***Core Algorithm Overview***

**Problem description**

We are tasked with designing an algorithm to solve the Vehicle Routing Problem, a generalization of the Traveling Salesman Problem in computer science. The problem requires that we find optimal routes for a set of vehicles that will deliver packages to locations. The problem is NP-complete, so an exact solution is impractical. Rather, we are tasked with finding an approximate solution that is practical to implement and meets the needs of our client.

Our version of the Vehicle Routing Problem includes several constraints. There are two vehicles. Vehicles can hold only 16 of the 40 packages we must deliver. Some packages can only be carried by a particular vehicle. Different packages have different time deadlines by which they must be delivered. Some packages are “delayed” and cannot leave the vehicle hub until certain times. We must assume that management can change the delivery deadline for any package at any time. All of these constraints increase the complexity of the problem.

**Algorithm description**

I wrote two algorithms that can be used to produce low-mileage routes to deliver the packages.

*Algorithm 1: Swapping for local improvement*

I first implemented a greedy hill-climbing algorithm. I randomize the routes, represented by arrays containing delivery addresses. I then loop through the arrays and check each pair of addresses to see if exchanging their positions would reduce the total miles a vehicle would need to travel to go to every address in each array. I ensure exchanges are true improvements by comparing the distances of shortest paths between addresses, found using Dijkstra’s algorithm with an indexed priority queue. By looking for marginal improvements relative to an initial condition, the algorithm converges to a local optimum where no single exchange can improve mileage. Pseudocode for the algorithm is in Figure 1.

I implemented practical efficiency optimizations to improve the average-case performance of the algorithm. First, the algorithm stops once it converges to a local optimum (“early stopping”), which nearly always occurs within ten iterations of the algorithm’s outer loop. Additionally, rather than calculate the total mileage for all routes (arrays) when checking the value of each exchange, I calculate only the change in mileage for the routes that would be affected. Finally, I halve the number of comparisons that occur by ensuring that I compare each address pair only once during each pass through the arrays.

*Figure 1: Swapping algorithm pseudocode*

For each pass through the arrays:

For each address in each array:

For each other address in each array:

If swapping address positions reduces mileage:

Swap address positions

Importantly, the algorithm is unlikely to find the global optimum the first time it converges to any optimum. Rather, it reaches a point where two or more exchanges are necessary to reach a better condition. This is a common issue in numeric optimization algorithms found in machine learning and econometrics. A simple solution to the local optima problem is to apply an algorithm multiple times with different initial conditions. I repeat my algorithm multiple times, each time randomizing the initial positions of the addresses within and between the arrays, and keep only the best result.

The algorithm has multiple strengths. First, it can approximate a global optimum fairly quickly. It also doesn’t require users to manually distribute packages between vehicles. With additional programming, I was able to account for some types of package constraints. For example, I can enforce the requirement that a package remain in a specific route and the vehicle capacity limit.

Unfortunately, the algorithm cannot incorporate requests for changes to routes after they have been formed. If management needed to change a package delivery deadline while a driver was mid-route, the algorithm could not provide a practical solution without additional modifications. The algorithm otherwise meets all of the project requirements.

*Algorithm 2: Nearest neighbor*

To address the drawback of the first algorithm, I implemented a second algorithm based on the “nearest neighbor” principle. Given a set of addresses, the algorithm selects the nearest neighbor as measured by shortest path distance. Assuming a vehicle travels to that nearest neighbor location, the algorithm then picks the nearest neighbor from the remaining set of addresses the vehicle intends to visit. This is repeated so that a route is ordered by relative shortest paths.

As before, shortest paths are calculated using an all-pairs Dikjstra’s algorithm. Dikjstra’s algorithm is based on the insight that the shortest path to a vertex **w** in a graph is the sum of the shortest path to an adjacent vertex **v** and the shortest path between **v** and **w**. Starting from a single source vertex, this insight can be inductively applied to find a shortest path from the source vertex to any other vertex in the graph so long as such a path exists. Pseudocode for Dijkstra’s algorithm is in Figure 2.

