WasteApp - Selective Garbage Collection

Miguel Azevedo Lopes
Rafael Fernando Ribeiro Camelo
Sofia Ariana Moutinho Coimbra Germer

April 2021

Contents

1. Introduction

1.1 Problem Statement

Development of a waste management app, integrated in a smart network that includes containers equipped with sensors that provide an updated status on the capacity of the container (how much trash can it still take). The containers are mapped. This feature allows users to not only know where the nearest trash container is, but also if it can take more trash or not.
Additionally, this app introduces a brand new service, trash pick up directly at home. This service is done by normal citizens who want to have an extra source of revenue. Thus, to provide this service the app needs to have both an interface for the regular users who want to have their trash collected and for users who wish to collect that trash and take it to the nearest recycling facility. In order to make this service as efficient as possible, the app provides an optimized route for the trash collectors hence reducing travel costs as well as the time spent on the road.

1.2 Iterative Analysis

We considered the route creation the main problem of this project and therefore that was the focus of our iterative analysis.

- Iteration 1 - One car with unlimited capacity picks up trash with an optimized route
- Iteration 2 - One car with limited capacity picks up trash with an optimized route
- Iteration 3 - Multiple cars with limited capacity pick up trash with an optimized route

2. Problem Formalization

2.1 Input Data

2.1.1 Map

The Map is a weighted (the weight is given by the distance between nodes) directed graph G = (N, E) composed of Nodes and Edges.

2.1.2 Node

N - Set of Nodes in the graph
Represent specific points such as Trash Containers, Houses or Waste Management Facilities

- **adj** - Adjacent edges to the Node
- **la** - Latitude
- **lo** Longitude

2.1.3 Edge

E - Set of Edges in the graph
Represent the roads that connect Nodes.

- **w**: Weight of an edge - The distance between two Nodes (Edge's weight)
- **dest**: Destination - Pointer to the node it leads to

2.1.4 Trash Containers $\in$ Node

Tc - Set of Trash Containers, where Tc(i) is the -ith element
Trash containers are selective storage sites for residual waste. They are a *Node* in our *Map* and have information about:

- **TcM**: Maximum Capacity - the maximum amount of waste it can hold.

- **TcC**: Current Capacity - the current amount of trash the container has, given by sensors in the container
- **Tct** : Trash Type

2.1.5 House $\in$ Node

H - Set of Houses, where House(i) is the -ith element
Houses is where the users of our app live. They are a *Node* in our *Map* and have information about:

- User - a pointer for the *User* who lives in that house
- Pick Up - a flag for when a *User* needs his trash to be collected.
- Amount Trash - how much trash the *User* has that needs to be collected.

2.1.6 Garbage Collection Facility $\in$ Node

G - Set of Garbage Collection Facilitys, where G(i) is the -ith element

- Name

2.1.7 User

U - Set of Users, where U(i) is the -ith element

- IDUser

2.1.8 Driver $\in$ User

D - Set of Drivers, where Driver(i) is the -ith element

- **Dc**: Pointer to the *Car* that the Driver uses to collect trash
- **Dm**: Amount of money the driver has earned for his collections

2.1.9 Car

C - Set of Cars, where Car(i) is the -ith element

- **Cc**:The capacity in liters that a car can take. Used to calculate how many trash bags it can take.
- **Cf**: The amount in liters that is already filled

2.2 Output Data

- A Graph (Map), with a set of Nodes (Trash Containers) that allow a person to know where the nearest Trash Container that's not full is located.
- Various trash collection trips, T1,T2,...,Tn (n is the number of trips). Trips are defined by a Driver, a ordered set of Nodes the driver must go to and the set of Edges he should use to get to those Nodes.

2.3 Restrictions

2.3.1 Input Data Restrictions

- N
  - $\forall n \in N$, adj(n) $\subseteq$E
  - $\forall n \in N$, la(n) >= $0^\circ \wedge$ la(n) <= 180°
  - $\forall n \in N$, lo(n) >= $0^\circ \wedge$ lo(n) <= 180°
- E
  - $\forall e \in E$, w(e) > 0, as the weight of each edge represents the distance.
  - $\forall e \in E$, dest(e) $\in N$, as each edge points to a node.
- Tc
  - Tc $\in N$

- $\forall t \in Tc$, $TcM(t) > 0$, because a cars container maximum capacity has to be greater than 0.

