Pathfinding In A Grocery Store

Clark Ohnesorge
ohnes013

May 10, 2020

1 Abstract

This paper explores two solutions for finding the shortest path between all specified pairs in a graph: a basic version of Floyd-Warshall's Algorithm and A* with the Manhattan Distance between points as a heuristic. Floyd-Warshall's Algorithm here uses one adjacency matrix and another matrix for storing the acutal path between specified pairs. A* uses a linked list method of remembering paths. The results show that A*, as implemented for the example problem, is significantly faster at the cost of increased memory usage. However, with a more deliberate design the increased cost for using A* could be reduced, and the computation time for Floyd-Warshall's Algortihm could lessened. Next, using the all the paths generated by either algorithm, this paper describes the implementation and analysis of a method of solving the Traveling Salesman Problem: a Greedy Heuristic Algorithm. The Greedy Heuristic Algorithm also produces one a very short path that connects every specified point of interest in a graph.

2 Introduction

With the rise of online shopping, competition amongst in-person retailers continues to heighten. While people have always valued their time, an increasing number of households is short on time despite how fast paced our world has become. Now more than ever, retailers are highly invested in designing and implementing new features and layouts that will help keep them relevant in our rapidly advancing technological age. Perhaps the most important factor to consider is the goal of consumers. Many instances of shopping are simple; an individual might come in with a specific product in mind, go straight to the product, and then leave. Here it may be in the retailer's best interest to make the trip efficient, to encourage repeat visits. However, one kind of shopping is consistently more complex: grocery shopping. A vast number of customers come in to the supermarket with some sort of list of essentials and walk throughout the store picking them up. While this might be a short daily trip for some, this is more often a drawn out, weekly experience done to stock up. Thus while these weekly shoppers may come in with some essential list, if they pass by goods that catch their eye they will often fold to their momentary craving and add the new good to their cart. This is good for the bottom line of grocers; the more products people buy the better. The question becomes: how can we get people to pass more products and make snap decisions? Again, people value their time now more than ever so gone are the days of idly wandering a store. It is far more likely that the discerning shopper will attempt to limit time spent at the grocery store, and thus fewer products will be shown off. That leads to the problem that I am considering here: the creation of grocery store layouts that can be tested with different pathing algorithms. With the assumption that shoppers will try to take the shortest path they can reasonably come up with, this software is able to simulate pathing through a store layout using a few different algorithms. These algorithms have different advantages given different situations, and I will go into more detail through this study. Before starting though, I had to examine relevant resources to see how pathing has been done before.

3 Related Works

The main goal of my project is to create an efficient path through multiple items, or nodes, in a grocery store. When considering solutions, one way of approaching this problem is as a TSP, or a Traveling Salesman Problem. However, solving a TSP relies not only on knowing the locations nodes, but also the distance from each node to every other node. Therefore, I will first discuss solutions to the All-Pairs Shortest Path Problem, finding the shortest path from each node to every other, and then I will discuss different approaches to solving a TSP.

3.1 All-Pairs Shortest Path Problem

Finding the shortest path between nodes is a well trodden area of computer science, and there exists more than one path finding algorithm. Here I'll examine three algorithm options laid out by a few sources: Djikstra's Algorithm, the Floyd-Warshall Algorithm, and Genetic Algorithms. First is Djikstra's Algorithm (DA). DA is an algorithm that visits every vertex, updating the shortest path to every vertex from the starting node as it goes [7]. In this way the shortest path to the goal vertex can be found, but it is first necessary to visit every vertex. Basically, this is an uninformed, base version of the A* algorithm. While A* uses initial heuristics, or estimates, to determine which vertices to explore DA just visits every vertex. At first glance A* seems to be a complete upgrade, seeing as it reduces the number of vertices that must be visited. However, it may not always be the case that there is prior knowledge of the problem, thus making DA the necessary algorithm. In grocery store pathing a good heuristic could be the straight line path from each location to the next goal item. Both [7] and [3] only discussed DA, not A*. In both of those studies DA of course found the optimal solution path, but in terms of computational time performed best for small item sets, specifically for item sets of less than 5 in [3]. For many physical, distance based problems a reasonable heuristic is easier to create, but there may be times where DA is the preferred algorithm. Perhaps the performance of DA could have been improved upon in [7] [3] by implementing a solution proposed in this next article. In this article [9] a version of DA is proposed that performs 40% fewer operations than a traditional DA as laid out here [4]. Beyond that, when applied to the all-pairs problem it performs 25% fewer operations than the Floyd-Warshall Algorithm[9].

