**Core Algorithm Overview**

**Stated Problem:**

This project uses **Dijkstra’s Shortest Path** algorithm to determine the most efficient method of routing packages for delivery. It will intake and parse a list of packages and attributes as well as a list delivery addresses (vertices) with distances (weight) to each other delivery address. The delivery addresses are stored in memory using a **weighted graph** object model. Using this data, the program will run the core algorithm to determine which package is the next closest delivery, moving from vertex to vertex, running the algorithm at each stop to determine the next shortest path. We track the weight of each traversed edge in order to output total miles driven by each truck once all packages have been delivered.

**Algorithm Overview:**

**Dijkstra’s Algorithm**, created in 1956 by computer scientist Edsger Dijkstra( [Dijkstra, E. W.](https://en.wikipedia.org/wiki/Edsger_W._Dijkstra) (1959). ["A note on two problems in connexion with graphs"](http://www-m3.ma.tum.de/twiki/pub/MN0506/WebHome/dijkstra.pdf)), is a variant of the **greedy algorithm**. This algorithm chooses whatever method is the next most efficient, but not always the most efficient path overall. It works in this way:

* A **weighted graph** is created with weighted edges connecting each vertex.
* For each vertex, the algorithm will determine that vertex’s total distance from the starting vertex, as well as it’s predecessor vertex in the path back to the start.
* All vertices distances are initialized to infinity, all predecessors are initialized to 0 or null, and all vertices are placed in an unvisited queue.
* The start vertex is assigned a weight of 0.
* The algorithm pops the start vertex from the queue, then checks the weight between the start vertex and each adjacent vertex distance.
* If the weight is less than the current distance to the adjacent vertex (infinity to start), the adjacent vertex distance is assigned the current weight value, and the predecessor vertex is assigned the start vertex.
* Once all adjacent vertices have been checked, the shortest distance is popped from the queue and its adjacent vertices are checked.
* If a shorter path is found from the start vertex, the adjacent vertex is updated with the new distance and predecessor vertex.
* The algorithm continues until the queue is empty, at which time, the distance of each vertex is now the shortest distance to the start vertex.
* In order to find the route, we work backward from the destination vertex to the start vertex using the stored predecessor vertex.

**Pseudocode:**

for vertex in graph:

set vertex distance = infinity

set predecessor vertex = null

create list ‘unvisited vertices’

place all vertices in unvisited vertices list

set start vertex distance = 0 # distance from start vertex

while unvisited vertices list is not empty:

set next shortest distance to first vertex in list

for vertex in unvisited vertices list:

if vertex distance < next shortest distance:

next shortest distance = vertex

current vertex = pop next shortest distance from list

for each adjacent vertex to current vertex:

distance = to distance between adjacent and current vertex

alternate distance = current vertex distance + distance

if alternate distance < adjacent vertex distance:

adjacent vertex distance = alternate distance

adjacent vertex predecessor vertex = current vertex

**Efficiency:**

The outer loop loops once for each vertex in the list, or O(V). For each loop, each vertex is looped over again, for O(V). For each adjacent vertex, each edge is checked once, for E times.

O(V\*V+E) = O(V^2+E) = O(V^2)

**Adaptability and Maintenance:**

The core algorithm is coded in a manner that allows a list of any size. Because it runs in O(V^2), as the number of vertices grows, so will the run time. A solution to this issue would be implementing the unvisited vertices data structure as a heap rather than a list. This data structure could reduce runtime from O(V^2) to O(V log V). Implementing this change does not change the core functioning of the algorithm, allowing ease of maintenance.