- $\forall t \in Tc$, $TcC(t) < TcM(t)$, because a trash container current capacity can never be greater than it's maximum capacity.

- $\forall t \in Tc$, $TcT(t) = PAPER \vee GLASS \vee PLASTIC \vee REGULAR$, because the type type of waste can only be one of these: paper, plastic,glass, regular

- C

  - $\forall c \in C$, $Cc(c) > 0$, because a trash container maximum capacity has to be greater than 0.

2.3.2 Output Data Restrictions

- Gf = (Nf, Ef)

  - Nf is a subset of N, so the rules that apply to N also apply to Nf

  - Ef is a substet of E, so the rules that apply to E also apply to Ef

- Df

  - $\forall d \in D$, $Dm(d) >= 0$, the driver can't lose money on the app, so the driver's balance needs to be 0 or positive.

2.4 Objective Functions

When tackling the problem at hand, we considered two main objectives:

- Given a location, defining a function to return the position of the closest Trash Container that still has free space

- A function that give us a path that visits all houses that want their trash collected with the minimum distance possible, in order to reduce the costs associated with travel. If f is a function of distance, our goal is to minimize it:

$$f = \sum_{i=1}^{n} d_i$$

d represents the distance between Houses
n is the number of Houses to visit

3. Solution

3.1 Pre-Processing of Input Data

In order to improve the temporal and spacial efficiency of our program we decided to pre-process our input data, more precisely, our graph.

- Remove all edges that aren't accessible (In case of construction in a street).
- When we have more than one edge with the same origin and destiny, choosing the edge with smaller weight and discarding the other (In the real world this means if we have several ways to get form A to B we choose the way that has a smaller distance).

3.2 Graph connectivity analysis

A directed graph is called strongly connected if there is a path in each direction between each pair of vertices of the graph and it is applied only on a directed graph.
As we mentioned above, our graph is directed, so if there is an edge (u,v) it doesn't necessarily mean that there is an edge (v, u) as well, due to the existence of one-way roads.
To optimize our solution we will ignore nodes of the graph that are inaccessible (sometimes roads are closed). In order to this, we will analyse the graph's connectivity.

3.2.1 Brute Force

One simple way to think about the graph's connectivity is Brute Force. This strategy would execute the following steps:

- **Step 1**: Find the shortest path between all pairs using one of the algorithms we will analyse later on.
- **Step 2**: Check if the distance is infinite between any two pairs (infinite distance means the node is unreachable), except for self loops

- If true : The component is not SCC

- Else : The component is SCC

Complexity Analysis: $O(|V|^3)$

3.2.2 Kosaraju's algorithm[1]

The Kosaraju's algorithm consists of three main steps:

- **Step 1**: Perform DFS traversal and push nodes to the stack before returning

- **Step 2**: Find the transposed graph by reversing the edges

  - Note: On reversing all edges of the graph, the type of graph won't change , so the SCC will remain as SCC

- **Step 3**: Pop nodes one by one from stack and repeat a DFS on the modified graph (Each successful DFS gives one SCC).

Complexity Analysis

- Time Complexity

  - The DFS over the Graph and the transposed Graph takes $O(|V| + |E|)$

  - The Transpose operation also takes $O(|V| + |E|)$

  - So, the total time complexity of this algorithm is $O(|V| + |E|)$

- Space Complexity

  - The Stack used takes $O(|V|)$

Algorithm 1 Kosaraju's algorithm

Main function

```
1: S = ∅
2: L = Stack()
3: function Kosaraju(G(N, E))
4:     for u ∈ N do SCC(u) ← NULL
5:     end for
6:     for u ∈ N do DFS_K(G, u)
7:     end for
8:     while !L.Empty() do
9:         u ← L.Pop()
10:        Assign(G, u, u)
11:    end while
12:    return SCC
13: end function
```

Auxiliar function 1

```
1: function DFS_K(G(N, E), u)
2:     if u ∈ S then return
3:     end if
4:     S ← S ∪ {u}
5:     for v ∈ Adj(G, u) do DFS_K(G, v)
6:     end for
7:     L.Push(u)
8: end function
```