The Floyd-Warshall Algorithm (FWA) is another option for finding the shortest path. It finds the shortest distance from each vertex to every other vertex and creates a matrix of those distances. From there the optimal distance to a goal vertex can be found. In [3] FWA performed the best for number of items greater than 5 and less than 20, which was the max number of items tested. However, FWA has a cubic time complexity proportional to the number of vertices and a large number of items could greatly impact performance. A solution is proposed in [8] with regards to finding the shortest path for a navigation system with a large number of nodes. This seems unnecessary for the context of grocery shopping because, intuitively, having significantly more than 20 discreet items seems rare.

Finally is the idea of using a Genetic Algorthim (GA) to solve the shortest path problem. The authors acknowledge initially that their GA was not made with the intention of competing with traditional shortest path algorithms [5]. GA's do not always find the optimal solution, they are useful for quick "good enough" solutions for exceedingly complex problems. Nevertheless the GA demonstrated here was tested on a set of 32 vertices and found the optimal path 98% of the time [5]. This is promising, given that finding the exact optimum path may not be as important as very quickly finding any ”good enough” path.

3.2 Approaches to Solving a TSP

Once a set of nodes and all of their shortest paths is set up, the groundwork is set for a Traveling Salesman Problem (TSP). For small numbers of nodes an optimal solution to a TSP can reasonably be found, however as sets of nodes get increasingly large it becomes impractical, or inconceivable, to work towards an optimal solution [6]. This is why emphasis is placed on algorithms for solving TSPs that find *more correct* solutions, rather than *optimal*, in reasonable amounts of time.

First are some more straight forward algorithms. One is the Nearest Neighbor approach. This solution travels from node to node always selecting the shortest path and avoiding loops. This is perhaps the greediest and most obvious choice, yet it performs decently and relatively efficiently [1]. Next is a Greedy Heuristic Algorithm proposed in the same paper as the Nearest Neighbor Approach [1]. For this solution, every edge between nodes is sorted, and then the shortest edges are picked as long as loops are not formed. This is another straightforward and direct approach, but again it performs admirably. While the result is slightly more optimal than the Nearest Neighbor approach, there is a trade-off, albeit not a large one, in time complexity [1].

Besides these 2 more basic approaches, there a few solutions that attempt to close the gap even further to an optimal one. The first approach is the creation of Genetic Algorithm (GA). Different paths are created, with the best path ”reproducing” in an attempt to create children with their parent’s best parts and random changes [1]. For a GA to be effective its creator must come up with a good heuristic, i.e. a way to judge the fitness of individuals that drives them in an optimal direction. Not only can finding a good judge of fitness be difficult, but implementing that judge can be complex as well. Here [1] the results were less than pleasing, as the GA performed much worse and much slower. This is likely a problem with the fitness test used in the article, but if other algorithms perform well enough it might not be worth the time to find a GA that is more effective. Next is another algorithm based off of nature: a Thermodynamic Approach to the TSP. In this approach a Monte Carlo algorithm is used to approximate solutions to TSPs. First an arbitrary ”solution” to the TSP at hand is chosen. Next a second solution is proposed by adjusting part of the previous solutions path. If this path is smaller than the previous path, select it. Otherwise use a ”temperature value” to decide whether to replace the old path with the new path[10]. As previously stated, this boils down to a form of a Monte Carlo algorithm. The advantages here, like in the use of a GA, is to try to avoid local minima that would threaten to hide a more optimal solution. In the study the Thermodynamic Algorithm does manage to perform well when the user chooses a "good" temperature value for the system, but not exceptionally. This algorithm, while better than the GA, offers little advantage over the greedy algorithms and has the disadvantage of speculating over the correct temperature value for a given problem. This article is more about opening up the door for future refinement of the idea, and the authors put it well when they say "There is perhaps more poetry than mathematics in the speculations presented in this section"[10].

Finally is the most complex of the algorithms discussed here: K-Length String Optimization [6]. This solution starts by attempting to find the shortest path from one node to another, with the caveat of going through a set S of other nodes. It solves this by recursing down, finding the shortest path through nodes in S and then bringing these solutions together at the end. This algorithm is intriguing because it guarantees optimal solutions for sets of $N \leq 13$, and does so fairly quickly with the time required ranging from 60 milliseconds for a 9 node problem to 1.75 seconds for a 13 node problem. The major drawback is the limitation on how many nodes can be used for the problem; as more nodes are introduced the procedure becomes very time consuming without the guarantee of an optimal solution [6].

