# Warehouse Pick Path Optimization Algorithm Analysis

Ryan Key
Edinboro University of Pennsylvania
rk096065@scots.edinboro.edu

Anurag Dasgupta
Edinboro University of Pennsylvania
adasgupta@edinboro.edu

**ABSTRACT**
Warehouse operational costs are heavily influenced by the efficiency in which workers are able to traverse the warehouse and gather items on orders around the warehouse that must be shipped to customers; this action accounts for over 50% of warehouse operations expenses. The act of traversing the warehouse is greatly optimized by following a designated pick path; however, algorithms for pick path generation are complex and heavily unexplored by the industry. Generating pick paths involves solving two common place graph theory problems: the shortest path problem and the traveling salesperson problem. We will analyze algorithms used for solving both of these problems and discuss the feasibility of generating pick paths through the use of the algorithms. We also introduce a simplified implementation to illustrate the viability of the described approaches.

**KEY WORDS**
Warehouse Pick Path Optimization, Traveling Salesperson Problem, Shortest Path, Algorithm Comparison

## 1. Introduction

A common goal of nearly all businesses is to reduce man-hours and increase the overall profit margin of the business. In warehouse related businesses, optimizing the efficiency of order picking can lend itself to great reductions in the time it takes orders to ship out, as well as, improving the overall effectiveness of its workers. Of all costs associated with warehouse operations, 55-65% of the operational funds are allocated towards order picking [1] [2], showing the importance of optimizing this phase of the warehouse process.
To better understand the problem at hand, we will now describe a common scenario in a warehouse and show how order picking fits into the mix. First, a warehouse is comprised of 3 primary components: receiving, storage, and shipping [2], Receiving is responsible for checking things into the warehouse; this is the entryway for all items into the warehouse stock. Once items have been received, they must be put away and stored. Storage can be of any form, although large shelving units, gaylords, and/or pallets are traditional options. The storage area of a warehouse is quite important and should be well organized to create an advantageous environment for order pickers. The final area, shipping, is similar to receiving. This is the exit for all items leaving the warehouse; any items ordered by customers of the warehouse must pass through this area before arriving at the customer's location [2]; a customer of the warehouse could be larger entities such as the operation of a large franchise storage warehouse, or a single customer that is ordering products from an online store.
After understanding the layout of a warehouse, we must look at the tasks performed by a warehouse. Nearly all activities at a warehouse are centered on receiving orders from customers. As an order comes into the warehouse, an individual in the warehouse becomes responsible for the order; this is the *order picker*. The order picker is responsible for gathering all items on the order from around the warehouse, also known as *picking*, and then placing them in the shipping area. Once in shipping, the order will be packaged in a box or pallet and shipped to the customer. The picking process can be time consuming and by far is the biggest operational expense of any warehouse [2].
Picking items for orders is the most costly part of the process, because order pickers must traverse the warehouse layout to find all items on the order, starting and ending in the shipping area; this is a *pick path*. For example, an order picker will print an order from the shipping area with 10 items on it. If there is a 20,000 square foot warehouse, it can be expected to have at least 500 unique storage locations throughout the warehouse defining where items are stored. This means, the order picker must determine an efficient route through the 500 locations to get to the 10 locations identified on the order, where any location can be travelled to from any other location; we know this because any practical warehouse layout will not create any isolated, or pendant, storage area. After the order picker has found all 10 items in the warehouse, they must then bring the items back to the shipping area so the order can be shipped and the order picking process can be started fresh.
Now to analysis this problem in terms of graph theory, finding a pick path involves solving two of the most common problems in graph theory; the Travelling Salesperson Problem (TSP) and the shortest path problem. The TSP is the problem of finding the shortest tour through n cities that visits every city exactly once, starting and ending in the same city [3]. Where in the case of pick path optimization, we want to visit every location on the order exactly once, generally starting and ending in shipping.
The shortest path/route problem comes in five varieties, two of which pertain to the work accomplished by this paper. The first being, finding the shortest path between some vertex, *x*, and some other vertex in the graph, *y* [4]. Depending on the implementation and algorithms used to optimize warehouse pick paths, this scenario will come

into play such as, wanting to make it from some last location on the order back to the single shipping location. More often, it will be the case where the following shortest path is sought: the shortest path between some vertex and all others [4]. This is the shortest path most often sought during optimized pick path generation, as distances between the current vertex and all other locations on an order commonly need to be found.