I implemented Dijkstra’s algorithm using an index priority queue and an edge-weighted graph data structure. I implemented the index priority queue using an array representing a heap-ordered binary tree. The priority queue is sorted by the weights of graph edges, and is used to determine the order in which to review graph vertices. This implementation is more efficient than a list-based implementation, allowing each graph edge to be reviewed only once. I used the textbook Algorithms 4e (Sedgewick & Wayne, 2011) as a reference source.

This second algorithm is flexible. It can be used to plan an entire route, or used in an ad-hoc manner to find the shortest path between two addresses. At any point, a vehicle can change course to meet a deadline. The algorithm meets all of the requirements in the project description. Its primary shortcoming is the need to manually assign packages to vehicles.

*Figure 2: Dijkstra’s algorithm pseudocode*

Vertex s = source vertex

distto[] = array of known distances from s to other vertices

edgeto[] = array of edge used to reach each vertex on shortest path

While priority\_queue is not empty:

Relax(graph, priority\_queue.remove\_minimum())

Function Relax(Graph g, Vertex v):

For each edge adjacent to v:

w = vertex on end of edge opposite v

if distto[w] is greater than distto[v] + edge weight:

distto[w] = distto[v] + edge weight

edgeto[w] = edge

if priority\_queue.contains(w):

change priority of w because we found a shorter path to it

else:

add w to priority\_queue

**Data structures**

The graph data structure is implemented as a list of adjacency lists. Each adjacency list corresponds to a vertex in the graph (a delivery address). Each adjacency list contains an entry for each directed edge starting at its corresponding vertex. There are two edges for each pair of vertices, because all vertices are connected and each vertex in a pair has a directed edge object in its adjacency list corresponding to every other vertex. The edges are implemented as objects with weights that correspond to the distances between vertices (in miles). The core operations implemented for the data structure take constant time in the worst case.

Alternatively, I could have implemented the graph as a V-by-V matrix of edge weights, where a non-null edge weight represents a connection between vertices. Such an implementation would allow for a constant-time check for a connection between any two vertices. Because we assume all vertices are connected, that operation is not necessary. The matrix graph has additional overhead in a key operation: it returns an iterable data structure containing edges adjacent to a given vertex in linear time, while the array of adjacency lists can return it in constant time. Use of the matrix implementation would increase the time complexity of Dijkstra’s algorithm by a multiplicative factor that is linear in the number of vertices, V.

A third way to implement that graph data structure would be to use a flat array of edges. Such an implementation is impractical because it requires examining every edge in the graph to return a list of edges adjacent to a specified vertex. Its use would increase the time complexity of Dijkstra’s algorithm by a multiplicative factor that is quadratic in V (linear in the number of edges).

I implemented an index priority queue for Dijkstra’s algorithm that uses a heap-ordered binary tree to implement core operations in logarithmic time. A heap-ordered binary tree is a binary tree that guarantees each parent node has greater weight (higher priority) than either of its children. It is similar to a sorted binary tree, but with a weaker invariant. The tree is balanced and implemented in an array. The array-based implementation makes index-based operations easy, so that the priority queue functions like an array that can efficiently output element with the least priority. In the context of this assignment, the priority queue was implemented to store vertices and weight their priority by their distance from a source vertex.

Other package data, used for general programming purposes, is stored in a dictionary implemented using a hash table with separate chaining. The separate chains are implemented as linked lists, and are stored in a python list. The hash table is designed to automatically resize itself to maintain constant time performance for core operations.

**Programming model and space-time complexity**

The worst-case time complexity of the all-pairs Dikjstra’s algorithm implementation is O(), where *E* is the number of graph edges and *V* is the number of graph vertices. Because there is an edge between every vertex in our graph, this can be written O(). The single-source Dijkstra’s algorithm looks at every edge in the graph and either changes the priority of a vertex or adds a vertex to a priority queue. The relevant priority queue operations are completed in worst case time of O(), so the single-source Dijkstra’s algorithm has a worst case time complexity of O(). The all-pairs version of Dijkstra’s algorithm simply applies the single-source version once for every vertex.