While the approaches to solving a TSP can vary greatly, they share a common theme about the trade-off between optimality and speed. The general consensus leans towards the most practical approach: often these are problems where a *near-optimal* answer is "good enough" and getting there quickly is the next most important.

4 My Approach

4.1 User Interface

For the user interface I primarily wanted to create something that was quick and easy for me to use in setting up multiple layouts. The scope of this project did not include the creation of a package for Graphical User Interfaces (GUIs); instead I decided on the Tkinter toolkit, a basic but powerful package. The layouts will be a grid of differently colored cells, so to create my layouts I implemented two objects: Cells and CellGrids. Cells are the individual cells in the grid. They contain information about themselves like their color and their coordinate location in the grid. They also have a function called draw that is used to graphically represent themselves. Black squares are the shelves that can't be walked through, red squares are the squares that must be reached to grab items, the green square is the entrance of the store, and of course white squares are walk-able space. While CellGrid is mostly a container for a set of Cells, it also handles the mouse-click events for drawing on the GUI canvas.

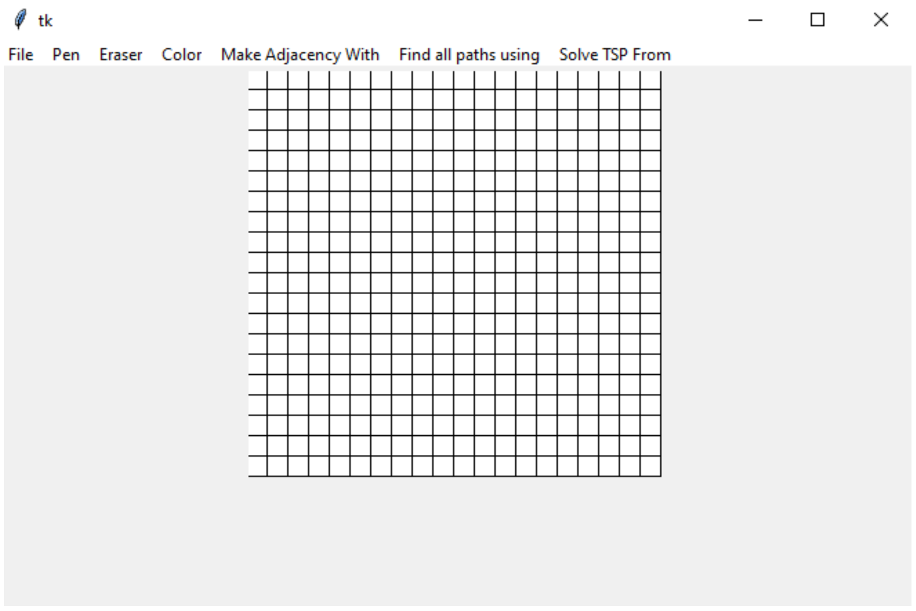


Figure 1: Blank Canvas

4.2 All Pairs Shortest Paths

The first problem I had to solve was generating the basic information needed to set up a TSP: I needed the shortest path from each shopping list item, which I will refer to as nodes, to every other node. I did this with two different algorithms.

The first algorithm that I used was Floyd Warshall's Algorithm (FWA) [3]. Initially, an adjacency matrix has to be created for FWA to be performed on. I set this matrix up by examining the grid drawn by the user and treating every cell as a node, with a distance of 1 to each directly adjacent cell. I then performed FWA on this adjacency matrix. Every time the adjacency matrix for FWA was updated I made sure to update another matrix that kept track of the actual path to get from cell to cell, not just the minimum distance. This was important for actually visualizing the path.

The next algorithm that I used was a version of the A* algorithm. A* is an algorithm that finds the shortest path between two specified nodes. Thus, I had a list of all of the shopping list item nodes and then performed A* on every pair of nodes. Here, I created a new class called Node, which was similar to Cell except that colors where not stored and a new pointer, parent, was add to keep track of the which Node the current Node came from. In essence this creates a path through a chain of Nodes. At the start A* adds the initial node to a priority queue and then pops it off to find its neighboring Nodes. Those neighbors have their distances calculated, and if they are not in the priority queue they are added to it. If they are in the priority queue already, the distance to that Node is updated if it is better than the previously stored value. A* makes use of something called a heuristic to determine if a path is worth exploring. For my heuristic I used the Manhattan Distance from the Node in question to the destination Node. At the end, the destination Node is returned. This destination Node will have its distance from the initial Node stored in it, and also a chain of parent pointers back to the initial Node.