To summarize the TSP/Shortest Path problem encountered when solving the pick path problem: we are solving a TSP between all locations on an order; starting and ending in the shipping area (which is also defined by a vertex). Then as the TSP is being solved, at each location we encounter the further difficulty of finding the shortest paths between the current node and all remaining locations on the order. This occurs because we are solving a TSP which assumes a complete graph where all nodes are connected with one edge between them; however, this is not guaranteed to be the case in the warehouse layout. We are guaranteed all locations will be reachable from any location in the graph, so we must define the shortest path to each of these vertices. After finding these shortest paths, we can then treat the problem like a normal TSP, where all edges defining the path between nodes are treated as one edge with one minimum distance associated with it.

Section 4 of this paper will focus on the algorithms and enhancements that can be used to find optimal pick paths by solving the TSP and shortest path problem. Section 5 will offer insight into an implementation for finding an optimal pick path.

## 2. Related Work

There have been hundreds of papers published and dozens of algorithms developed around solving the shortest path problem alone, as a large number of mathematical optimization problems are mathematically equivalent to the shortest path problem [4]. In the same respect, the TSP has been analyzed by dozens of professions, researched to no end, and proved to be a member of the NP-Complete problem set, with numerous heuristics developed that present polynomial time solutions within a fair degree of accuracy [3].

In contrast, there has been limited research completed around warehouse efficiency, and more specifically pick path optimization. There is some degree of research related to the business operations of warehouses [2]; however, there is almost no research related to the algorithms required to solve the problems of optimizing warehouse operations. This is the gap in research we have aimed to fill throughout the course of this paper.

## 3. Contribution

With an astonishing amount of research in solving shortest path problems and the travelling salesperson problem, we aim to explore popular algorithms for solving these problems, taking a closer look at how each algorithm works and the practicality of generating optimal pick paths with these algorithms. We will look at the role of each algorithm in solving the warehouse pick path optimization problem and evaluate the characteristics of the algorithm. We will look at the timing complexity of these algorithms, as well as, the flaws and potential concerns for implementing each algorithm.

We also introduce a basic implementation used for solving the warehouse pick path problem. In this regard, we focus on the components of the implementation and the improvements that should be made before using the algorithm to generate optimized pick paths.

## 4. Algorithms

There are a plethora of shortest path and TSP algorithms available, this section will focus on a handful of popular algorithms used for solving these problems. We describe each algorithm and the way in which it works. We then make mention of its time complexity, closeness to the optimum solution, and give a brief analysis on whether the algorithm is practical for usage in generating optimized pick paths.

### 4.1 Shortest Path Algorithms

This section will focus on algorithms used to find the shortest path between some vertex and all others in the graph; algorithm enhancements will also be considered.

#### 4.1.1 Dijkstra’s

Dijkstra’s algorithm is used to find the shortest distance between some starting vertex and all other vertices in the graph [5]. Dijkstra’s algorithm is quite popular for its performance, with a worst case performance of $O(|E| + |V|\log|V|)$, where E = number of edges and V = number of vertices [3]. The algorithm is also easy to alter so that Dijkstra’s will not only return the distance of the shortest path to each vertex, but also the path to traverse.

In pick path optimization, Dijkstra’s is quite useful as it can be used at each location to find the shortest distance between this location and all remaining locations on the order. This is quite practical and is often exactly what we need to solve in the process of generating an optimal pick path in the midst of solving the TSP portion of the problem.

Section 4.1.3 further elaborates on enhancements that can be made to improve Dijkstra’s algorithm.

#### 4.1.2 Floyd’s

Floyd’s algorithm is used to find the all-pairs shortest paths, meaning that in one run Floyd’s can find the shortest path between all vertices on a graph [3]. This can be seen as extremely advantageous to Dijkstra’s in finding pick paths, because the algorithm can be run once before solving the TSP end of the problem. By running the algorithm once, and storing the result to be referenced throughout the solving of the TSP, we are able to greatly reduce the time spent determining the distance between the current location and all other locations on the order; however, we do not believe Floyd's will always be more efficient than running Dijkstra's at each location while solving the TSP, considering its worst case performance is $O(|V|^3)$ [5].

