ALGORITHM OVERVIEW

The following is a pseudocode representation of Dijkstra’s algorithm which was used for the application’s solution to this problem.

graph G

unvisited → empty

visited → empty

dist → empty

dist[first] = 0

for vertices in G:

add vertex to unvisited

visited.push(unvisited.pop(0))

nextNode = 0

while unvisited is not empty:

min = infinity

nextNode = next shortest distance in unvisited

// remove from unvisited

current = unvisited.pop(nextNode)

// add to visited

visited.push(current)

// update distances in remaining nodes in unvisited

For vertices in G:

If vertex in unvisited:

dist[vertex] = distance from current

return visited

This application exhibits an overall space complexity of O(n) and an overall time complexity of O(n). This is because the number of stored items grows linearly as packages and vertices are added. The algorithm itself has a O(ElogV) derived from the inside loop occurring on a smaller and smaller set of vertices as the while statement continues. All of this happens after a loop through the vertices to initialize them. Other areas of note include the storing and retrieval of data from the hash table. The storing of the data has O(1) time complexity and the retrieval has O(1). The hash table itself has space complexity of O(n) as it consistently grows with each new entry. Most other operations are O(1) time complexity.

The beauty of using this implementation of Dijkstra’s algorithm is that it scales pretty well. There will be a noticeable increase in computation time once the amount of routes is increased a significant amount, but that should, in theory, not get too large as a truck can only hold so many packages at a time. This will keep the best solution always relatively quick to solve as the number of possible routes should stay as low as the number of packages.

This software is efficient in the way that it attacks the problem by breaking up the problem into pieces and solving the pieces without unnecessary steps. The application also uses these pieces as guidelines for reusable sections of code that are organized and structured. This allows for code reusability, ease of feature addition, and the ability to quickly debug the application. These attributes make the software more maintainable whether in the case of adding features because of market changes or debugging a small bug during development.

The data structure housing the package data is a bucket hash table. This data structure is beneficial because it handles hash collisions gracefully by placing them in ‘buckets’ that can be traversed quickly to retrieve values. A weakness of this type of hash table occurs if you choose an incorrect number of buckets at the start or if the number of buckets chosen is too small and the problem scope scales too large for the table to handle. If there are too few buckets, then you essentially spend an inordinate amount of time traversing the buckets looking for the value rather than having spread out the values as is the intention of this data structure. Another possible choice for a data structure here would be a variation of the hash table with a different collision mitigation technique. One such choice could be a hash table with quadratic probing. The benefit of using this technique would be that it uses a clearly defined equation to traverse the hash locations to avoid a collision. This equation is then used to retrieve the stored data as well and this would result in an efficient data structure for both insertion and lookup. As for another possibility, a set or dictionary could be a good option here as it requires distinct entries and the packages are all distinct. The benefit of this data structure over a hash table would be that the use of keys could allow for fast retrieval of package information when the key is known. The downfall is that when the key is not known, it may take much longer to retrieve the data than a hash table.