4.3 Solving the TSP

I had to solve the TSP for both cases: using an adjacency matrix from FWA and using the Nodes created using A*. While some of the very specific details about which structure was being called were different, the general process was the same. I used a Greedy Heuristic method for trying to find the shortest path between all the nodes [1]. First I sorted all of the distances, i.e. edges, between grocery store item nodes. I chose to use the default sort() function provided in Python. This sort algorithm is called TimSort; it is a hybrid between insertion sort and merge sort. It is actually stable and quite fast, with an average case performance of $O(nlog(n))$ [2]. Next I made a pass through the edge list to remove the duplicates of every edge that resulted from finding the distance between every node. Now that the list of edges was ready I was able to greedily select the shortest edges by starting at the beginning of my edge list. The conditions for adding an edge to path were: 1) The nodes that the prospective edge connected must not already be connected to two edges and 2) The new edge must not create a cycle. Checking for a cycle initially seemed tricky, however it is actually simple enough. For each prospective edge I performed Breadth First Search to try to find a cycle. This adds little overhead, since there are so few edges in each iteration of the path we are creating. Finally, since we were avoiding a cycle until this point, I find the last two nodes without two edges and connect them.

5 Experiment and Results

In this paper there are two methods being compared: 1) An adjacency matrix approach with all shortest paths found by Floyd Warshall's Algorithm and 2) A Node based approach with all shortest paths found by the A* algorithm. For both of these methods I will determine the time and space usage in a few settings: 1) A small layout of 10x10 cells, 2) A medium layout of 20x20 cells, and 3) A large layout of 30x30 cells. Based on the size of the layout I assume that the number of items being shopped for is higher. Small layouts have 5 items, medium layouts have 10 items, and large layouts have 15 items. For each size layout there was a base shelving layout that was created. Then in each of those base layout locations for shopping list items was randomized. In total, there were 35 random layouts generated per size layout, with that data being averaged for the final numbers. The following images are the base layouts with examples of item locations that were randomly generated. I attempted to simulate reasonable supermarket layouts. I previously mentioned the color scheme for the GUI but I will mention it again here for legibility: Black squares are the shelves that can't be walked through, red squares are the squares that must be reached to grab items, the green square is the entrance of the store, and of course white squares are walk-able space.