**4.1.3 Dijkstra Enhancements**

The reliability, popularity, and speed of Dijkstra's makes it a heavily implemented and researched algorithm, especially in the realm of map routing and GPS programming. Under this umbrella, there have been numerous algorithm enhancements suggested [6]. These enhancements can be used to potentially greatly reduce the time it takes Dijsktra's to run.

**4.1.3.1 Subgraph Partitioning**

One of the most reasonable enhancements for pick path optimization is the idea of partitioning a graph into a subgraph, where the subgraph contains a limited number of unused/untraversed vertices [7]. For example, if a GPS were determining an optimal route from Washington, DC to New York City, it would not need to consider any vertices in California or Florida, as it is unreasonable to traverse that part of the graph when travelling from DC to NYC. This has a practical application to the pick path problem, as there is no need to look at the south side of the warehouse if no locations on an order pertain to that portion of the warehouse.

The holdup with the subgraph concept is the fact that there must be an algorithm run to determine what subgraph should be looked at and then form that subgraph [8]. This can be a costly operation and it can be difficult to predict what vertices should be dropped when developing a subgraph of the warehouse per order. It seems that in most circumstances, it would be more costly to determine what subgraph to send through Dijkstra's rather than simply running Dijkstra's algorithm on the entire graph.

**4.1.3.2 Bidriectional Search**

Bidirectional search is an extension of Dijkstra's algorithm specifically targeting a two-node shortest path problem, when a starting vertex and target vertex are explicitly given [9]. When given these two points it is possible to create a mapping for the set of nodes and the set of edges such that Dijkstra's algorithm can be adapted to start running from the start vertex and the target vertex simultaneously, where each thread will meet in the middle of the path, reducing the time taken to find the shortest path between two points using Dijkstra's [9].

Bidirectional search initially seems like a practical enhancement for solving the pick path problem, although after considering the problem this is not the case. Dijkstra's algorithm is run in order to determine the shortest distance between the current location node and *all* other locations on the order. The pick path problem does not typically involve finding the shortest path between the current vertex and one other location vertex in the graph, except if there is only one location on the order. It may be the case that some warehouses would find it advantageous to implement this Dijkstra's enhancement for this special case but, in general, order pickers are able to efficiently find their own path to orders with less than three locations on them [2]. This enhancement is not recommended for use in pick path generation algorithms.

**4.2 Travelling Salesperson Algorithms**

This section will focus on algorithms used to find an exact optimized solution to the TSP, as well as, approximation algorithms that offer solutions within some guaranteed degree of closeness to the optimal solution.

**4.2.1 Exhaustive Search**

The exhaustive search algorithm offers the only implementation that can produce the guaranteed shortest tour to the TSP every time. This algorithm searches through all permutations of tours, computing the distance travelled by each; if a new shortest tour is found it is stored as such until all possible tours have been checked [3]. Exhaustive search will always produce the shortest path because it is looking at every possible tour that could be taken. This is ideal in terms of a guaranteed shortest path; however, the performance of the algorithm is quite awful; having a time complexity of $O(n!)$ [3] [4]. This is not a recommended approach for generating optimized pick paths.

**4.2.2 Nearest-Neighbor**

The nearest-neighbor algorithm is a very simple algorithm to understand and implement. The algorithm starts at some random city, travelling to the city closest to the current city, until all cities have been visited. Once at the final city, come home. This algorithm cannot guarantee any degree of accuracy as to how close it will be to the optimum solution [3]. For this reason alone, we do not recommend using this when generating optimal pick paths.

**4.2.3 Multifragment-heuristic**

The multifragment-heuristic algorithm works by looking at the edges of the graph, rather than vertices. The algorithm approaches the problem by creating a minimally weighted set of edges that makes each vertex in the graph of degree 2 [3].

The algorithm is as follows: first sort the set of edges by their weights and set the shortest distance set of edges to empty. Then, for the number of cities in the graph, add the shortest edge left in the set to the shortest distance set of edges, provided the addition of this edge does not make any vertex greater than of degree 2. After the loop has been completed, the shortest distance set of edges will contain the approximate shortest distance [3] [5].

This algorithm generally creates a more optimal result than the nearest-neighbor algorithm, but it also does not guarantee any degree of accuracy [3]. For this reason, this algorithm is also not an ideal implementation for the pick path problem.

