot of memory when it did memory workload decomposition. The execution time of tiler job was similar to memory utilization at increase and decrease, so I-EAS analyzed that tiler job was a lot of memory workload.



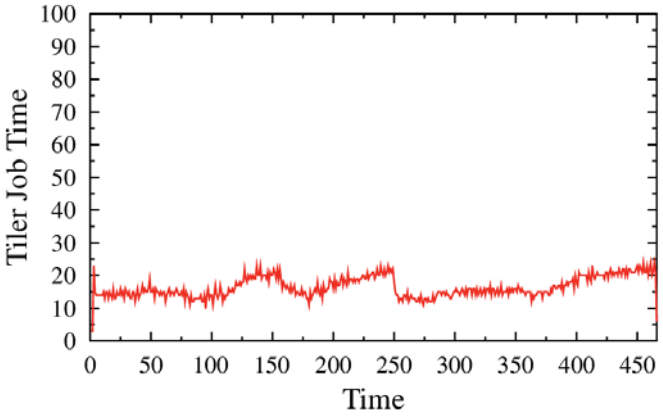


Figure 1. The relation between vertex, tiler job execution time and memory utilization

We analyze the relation among vertex job, tiler job and memory utilization through the experiment of Basemark ES2 is GPU benchmark. Memory utilization appears the increase and decrease according to vertex job and tiler job. The execution time of vertex job and tiler job is equal in Linux system because CPU gives GPU a job that includes vertex and tiler job. We concluded that vertex job and tiler job have workload to use the memory considering this relation.

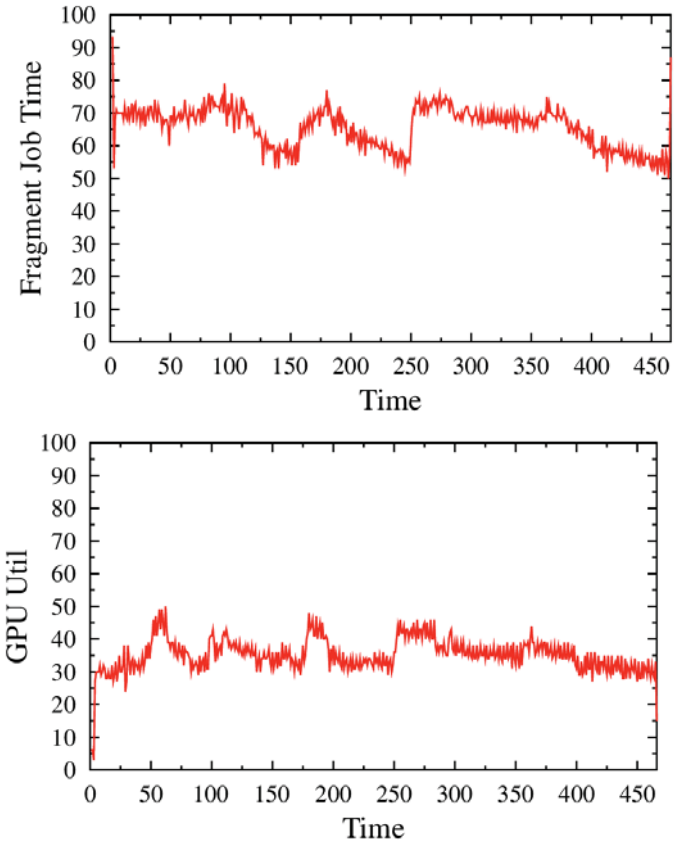


Figure 2. Figure 1. The relation between fragment job execution time and GPU utilization

We found that the flow of fragment job conversely moves to memory utilization and is similar to GPU utilization in Figure 2. So we analyze that fragment job use a lot of GPU core through this experiment. In consideration of the characteristics of the GPU job, it is essential to consider not only tiler job but also vertex job in memory workload decomposition.

$$Util_{new}^{GPU} = Util_{current}^{GPU} \times \frac{Freq_{current}^{GPU}}{Freq_{\max}^{GPU}} \times (1 - \frac{Exec\_time_{tiler}}{Exec\_time_{total}}) \quad (5)$$