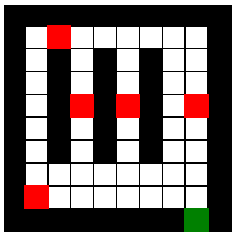


Figure 2: Small 10x10 Base Layout

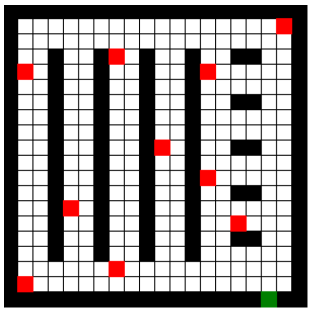


Figure 3: Medium 20x20 Base Layout

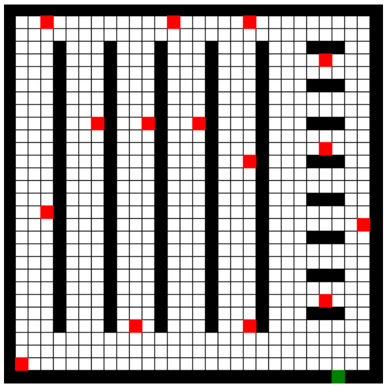
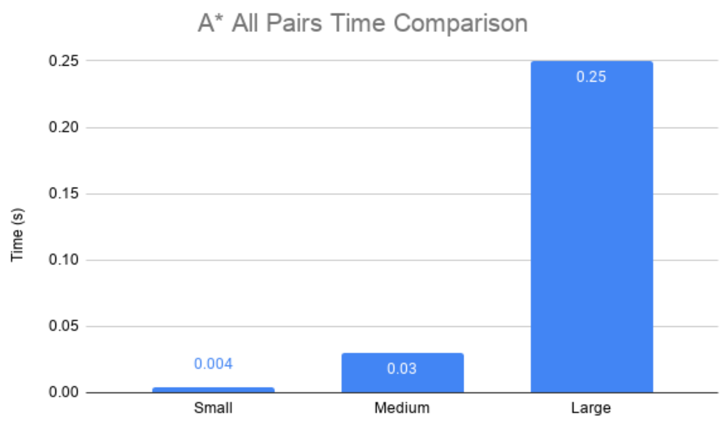
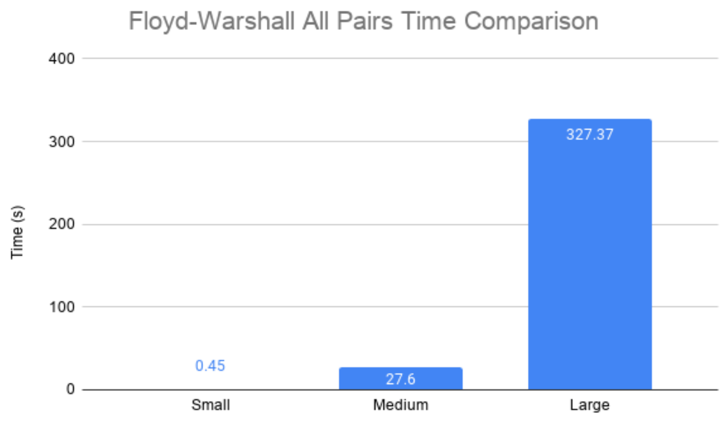


Figure 4: Large 30x30 Base Layout

5.0.1 All Pairs Shortest Path Solution Times

To set up the TSP I first have to find the shortest distance between all pairs of nodes. To do this I tested two different methods: Floyd-Warshall's Algorithm (FWA) and A*. For each size of graph, as shown above, I recorded the time it took to complete each algorithm on 35 randomized item node sets. The averaged results are shown in the below charts.

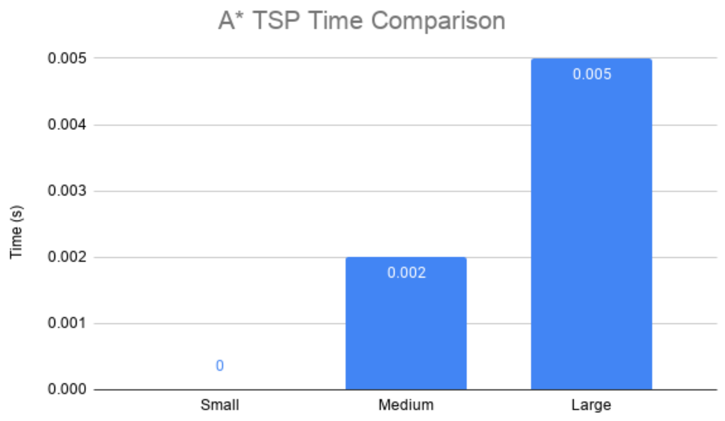


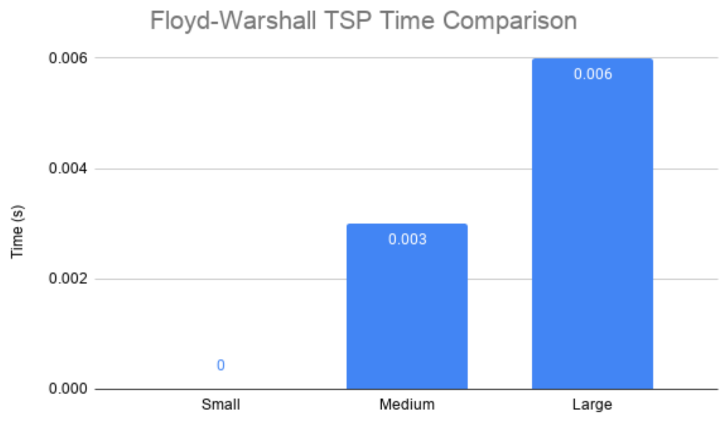


The time each algorithm takes increases exponentially, rather than linearly, with size. While not surprising, the results are a bit unfortunate because the more cells there are in the graph the better the representation of the store you have, and the better the representation of the store the more useful the pathfinding. However there is a silver lining: the scale of each figure. The difference in scale is most emphasized by the time difference between the large graphs. For the largest size tested, 30x30 cells, the time difference between A* and FWA is a factor of 1000, with A* taking 0.25 seconds on average and FWA taking 327 seconds. This is quite a significant disparity. At the smallest size of graphs, 10x10 cells, there is still a wide margin between times, however for a human this difference is meaningless except in situations where the algorithm was repeated numerous times. This time difference arises from the logic of each algorithm. FWA explores *every* path when finding the best one, while A* uses a heuristic to assume which paths it needs to explore.