**4.2.4 Ant Colony Optimization**

For solving the TSP, there are a number of different ant colony solution algorithms available, many of which are based on genetic algorithms. These algorithms are modeled after the natural ability of an ant colony to find the shortest path to their food source [10]. When ants arrive at decision points in their travels, they have no knowledge of what lies ahead of them or what distance must be travelled based on their decision. Since the choice is random, it can be expected that when presented with two directional choices (both ending at the same point), half the ants will go right and the other half will go left. Eventually, ants will be travelling to-and-from this location so ants will be choosing direction when headed both directions to-and-from the food source. As the ants travel, they release pheromones. After the ants have been travelling for a short time, the pheromone will accumulate on both paths; eventually the shorter path will have a much higher accumulation and this will begin to attract all the new ants to this path. Ants are able to discover the shortest paths between their food sources by measuring the amount of pheromone deposited on each decision path [10] [12].

One proposed Ant Colony algorithm for solving the TSP is the Ant Colony System (ACS) [10]. The algorithm's primary feature is the use of agents as *ants*. These ants work in a threaded, parallel fashion, simultaneously searching for a *good* solution to the TSP. The ants communicate on a global level, as well as, indirect communication through pheromone release on the edges. Each ant acts independently searching for a solution, using pheromones as a form of *memory* and making iterative improvements on its path selection. In the end it is proposed that the shortest path can be found by examining the pheromones left on each edge and selecting the maximal pheromone-weighted edges in order to form an optimal solution [10].

Dorigo presents numerous results of ACS tests in relation to other top-notch TSP algorithms [10]. Moreover, ACS presents accurate results for both small and large problems. The algorithm was able to produce the optimum tour in all tours with less than 100 cities in a minimal number of runs. For larger travelling salesperson problems (198 to 1577 cities), ACS was able to generate optimal paths within 3.5% error from the optimum [10]. This solution is recommended in terms of accuracy; however, it is not a practical implementation for many due to its degree of difficulty.

**4.2.5 Twice Around the Tree**

The twice-around-the-tree algorithm is a minimum spanning tree-based algorithm [3]. These types of algorithms leverage the connection between Hamiltonian circuits and spanning trees, where a Hamiltonian circuit minus one edge produces a spanning tree [3].

The algorithm works by first constructing a minimum spanning tree of the graph. Then, starting at some random node, perform a walk around the spanning tree that was constructed (using a Depth First Search) and keeping track of vertices passed through. Then search the list of vertices that was generated; delete all repeats of nodes so that each vertex only appears once, except the start/end vertex. The start/end vertex should appear at both the beginning and end of the list. This produces a Hamiltonian circuit that is an approximation for the shortest path between all nodes [3].

This algorithm can be performed in polynomial time, although its exact timing depends on the implementation of the first step, where a minimum spanning tree is constructed. An MST can be constructed using any popular algorithm such as Prim's or Kruskal's [3] [4].

Another benefit of this approach is that, it is guaranteed that accuracy of the shortest tour generated by this algorithm is at most twice as long as the optimum tour. This algorithm is recommended based on its guaranteed upper bound and the fact that the algorithm is performed in polynomial time.

**4.2.6 Christofides' Algorithm**

Christofides' algorithm works similarly to the twice-around-the-tree algorithm as it also works with minimum spanning trees. Christofides' utilizes more advanced implementations of graph theory to form a guaranteed lower cost tour than the previously discussed algorithms [3] [4] [5] [11].

Christofides' first creates a minimal spanning tree, $T$, using some known algorithm. Then create a set of all odd degree vertices, $V$. Then find a perfect matching, $P$, with the minimum weight of the graph over the vertices in $V$. This will create a set of minimally weighted edges without any common vertices from $V$. Then, add the edges from $P$ and $T$ to form a multigraph, $M$. A multigraph is simply a graph that allows parallel edges. Now form an Euler circuit from $M$, call it $E$. This will produce a circuit that visits every edge once. Now, remove edges that visit nodes more than once. This will create a Hamiltonian circuit, which as we previously defined is a solution to the TSP [3] [5] [11].

Christofides' algorithm can be performed in polynomial time and produces a minimal tour that is guaranteed to be within 1.5 times the optimum tour [11].

This algorithm is highly recommended for implementation in finding an optimal pick path.