Auxiliar function 2

```
1: function Assign(G(N, E), u, root)
2:     if SCC(u) ≠ NULL then return
3:     end if
4:     SCC(u) ← root
5:     for v ∈ Adj(G^T, u) do Assign(G, v, root)
6:     end for
7: end function
```

3.2.3 Tarjan's algorithm[2]

The Tarjan's algorithm is based on the following steps:

- **Step 1**: DFS search produces a DFS tree/forest
- **Step 2**: Strongly Connected Components form subtrees of the DFS tree.
- **Step 3**: If we can find the head of such subtrees, we can print/store all the nodes in that subtree (including head) and that will be one SCC.
- **Step 4**: There is no back edge from one SCC to another (There can be cross edges, but cross edges will not be used while processing the graph).

Algorithm 2 Tarjan's algorithm

input: graph G = (N, E)
output: set of strongly connected components (sets of vertices)
Main function

```
1: function TARJAN(G(N, E))
2:     for u ∈ N do
3:         id(u) ← NULL
4:         SCC(u) ← NULL
5:     end for
6:     for u ∈ N do
7:         if id(u) = NULL then
8:             DFS_T(G, u)
9:         end if
10:    end for
11:    return SCC
12: end function
```

Auxiliar function 1

```
1: nid ← 1, L ← STACK()
2: function DFS_T(G(N, E), u)
3:     L.PUSH(u)
4:     id(u) ← nid + +
5:     low(u) ← id(u)
6:     for v ∈ ADJ(G, u) do
7:         if id(v) = NULL then
8:             DFS_T(G, v)
9:             low(u) ← min{low(u), low(v)}
10:        else if v ∈ L then
11:            low(u) ← min{low(u), id(v)}
12:        end if
13:    end for
14:    if low(u) = id(u) then
15:        while (v ← L.POP()) ≠ u do SCC(v) ← u
16:        end while
17:        SCC(u) ← u
18:    end if
19: end function
```

Complexity Analysis

- Time Complexity

  - The DFS over the Graph and the transposed Graph takes $O(|V| + |E|)$

  - The Transpose operation also takes $O(|V| + |E|)$

  - So, the total time complexity of this algorithm is $O(|V| + |E|)$

- Space Complexity

  - The Stack used takes $O(|V|)$

3.2.4 Conclusion

Although Brute Force is a very simple approach of solving this problem, we won't consider the possibility of implementing this algorithm, due to its low efficiency.
In this way, we will consider the Kosaraju's and the Tarjan's algorithm. Both the methods have the same linear time complexity, but the techniques or the procedure for the SCC computations are fairly different. Tarjan's method solely depends on the record of nodes in a DFS to partition the graph whereas Kosaraju's method performs the two DFS (or 3 DFS if we want to leave the original graph unchanged) on the graph.
So, for now, we think the best solution is the Tarnjan algortihm since it only does one DFS.

3.3 Pre-calculation of the minimal path between two nodes

The first option we considered to solve the problem of determining the path between two nodes was using either the Dijkstra or the Floyd-Warshall's algorithm to pre-calculate the minimum path between all pairs of nodes at the program's start, hence removing the need to calculate those values afterwards.
With that in mind, it was also clear from the start that when we consider

a graph with the dimension of a city, the computation time of this pre-calculations could be quite high. Since time complexity is very important in a graph of significant dimensions, we don't plan to use this option however we plan to test them in a later stage of the project and review our choice.

3.3.1 Dijkstra's algorithm[3]

Dijkstra's algorithm has many variations, originally it found the shortest path between two $Nodes$ of a given $Graph$. The variant we will be using fixes one $Node$ as the 'origin' and creates a shortest path graph based on the distance between the origin and every other node.

The fundamental idea is that, at each call of the algorithm, we have a set of $Nodes$ that have not been processed yet, then using a minimum priority queue, Q, sorted by the distance between the current $Node$, and the origin, we go through the queue extracting the closest $Node$, cn, and updating the distance travelled so far by the weight of the connecting edge between the current $Node$ and the next one in Q. The algorithm ends when we find the target or when Q is empty which means that there isn't any available path for that combination of origin $\rightarrow target$.