5.0.2 TSP Solution Times

The times recorded for solving the TSP are much less interesting, with a linear correlation between size and time.





For timing I used the built in Python package Time. I then recorded the time at which the function started and the time at which the function ended, and recorded the difference. For smaller scale problems the Greedy Heuristic method used for solving this TSP was so fast that the difference in time was recorded as 0. Obviously this literally can not be the case, however the time taken is clearly minuscule. The reason for the similarity between graphs is due to the similarity in implementation for solving the TSP. Two methods had to be made to interact with the different data structures used by each algorithm, but the general process was the same. While it is possible that the data structures used by A* are ever so slightly faster than the ones in FWA, I find this unlikely. I believe that with further tests, perhaps hundreds or thousands instead of 35, the time difference between solving the TSP with different structures would disappear.

5.0.3 TSP Solution Differences

The TSP process is nearly identical for for the FWA case and the A* case; different data structures were used for each algorithm, so the calls in each actual function had to match but the process did not change. Here I present some example solutions for each size of graph.

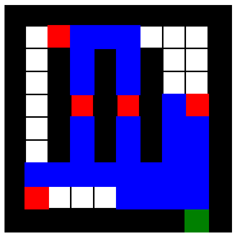


Figure 5: Small A* Solution

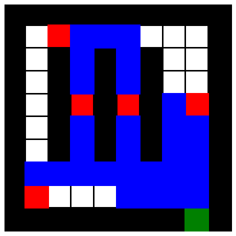


Figure 6: Small Floyd-Warshall Solution

For the small graph, the algorithms always returned the exact same path and path length. This is probably owed to the fact that at such a small size there are very few paths available to choose from.

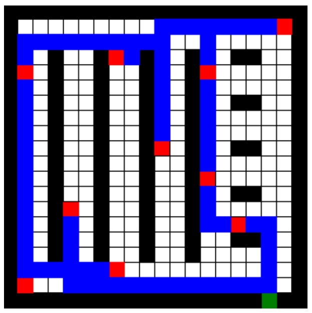


Figure 7: Medium A* Solution

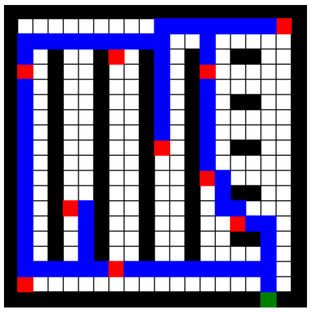


Figure 8: Medium Floyd-Warshall Solution

At the medium size graph is where the differences begin. Again, the path length is the same for each case, which is a good sign for the A* algorithm. The difference is the actual path taken. While the path is sometimes slightly different, only barely so as visualized in the above example graphs.