In order to calculate workload using purely GPU core, I-EAS calculated memory workload that is the rate of execution time of tiler job over entire execution time. Memory workload obtained by memory workload decomposition is used for calculating core workload in Equation (5).

$$Util_{new}^{GPU} = Util_{current}^{GPU} \times \frac{Freq_{current}^{GPU}}{Freq_{\max}^{GPU}} \times \frac{Exec\_time_{fragment} + \alpha \times (Exec\_time_{vertex} + Exec\_time_{tiler})}{Exec\_time_{tiler}} \quad (6)$$

In consideration of between GPU job and memory utilization, we improve memory workload decomposition to figure the new utilization. For calculating memory workload, we add execution time of vertex job and execution time of tiler job. Then, We multiple this sum by α**.** The value of α is the rate of execution time of vertex job and tiler job using the memory over entire execution time. We measure each job execution time using memory through GPU benchmark test, then we calculate the α. GPU benchmarks are Basemark ES2, Antutu and Quadrant for 2D and 3D graphic. [7, 8, 9]

$$4099 = 7667 \times W_{vertex} + 7667 \times W_{tiler} + 30557 \times W_{fragment}$$
$$4140 = 9414 \times W_{vertex} + 9414 \times W_{tiler} + 42434 \times W_{fragment}$$
$$7662 = 11166 \times W_{vertex} + 11166 \times W_{tiler} + 22332 \times W_{fragment}$$

After 10 runs of each benchmark, we get the linear equation with three unknown workload vales and coefficients averaged among benchmark execution times. The formula is composed entire time using memory and execution time of each job. In the formula, workload of vertex job and tiler job is processed as one. As a result, we calculate that workload value of fragment job almost is 0 and workload value of vertex job and tiler job almost is 0.63.

4 Conclusions

In this paper, we analyze that task complexly impose load on CPU, GPU and memory and do memory workload

decomposition. Low power policy of CPU and GPU calculate effective workload purely using core for energy efficient frequency by using workload decomposition. In the low power policy of CPU, we calculate effective workload purely using CPU core through PPMU and use it to find energy effective frequency on this basis. Also in the low power policy of GPU, we calculate effective workload purely using GPU core through characteristic of GPU job and use it to find energy effective frequency on this basis.

5 Acknowledgment

This research was supported by the MISP(Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW)(R7116161027000102) supervised by the IITP(Institute for Information & communications Technology Promotion), partly by the ICT R&D program of MSIP/IITP [R01141600460001002, Software Black Box for Highly Dependable Computing], partly by the National Research Foundation of Korea(NRF) Grant funded by the Korean Government(MSIP) (NRF-2015R1A5A7037751) , partly by the Materials and Components Technology Development Program of MOTIE/KEIT [10046595, Storage solution for smart device], and partly by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the C-ITRC(Convergence Information Technology Research Center) (IITP-2016-H8601-16-1005) supervised by the IITP(Institute for Information & communications Technology Promotion)

6 References

[1] "Cpu frequency and voltage scaling code in the linux(tm) kernel." [Online]. Available: https://www.kernel.org/doc/Documentation/cpufreq/governors.txt

[2] Linaro: Energy Aware Scheduling. [Online]. Available: https://wiki.linaro.org/WorkingGroups/PowerManagement/Resources/EAS

[3] Hewlett-Packard: Intel, Microsoft, Phoenix, and Toshiba. The acpi specification: revision 3.0b (2008), http://www.acpi.info/spec.htm

[4] Kyoungsu Jun, Hyunmin Yoon, Pyoungsik Park, Minsoo Ryu, "An Energy aware scheduler and DVFS method using effective workload" The Korean Institute of Communications and Information Sciences, pp. 637-640, 2016.