The main loop of the swapping algorithm described above has worst-case time complexity of O(), where *V* is the number of delivery addresses. The algorithm first calculates and saves an all-pairs Dijkstra’s object. Further operations on the object—i.e. learning shortest path distances between two vertices—are constant-time operations. Next a set of arrays containing addresses—representing routes—are shuffled in linear time. The algorithm loops through delivery addresses in a nested fashion, leading to comparisons. Each comparison has a cost of *4C*, where *C* is the capacity of a vehicle, due to the way a swap is evaluated in the program. The full *0.5\*4\** operations repeat for a fixed number of iterations or until convergence to a local optima, usually within ten iterations. The time complexity of the entire process is:

Here, *R* is a user-selected number of times to apply the local algorithm in search of the global optimum, while *I* is a constant number of iterations used to find a local optimum. Though a user could select any values for *R* and *I*, realistic choices of *R* and *I* are 30 and 15, respectively. Selecting greater values of *R* can increase the likelihood of approximating a global optimum, but values of *R* as low as 5 or 10 are effective in practice. Increasing the value of *I* above 15 will not improve outcomes except in exceptionally rare cases.

The worst case time complexity (equation 2) is dominated by the cost of all-pairs Dijkstra’s algorithm. However, it is important to consider that time complexity refers to the rate of growth in run time. As the number of packages delivered increases, the use of Dijkstra’s algorithm will increase in importance. When the number of delivery addresses is small, the real-world run time of the algorithm is dominated by the second term of equation 1 because the constant terms *R*, *I*, and *C* are relevant.

The entire algorithm uses additional space proportional to . All-pairs Dijkstra’s algorithm uses additional space proportional to . The graph uses additional space proportional to , which in our context is .

The time complexity of the second algorithm I implemented is dominated by Dijkstra’s algorithm in the worst case, and also when *V* is small. The additional components of the algorithm are minimal because the algorithm depends on the user to manually select the stops associated with each route. The algorithm essentially sorts of the array based on relative (i.e. non-constant) weights. My simple implementation is quadratic in the size of a route. Where Y is the number of routes, it has a time complexity of:

The worst case time complexity is described by equation 2. The algorithm uses no additional data structures beyond those used in the all-pairs Dijkstra’s algorithm, so its space complexity is proportional to .

**Scalability, efficiency, and maintainability**

Both algorithms I implemented are scalable in practice. Dijkstra’s algorithm, which dominates the time complexity of the algorithms, has a cubic growth function. This suggests the algorithm may not be scalable to large operations consisting of tens or hundreds of thousands of packages. However, the algorithm can be computed comfortably from a standard personal computer when the number of locations to visit numbers in the hundreds or low thousands for each hub.

The algorithm code is written modularly and generically in order to be expandable and maintainable, but the overall software would benefit from practical “quality of life” improvements. For example, the data is currently stored in CSV files. The data could be cheaply and efficiently stored in a SQL database with which the software could communicate. The software would also benefit from a user interface with functions that allowed users to do basic tasks like adding and removing location and package data. The code I implemented provides a set of useful API’s that can be used as part of a broader application.

**Alternative algorithm**

As the client’s delivery business increases in size, I recommend further consideration of classes of algorithms based on meta-heuristics such as Tabu Search, Simulated Annealing, and Evolutionary Programming. These meta-heuristics lead to classes of scalable algorithms that can produce near-optimal solutions to the Vehicle Routing Problem. The algorithms find near-global optima by using probabilistic techniques to avoid settling on local optima. Although the technical details of these algorithms are beyond the scope of this course, it is worth noting that entire books have been written about the use of these algorithms for the Vehicle Routing Problem.

**Reflection**

If I were to do this project again, I would implemented a meta-heuristic algorithm such as simulated annealing, Tabu-search, or evolutionary optimization. These meta-heuristic algorithms produce state of the art solutions to the Vehicle Routing Problem. Although these algorithms are beyond the scope of this course, I believe I have sufficient background in statistics to learn how to implement them from a good textbook. I did not implement one of these algorithms because of the time investment required to read an additional textbook.