## 5. Code

Throughout the research of this paper, a small case study project was developed to show that the algorithms described could be implemented to create a usable pick path generator. The application is a C# Windows Form application that provides the basic implementation to create a warehouse layout in the form of a graph, save it, and then use it to generate an optimal pick path based on a handful of algorithms described above.

The initial step to use this application is the process of converting a warehouse layout into a graph. We will use *Figure 5.1* as an example throughout this section. First we must define the components of this layout. All whitespace in the layout represents aisles in the warehouse that can be traversed to travel around the warehouse from location to location. The large rectangular gray square in the bottom of the layout is the shipping area of the warehouse. As described in Section 1, this is typically the start and end point of the pick path. All remaining gray squares are storage locations in the warehouse such as shelving units or pallets. It is also important to note, distances are associated with each aisle and the shelving units; this comes into play as we move through the transformation of the layout into a graph.

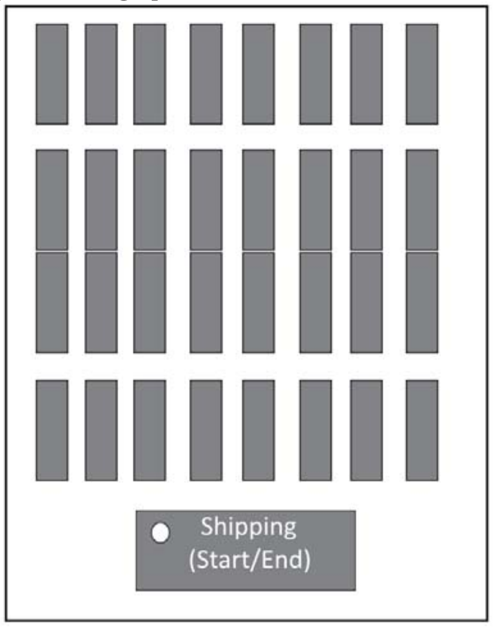


**Figure 5.1: Warehouse Layout**

After we have our layout defined, we can begin the process of turning this into a graph that can have traditional TSP and shortest path algorithms applied. The first step in this transformation is to create vertices at the intersection of all aisles, we do this because each aisle is an edge and the edges meet at intersections. Then draw each aisle as an edge.

Now, it is time to assign weights, of a uniform unit, to each edge. These weights represent the distance from the middle of one aisle intersection to the center of the neighboring aisle intersection. After assigning weights, it is time to randomly and uniquely assign each vertex an ID. This is done so that the user is able to interact with the Windows Form application, because the vertex ID correlates to an application vertex name. With that said, the application presents nodes to the user in the naming convention of "v1" to "vMax#Vertices", so for the sake of simplicity, we will name our layout graph vertices as such. After completing the process defined above, a graph similar to *Figure 5.2* is developed. Note we must also select a vertex to represent the shipping area at this time.