[5] Pyoungsik Park, Hyunmin Yoon, Kyoungsu Jun, Minsoo Ryu, " A DVFS method using effective workload of CPU, GPU and Memory" The Korean Institute of Communications and Information Sciences, pp. 878-881,2015

[6] Ian Bratt, "The ARM® Mali™-T880 Mobile GPU" [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.50-GPU-Epub/HC27.25.531-Mali-T880-Bratt-ARM-2015_08_23.pdf

[7] [Online]. Available: https://www.basemark.com/

[8] [Online]. Available: http://www.antutu.com/en/index.shtml

[9] [Online]. Available: http://www.aurorasoftworks.com/

User-level Deterministic Replay via Copy-to-User Function Tracking

Hanjun Shin[1], **Seokyong Jung**[1] **and Minsoo Ryu**[1*]
[1]Department of Computer Science and Engineering, Hanyang University, Seoul, Korea
{hjshin, syjung}@rtcc.hanyang.ac.kr, msryu@hanyang.ac.kr

Abstract – *The size and complexity of computer software programs has rapidly increased in the recent years, therefore the possibility of occurrence of faults and bugs in the software is also amplified. Most of these bugs can be eliminated by repetitive debugging and execution of software during the software development and testing process. However, there are some exceptions such as synchronization problems which cannot be reproduced through repeated program executions. Such synchronization problems arise due to the occurrence of nondeterministic events, for example, interrupts or Inter-Process Communication (IPC). Record and Replay tools that capture the state of software by recording non-deterministic events are often used for software debugging. In the current research, we suggest a record-replay mechanism for user –level application running in single threaded environment based on kernel to user data transfer. The proposed approach is able to replay an application by capturing all the kernel-to-user level data transfers.*

Keywords: Software debugging, deterministic replay mechanism, analysis kernel and user interaction

1 Introduction

In order to provide necessary functionalities and resource optimization for a wide variety of hardware platforms, a large increase in the size of computer software programs along with complexity is witnessed in the recent years. The ever-evolving process of software development process therefore, has become difficult and the possibility of existence of software bugs has also increased. In order to eliminate these bugs, many debuggers are developed such as GNU Debugger that are useful in many situations. These debuggers can help to reproduce a problem in software by repeated program executions. However, all of the bugs cannot be resolved with the aid of debuggers, for example, synchronization issues. Such faults cannot be reproduced by simple debuggers because their occurrence depends on various external factors, which we call non-deterministic events. These non-deterministic events are generated by the external hardware devices generally through interrupts. In order to analyze and get rid of subtle bugs like synchronization, such non-deterministic events need to be captured and executed repetitively. A promising approach to solve such complex bugs is to record necessary events and then replay the software based on those recorded events. This approach is called deterministic replay as it allows to deterministically reproduce the events, analyze and fix the fault that appear in the recorded run. An example of the repetitive execution of nondeterministic events in debugger is UndoDB [1] ,which is an extension of GNU Debugger by Undo Software. UndoDB catches every nondeterministic event to create the snapshots and has the functionality to play-back the flow of program execution using these snapshots. But, this approach is adopted in initial software development and debugging environments and is not suitable for recording and reproducing a fault in real runtime environments.

In the past few years, a number of approaches for recording the runtime behavior of program and then replaying it offline have been proposed. Most of the suggested methods achieve it through record and replay of hardware interrupts [2-4]. Recording the hardware interrupts and replaying them later encompasses the whole system behavior. However, the behavior of a particular user application cannot be determined by tracing the hardware interrupts only. In order to record and replay a particular user application efficiently, the adopted methodology should be able to capture all the non-deterministic events that affect the application including the hardware interrupts.

The user-level application developers assume that their code is deterministic and will always give the same results. However, the non-deterministic events occurring in the system strongly influence program behavior in real run-time environments. Modern operating system abstracts the hardware-related services, (e.g. accessing network card), IPC and other kernel services from user-level applications. If the application needs to use such services, it is achieved through system calls. The results of execution of non-deterministic events in kernel mode as a result of system call invocation are stored in kernel memory area and sent to user space by kernel to user memory copy function, or the return values of the system call.