Algorithm 3 Dijkstra's algorithm

```
1: function DIJKSTRA(G(N, E), origin)
2:     Q ← ∅                                   ▷ Creating an empty Priority Queue
3:     for node ∈ N do                                          ▷ Initialization
4:         dist(node) ← ∞
5:         prev(node) ← NULL
6:         Q.insert(node)
7:     end for
8:     dist(origin) ← 0
9:     while |Q| > 0 do                                          ▷ Main cycle
10:        cn ← Node of Q with least dist(cn)
11:        Q ← Q\{cn}
12:        for adjn ∈ ADJ(G, cn) do
13:            if dist(adjn) > dist(cn) + w(cn, adjn) then
14:                dist(adjn) ← dist(cn) + w(cn, adjn)
15:                prev(adjn) ← cn
16:                Q.update_Priority(adjn, dist(adjn))
17:            end if
18:        end for
19:    end while
20:    return dist, prev
21: end function
```

3.3.2 Floyd-Warshall algorithm [4]

The Floyd-Warshall algorithm was published in its currently recognized form by Robert Floyd in 1962. This algorithm is a simple and widely used algorithm to compute shortest paths between all pairs of vertices in an edge weighted directed graph.

Algorithm 4 Floyd-Warshall algorithm

```
1: function FloydWarshall(G(N, E, w))
2:     for i, j ∈ N do                                   ▷ Initializations
3:         dist(i, j)^(0) ← ∞
4:         for k ∈ N do prev(i, j)^(k) ← NULL
5:         end for
6:     end for
7:     for i ∈ N do dist(i, i)^(0) ← 0
8:     end for
9:     for (i, j) ∈ E do                                 ▷ Add edges
10:        dist(i, j)^(0) ← w(i, j)
11:        prev(i, j)^(j) ← i
12:    end for
13:    for 1 ≤ k ≤ |N| do                                ▷ Main cycle
14:        for 1 ≤ i, j ≤ |N| do
15:            c' ← dist(i, k)^(k−1) + dist(k, j)^(k−1)
16:            if c' < dist(i, j)^(k) then
17:                dist(i, j)^(k) ← c'
18:                prev(i, j)^(k) ← prev(k, j)^(k)
19:            end if
20:        end for
21:    end for
22:    return dist^(|N|)
23: end function
```

Complexity analysis

The comparisons made by the Floyd-Warshall algorithm are done with $O(|V|^3)$. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal. It performs better in denser graphs and Dijkstra's algorithm can produce the same result in time $O(|V|(|E| + |V|) \log |V|)$, or $O(|V|^2 \log |V|)$ if the graph is sparse.

3.3.3 Conclusion

After reviewing both algorithms we came to the conclusion that, since our graph is likely to be sparse, the Dijkstra's algorithm would prove more efficient to do the pre-calculations.

3.4 Minimal path between two nodes

Since pre-calculation is not a viable option, the next logical step is to calculate the minimal path between two nodes whenever it is necessary.
There's a number of different algorithms to do this calculation, however we decided to focus once again on Dijkstra's algorithm and the A* algorithm because we think they're the ones that best suit this particular case.

3.4.1 Dijkstra's algorithm [3]

Of all the algorithms we could consider to solve this problem Dijkstra's stands out for it's ease of implementation. With a few modifications to the algorithm presented earlier we can make the algorithm stop once it finds the destination node.
Despite the advantages stated before, Dijkstra's algorithm could prove highly inefficient in highly populated graphs if the distance between the two nodes is significant.

Complexity analysis

Previously analyzed. View 3.2.1

3.4.2 A* algorithm[5]

A* was created as part of the Shakey project, which had the aim of building a mobile robot that could plan its own actions.It's an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (in this case least distance travelled)
At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of

the cheapest path from n to the goal (Euclidean distance for instances). It terminates when the path it chooses to extend is a path from start to goal or if there are no paths eligible to be extended.