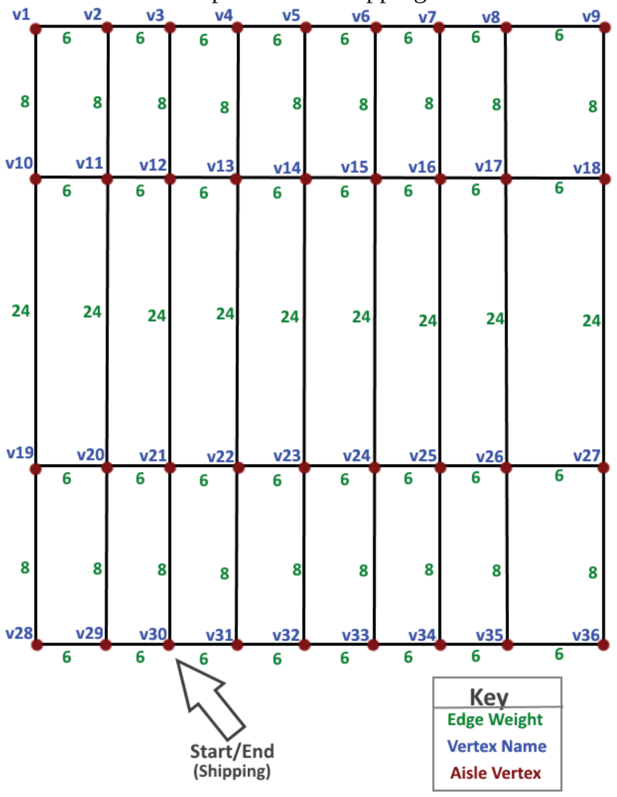


**Figure 5.2 Warehouse Layout Graph**

It can be seen that the red dots represent our vertices; the blue represents the identities of our vertices. We see we have a total of 36 vertices in our layout. Next, create all of the edges and place the weight along each edge in green. It is recommended that a thorough comparison of *Figure 5.1* and *Figure 5.2* be done to clearly see the figures represent the same warehouse layout. The entire reason for drawing this graph layout is to prepare all metadata that is collected by the application; planning and creating this image representation helps to reduce data entry errors and greatly reduce the time taken to turn the layout into a graph theory applicable problem. After our graph and its metadata are complete, we can begin using the application.

At this point it is important to note that this graph is a representation of the warehouse layout that will allow us to traverse from any aisle intersection to any other aisle intersection in the warehouse. This is different from the pick path problem, as the pick path problem travels from storage location to storage location. We eliminated the location vertices from our graph, as the same algorithms and processes apply to a graph going from aisle intersection to aisle intersection as a graph traversing location to location. This is true, because locations are found in our storage units. Our storage units are represented by the edges in our graph (see *Figure 5.2*). This means if we want to work with locations, rather than aisle intersections, we turn one aisle edge into multiple location edges. For example, if (from *Figure 5.2*) the storage unit between v18 and v27 contained 5 locations, we would add 5 vertices to the one edge of length 24. These 5 locations would then be joined by smaller-portioned edges whose sum would add up to the original length of 24. This shows that by performing our analysis on the graph of aisle intersections, we are able to create simplified tests with fewer vertices without loss of generality for locations. As an example of a complete location graph see *Figure 5.3*; it shows pink dots that represent each location. It is easy to see that this graph has the same properties as *Figure 5.2*.

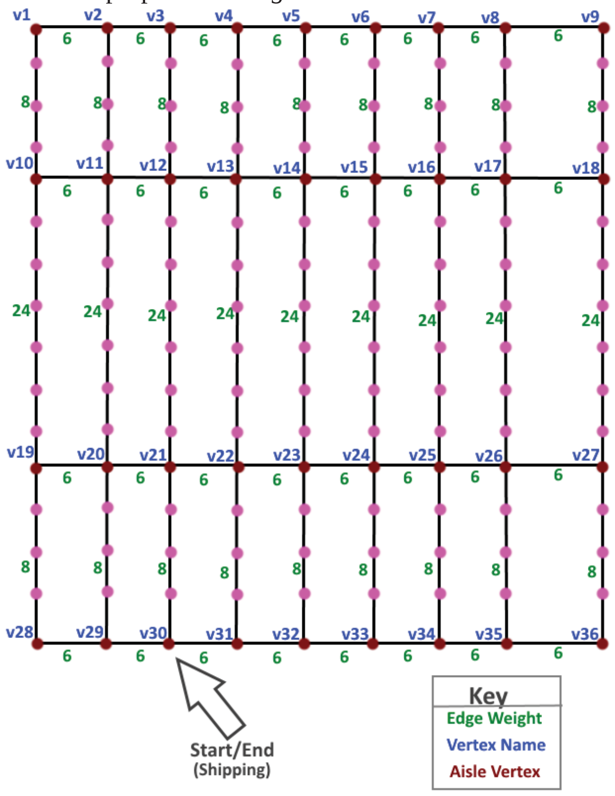
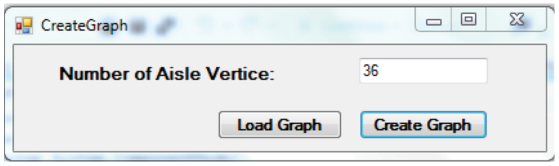


Figure 5.3 Layout Graph with Location Vertices

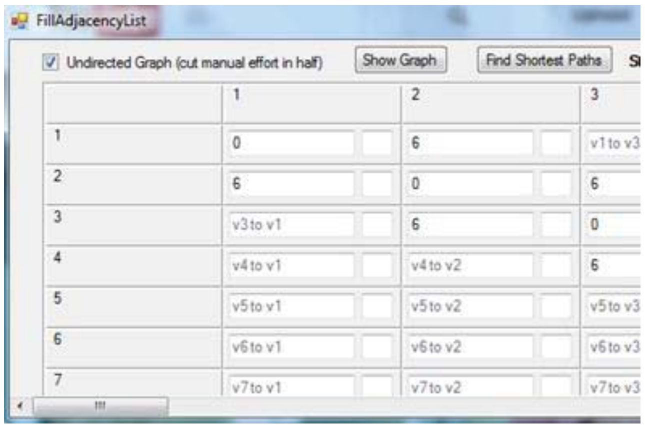
Now we must enter the graph from *Figure 5.2* into the C# application. After launching the application, we first enter the number of vertices and select “Create Graph”, see *Figure 5.4*.