A number of record and replay approaches have been proposed in literature for replaying user-level applications based on analyzing, recording and replaying of invoked system calls, for example see [5-7]. System call analysis is performed

ISBN: 1-60132-434-0, CSREA Press ©

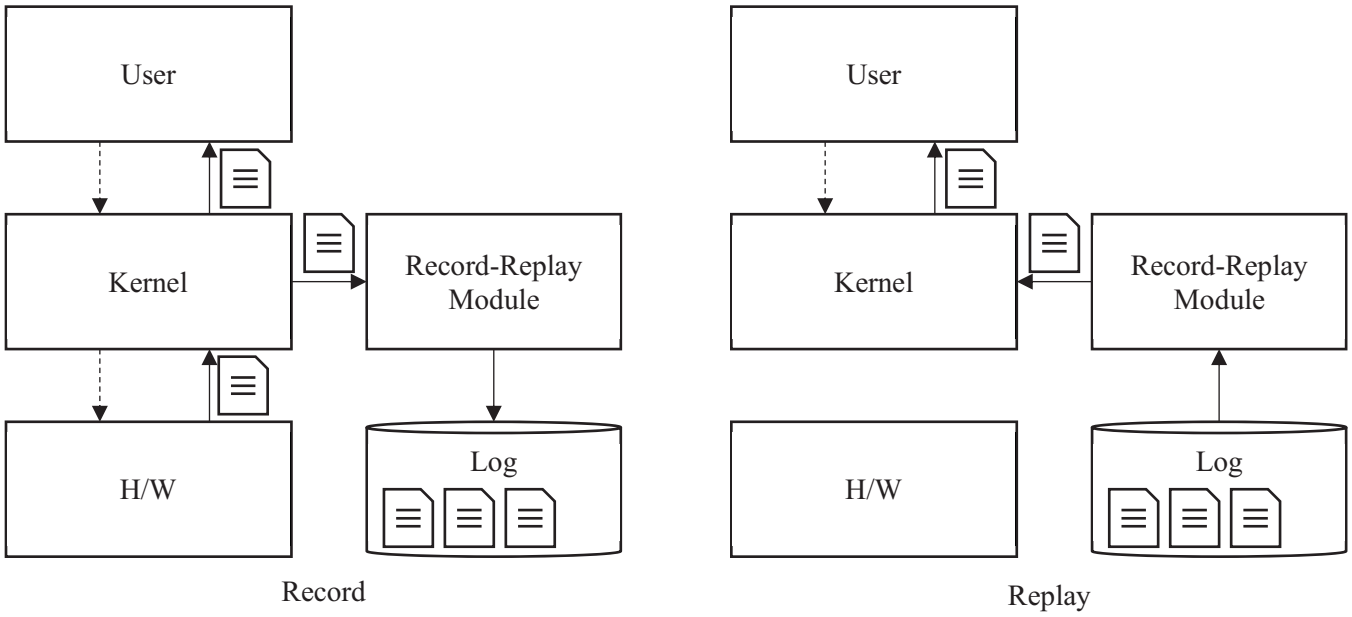


Figure 1. Record-Replay Mechanism

by keeping track of the changes in a particular memory address space which will be updated by kernel as a result of system call execution. However, such system calls analysis can detect the change in an address space through an address pointer passed as an argument to the system call and its returned value. However, there are few system calls for example, Linux system call exec() does not pass the address pointer as an argument, instead it writes the contents of executable file at user address space. Similarly, there are few arguments which are passed as opaque parameters whose type and function cannot be determined at the time of declaration, for example, Linux system call ioctl(). The purpose of each argument of the ioctl() system call is determined by its peripheral device driver, so it is not possible to figure out those arguments in advance. Therefore, such the methods using system calls cannot be used to accurately record and replay user applications.

In this paper, we suggest a record-replay mechanism for user application running in a single threaded environment using kernel to user data transfer. It detects all the non-deterministic events by capturing kernel-to-user data transfers which other approaches are unable to do. To be more specific we focus on non-determinism that is associated with copying data to user memory area from kernel area.

The paper consists of three sections. Section 2 describes our proposed mechanism for logging kernel to user data and replaying it. Section 3 gives a brief conclusion.

2 Overview of Record-Replay

2.1 Main Idea

The main purpose of our record-replay approach is logging of all the non-deterministic events during record time and generate those events during replay time. All non-deterministic events influencing a particular user application occur thorough invocation of various system calls. So, the basis of our proposed approach is logging all those system calls.