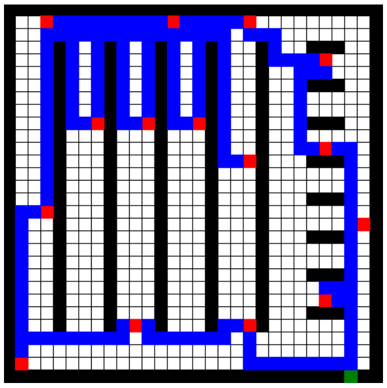


Figure 9: Large A* Solution

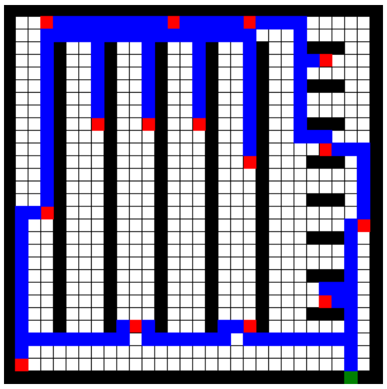
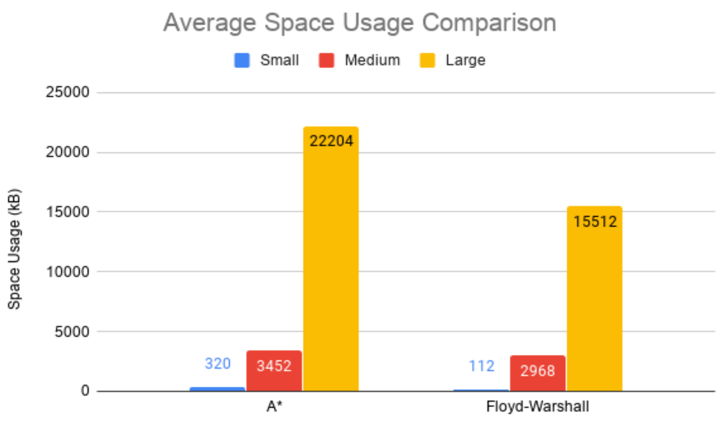


Figure 10: Large Floyd-Warshall Solution

Finally, for the large graph the path length is again the same for each case. The exact path taken does differ sometimes, but only barely so as visualized in the above example graphs.

5.1 Memory Usage

Many years ago memory usage and time might have been held on more equal ground. However, as computer architecture improves memory usage become decreasingly important. That's not to say that it isn't important to have a space efficient program, just that it doesn't make as big of an impact. That being said, here are the results comparing peak memory usage for each algorithm.



The peak memory usage occurs when storing all shortest paths between pairs. At first this disparity doesn't seem so bad, but then realize that FWA stores *every* shortest distance and path to *every* cell in adjacency matrices, while the A* algorithm uses a linked node model to store paths and distances to only the item nodes to be visited. When viewed in that light, the memory usage of A* as implemented here is enormous compared to FWA.

6 Conclusion

The application I have created is usable, but quite clunky. With more time I would have made it even more user friendly with cleaner and clearer paths between nodes, a way to permanently save a graph you have created, and a way to load saved graphs. If these features were added this application would creep closer towards useful in the real world versus purely useful as a project and for analysis. In terms of the algorithms, the results produced here seem to lean in favor of A*, but there are also benefits to using Floyd-Warshall's Algorithm (FWA). FWA takes more time to find every single path but uses less memory, and that could actually be a strength. Imagine a situation where the layout of a store is unchanging, but the item locations might change. Running FWA once gives all the paths, so every TSP is forever set up with distances for that layout. The efficiency of FWA can even be increased lending even more credibility to its use [8]. A* on the other hand, has the advantage of speed because of its specialization towards finding only the necessary paths, but its implementation here made large use of memory. From a human perspective, creating Nodes and connecting them made coding with A* quite easy, since I didn't have to go back and forth between tables and matrices. For efficiency sake though, if the A* used here were redesigned to use the structure of FWA, it would demonstrably lower the memory usage. Purely based on the application on display, A* is the way to go for finding all shortest paths between pairs.

References

[1] H. A. Abdulkarim and I. F. Alshammari. Comparison of algorithms for solving traveling salesman problem. *International Journal of Engineering and Advanced Technology*, 4(6):76–79, 2015.

[2] N. Auger, V. Jugé, C. Nicaud, and C. Pivoteau. On the Worst-Case Complexity of TimSort. In Y. Azar, H. Bast, and G. Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] J. C. Dela Cruz, G. V. Magwili, J. P. E. Mundo, G. P. B. Gregorio, M. L. L. Lamoca, and J. A. Villaseñor. Items-mapping and route optimization in a grocery store using dijkstra's, bellman-ford and floyd-warshall algorithms. In *2016 IEEE Region 10 Conference (TENCON)*, pages 243–246, 2016.

[4] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):373–556, 1969.

[5] M. Gen, R. Cheng, and D. Wang. Genetic algorithms for solving shortest path problems. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation*, pages 401–406, 1997.

[6] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10):2245–2269, 1965.

[7] K. Magzhan and H. M. Jani. A review and evaluations of shortest path algorithms. *International Journal of Scientific & Technology Research*, 2(6):99–104, 2013.

[8] D. Wei. An optimized floyd algorithm for the shortest path problem. *Journal of Networks*, 5(12):1496–1504, 2010.

[9] J. Y. Yen. Finding the lengths of all shortest paths in n-node nonnegative-distance complete networks using n^3 additions and n^3 comparisons. *Journal of the Association for Computing Machinery*, 19(3):423–424, 1972.

[10] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.