Number of Aisle Vertice: 36

Figure 5.4 Create Graph with Number of Vertices

After selecting “Create Graph”, the application dynamically generates a form that is essentially an adjacency matrix. Each cell of the matrix has two input cells, we enter the distance between the vertices in the first input cell, ignoring the second. To fill this adjacency matrix form, we translate the metadata from our drawn layout graph in *Figure 5.2* to the adjacency matrix, entering the distance weight for each edge as in *Figure 5.5*.

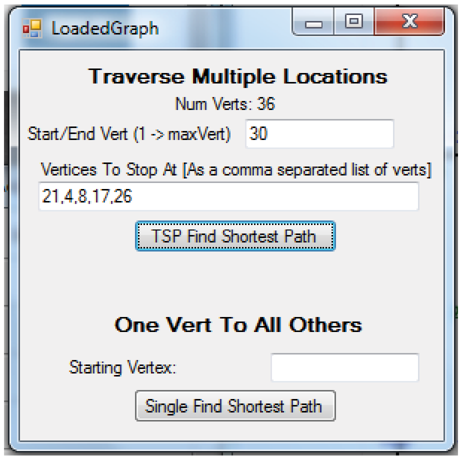


[✓] Undirected Graph (cut manual effort in half)

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 6 | v1 to v3 |
| 2 | 6 | 0 | 6 |
| 3 | v3 to v1 | 6 | 0 |
| 4 | v4 to v1 | v4 to v2 | 6 |
| 5 | v5 to v1 | v5 to v2 | v5 to v3 |
| 6 | v6 to v1 | v6 to v2 | v6 to v3 |
| 7 | v7 to v1 | v7 to v2 | v7 to v3 |

Figure 5.5 Fill Adjacency Matrix

After entering all of the metadata from the graph, we need to “Save Graph” so we can use this graph to solve the pick path problem from another form. After saving the graph, go back to the form from *Figure 5.4* and select “Load Graph”, select the graph that was just created and saved from the prior step. Now, enter the vertex that is the shipping area and the list of all vertices to visit as seen in *Figure 5.6*.



Traverse Multiple Locations

Num Verts: 36

Start/End Vert (1 -> maxVert) 30

Vertices To Stop At [As a comma separated list of verts]

21,4,8,17,26

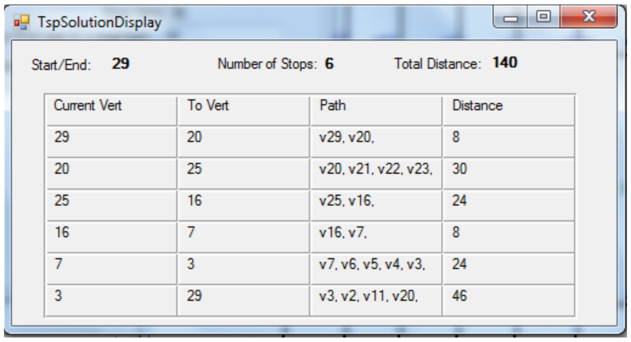
One Vert To All Others

Starting Vertex:

Figure 5.6 Find a Tour

After entering all information, select “TSP Find Shortest Path”. This generates an optimized pick path for the specified vertices using Dijkstra’s algorithm to find the closest neighboring vertex to each current vertex as we traverse the graph. The application uses the nearest-neighbor approach for solving the TSP end of the pick path problem, and Dijkstra’s algorithm to find the shortest paths.

After the optimal tour has been found using the above algorithms, it is displayed to the user. It is important to note that this form starts counting vertices at 0, rather than 1 so all vertex names are decremented by a constant of 1. As *Figure 5.7* shows, the form displays the vertices in the order in which they should be traversed, showing the exact path to get to each location as well as the path distance. All of these components make it easy to piece the tour together and present a usable path to the end users.