Note that all system calls may not trigger non-deterministic events. Therefore, selecting which system calls to log is important. Keeping in view this fact, we categorize the system calls as follows.

- I/O Function: I/O device and IPC system calls including disk I/O, network I/O such as file operation and socket operation
- Process control Function: fork, exec, exit
- Memory control Function: mmap, brk

I/O function calls are used for interacting with external peripherals or processes and so, the result of these system calls cannot be predicted by the user application, so we consider these as non-deterministic events. Second, process control functions manage a process status. Such calls only change or get the value of process control block, so the associated events can be considered deterministic. Finally, memory control functions manage a virtual memory address space which is not

affected by any hardware or external events, so they are deterministic.

The non-deterministic results of a system call are forwarded to user space by updating a particular user memory area and by the return values of the system call, depending on specific environment. For example, read() system call writes data from an I/O device or socket to user address space that is passed by an argument and its return value is the size of data. Thus, logging the non-deterministic factors caused by system call means to store that data and return value in a log. During replay, when the same system call is invoked, record-replay does not pass the data from hardware, instead the data is passed from the log.

2.2 Implementation

The general procedure of invoking a system call is shown in figure 1. First a user application invokes a system call and switches to kernel mode. Kernel processes a system call and if it needs to use hardware resources, it sends a request to hardware. When the I/O transaction has been completed, it sends the result of execution to kernel using interrupts. If the kernel needs to send data to user application, it copies the data to user area.

The proposed flow of recording a user application is as follows. As shown in Figure 1, when user application invokes a system call, in view of the categories defined in section 2.1, recorder first checks whether the system call needs to be recorded or not. If the system call is not a target for recording, then it is executed normally. However, if a system call is identified for recording, then after step when kernel needs to send data to user application, record-replay module captures the data from kernel . The data is stored in a log which includes precise order of events and the data associated. At the same time the system call is normally executed. During the replay process, the target system call is not sent to the kernel, instead, the data associated with that particular system call is sent to user application, through the logged data in the storage.

3 Conclusions

In this paper, we implement deterministic record-replay mechanism which handle the data transfer between kernel and user application. This mechanism provides to detect nondeterministic events more than to analyze only system call arguments. Our approach is useful that it can detect more events, and perform the deterministic record-replay more completely.

4 Acknowledgment

This work was supported partly by the ICT R&D program of MSIP/IITP. [R01141600460001002 , Software Black Box for Highly Dependable Computing], partly by the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the C-ITRC(Convergence Information Technology Research Center) support program(IITP-2016-H8601-16-1005)supervised by the IITP(Institute for Information & communications Technology Promotion), partly by the MISP(Ministry of Science, ICT & Future Planning), Korea, under the National Program for Excellence in SW)(R71161610270001002) supervised by the IITP(Institute for Information & communications Technology Promotion), partly by the Materials and Components Technology Development Program of MOTIE/KEIT [10046595, Storage solution for smart device], and partly by the National Research Foundation of Korea(NRF) Grant funded by the Korean Government(MSIP). (NRF-2015R1A5A7037751)

5 References

[1] "Increasing software development productivity with reversible debugging," *Undo Software,* 2014.

[2] G. Gracioli and S. Fischmeister, "Tracing interrupts in embedded software," in *ACM Sigplan Notices*, 2009, pp. 137-146.

[3] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, "Replay debugging of real-time systems using time machines," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003, p. 8 pp.

[4] D. Stodden, H. Eichner, M. Walter, and C. Trinitis, "Hardware instruction counting for log-based rollback recovery on x86-family processors," in *Service Availability*, ed: Springer, 2006, pp. 106-119.

[5] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, *et al.*, "R2: An application-level kernel for record and replay," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 193-208.

[6] O. Laadan, N. Viennot, and J. Nieh, "Transparent, lightweight application execution replay on commodity multiprocessor operating systems," in *ACM SIGMETRICS performance evaluation review*, 2010, pp. 155-166.

[7] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging," in *USENIX Annual Technical Conference, General Track*, 2004, pp. 29-44.