TspSolutionDisplay

Start/End: 29 Number of Stops: 6 Total Distance: 140

| Current Vert | To Vert | Path | Distance |
|---|---|---|---|
| 29 | 20 | v29, v20, | 8 |
| 20 | 25 | v20, v21, v22, v23, | 30 |
| 25 | 16 | v25, v16, | 24 |
| 16 | 7 | v16, v7, | 8 |
| 7 | 3 | v7, v6, v5, v4, v3, | 24 |
| 3 | 29 | v3, v2, v11, v20, | 46 |

**Figure 5.7 Pick Path Tour**

This application shows the feasibility of using the algorithms described in Section 4 to generate optimized pick paths using simple vertex and edge data structures. This application does not; however, implement a recommended TSP algorithm, as the nearest-neighbor approach does not guarantee any degree of accuracy in regard to the optimum tour [3]. With a TSP algorithm guaranteeing an upper bound, this application would be far more reliable and recommended for use.
In short, we are able to demonstrate the practicality of using advanced TSP and shortest path algorithms for solving the optimized pick path, although further implementation is required to guarantee any degree of accuracy.

## 6. Future Work

It is our goal to implement an optimized pick path finder that implements a TSP solution with a guaranteed upper bound to the optimum tour. This will involve reworking the vertex and edge data structures, as well as storing them in a more optimal data structure than a list; preferably using a data structure that closely lends itself to finding a minimal spanning tree based on the fact that many upper bound TSP solutions first find a MST before solving the TSP. This will make it more efficient to implement algorithms such as Christofides' or twice-around-the-tree.

## 7. Conclusion

In conclusion, pick path optimization is a component of warehouse operations with much room for improvement in efficiency. To optimize the creation of pick paths, further research must be done in heuristics for solving the travelling salesperson problem with tight upper bound guarantees. The algorithms analyzed in this paper are capable of generating pick paths, although to guarantee any degree of accuracy to the optimum pick path tour, more advanced and complicated algorithms must be implemented such as Christofides' or twice-around-the-tree. With the implementation of these algorithms, optimal pick paths can be reliably generated and used for directing order pickers.

## References

[1] Theys, C., Braysy, O., Dullaert, W., Raa, B., 2010. Using a TSP heuristic for routing order pickers in warehouses. *European Journal of Operational Research 200* (3), 755–763.

[2] Bartholdi, J.J., Hackman, S.T., 2006. *Warehouse and Distribution Science*. Release 0.96.

[3] A. Levitin, *The design and analysis of algorithms 3$^{rd}$ Edition* (Upper Saddle River, NY: Addison-Wesley, 2012).

[4] N. Deo, *Graph theory with applications to engineering and computer science* (Englewood Cliffs, NJ: Prentice-Hall, 1974).

[5] T. Cormen, C. Leiserson, R. Rivest, & C. Stein, *Introduction to algorithms* (Cambridge, MA: MIT Press, 1990).

[6] F. Zhan, "Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures," *Journal of Geographic Information and Decision Analysis, Vol.1*, No.1, pp. 69-82, 1998.

[7] Q. Song, X. Wang. Partitioning Graphs to Speed Up Point-to-Point Shortest Path Computations. *Decision and Control and European Control Conference (CDC-ECC), 2011 50th IEEE Conference on*. IEEE, 2011.

[8] R. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up Dijkstra's algorithm. *4th International Workshop on Efficient and Experimental Algorithms* (WEA), pages 189–202, 2005.

[9] Pohl. I. Bi-directional search. In B. Meltzer and D. Michie (Eds.). *Machine Intelligence 6* (American Elsevier. New York. 1971) 127-140.

[10] Dorigo, M., & Gambardella, L. M. (1997). Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation, 1* (1), 53-66.

[11] CHRISTOFIDES, N. 1976. Worst-case analysis of a new heuristic for the traveling salesman problem. *Symposium on New Directions and Recent Results in Algorithms and Complexity*, J. F. Traub, ed. Academic Press, Orlando, Fla., p. 441.

[12] F. Chen, H. Wang, C. Qi, and Y. Xie, "An ant colony optimization routing algorithm for two order pickers with congestion consideration," *Computers & Industrial Engineering, vol. 66*, no. 1, pp. 77–85, 2013.