Algorithm 5 A* algorithm

```
1: function ASTAR(Start, Goal)
2:     openSet ← {Start}
3:     cameFrom ← empty map
4:     costMap ← map initialized with ∞
5:     costMap[0] ← 0
6:     bestCostMap ← map initialized with ∞
7:     bestCostMap[0] ← H(Start)
8:     while costMap is not empty do
9:         current ← the node in costMap having the lowest bestCostMap[] value
10:        if current = goal then
11:            return path obtained by backtracking over locations in cameFrom
12:        end if
13:        openSet.Remove(current)
14:        for each neighbor of current do
15:            tentative ← costMap[current] + d(current, neighbour)
16:            if tentative < costMap[neighbour] then
17:                cameFrom[neighbour] ← current
18:                costMap[neighbour] ← tentative
19:                bestCostMap[neighbour] ← costMap[neighbour] + H(neighbour)
20:                if neighbour not in costMap then
21:                    openSet.add(neighbor)
22:                end if
23:            end if
24:        end for
25:    end while
26:    return failure - goal never reached
27: end function
28:
29: function H(n) ▷ Calculates the Euclidean distance between the node n and the goal node
30:    x ← (n.longitude − goal.longitude)^2
31:    y ← (n.latitude − goal.latitude)^2
32:    return √(x + y)
33: end function
```

Complexity analysis

The space complexity of A* is roughly the same as that of all other graph search algorithms, as it keeps all generated nodes in memory. Time complexity varies according to the heuristic function used.

3.4.3 Conclusion

When comparing Dijkstra's algorithm to A* we must consider that Dijkstra's is in fact a special case for A* where the heuristic is zero. A* search only expands a node if it seems promising. It only focuses to reach the goal node from the current node, not to reach every other nodes. It is optimal, if the heuristic function is admissible. So if the heuristic function is good at approximating the future cost, then it will need to explore a lot less nodes than Dijkstra which could prove positive in our particular case (a lot of houses that can potentially have a considerable distance between them).
For this reason as well as all the reasons previously stated, we decided to go with the A* algorithm, with Euclidean Distance as the heuristic function.

3.5 Shortest path passing through multiple nodes

After determining the best way to compute the distance between two nodes, the next main issue at hand is to create a route that passes through all the nodes (houses) we want to visit. When researching about this issue we encountered the famous Travelling Salesman Problem.
The travelling salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?", which is very similar to the problem we are facing, "Given a list of houses and distances between each pair of houses, what is the shortest possible route that visits each house exactly once and ends in a Waste Management Central ?".

3.5.1 Exact solution versus approximate solutions

When tackling this problem there are two possible approaches, the ones that return exact solutions and the ones that provide us with approximations. Naturally, we considered the exact solutions first. The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute-force search). The running time for this approach lies within $O(n!)$. Confronted with this values we looked for more sophisticated solutions however even the time complexity for best solutions (Held–Karp algorithm) were still very high at $O(n^2 2^n)$. Normally this would prove problematic, however let's revise our proposed iterations:

- **Iteration 1** - One car with unlimited capacity picks up trash with an optimized route:
  Considering a car with unlimited capacity, if the number of houses to visit were high the time complexity of running a brute force search for the best option would be very high ($O(n!)$, the factorial of the number of houses, so this solution becomes impractical even for only 20 houses). In this case we would have to consider approximate solutions.

- **Iteration 2** - One car with limited capacity picks up trash with an optimized route:
  When we consider a car with limited capacity the analysis changes. In Portugal last year the Renault CLIO was the car with the most sales so we'll use it as an example. Its trunk has 284 liters in capacity which means it can take up to 9 trash bags with a 30 liter capacity . In the worst case scenario, each house has only 1 trash bag to be collected which means the driver will have to go to 9 houses before going to a central to dispose of the trash bags. This means that regardless of the number of houses the driver has to visit, the algorithm that creates the route will only have to consider 9 houses when it does its calculations, after which the algorithm runs again with the waste management central as the starting point.

- **Iteration 3** - Multiple cars with limited capacity pick up trash with an optimized route:
  No changes here. If anything, more cars means each driver has to visit less houses.

At this point we started analysing the Held-Karp algorithm, however during this analysis we neglected a decisive factor. The Held-Karp algorithm optimizes a route given a set of houses, however we still needed an algorithm to create that set of houses.

We then considered a cluster approach (analysis below), only to once again conclude that it wasn't the best solution for our problem.

In the end we decided that simplicity was the best option and because of that our ultimate choice is the Nearest Neighbour algorithm, which eliminates the need for the "set of houses" mentioned earlier. This algorithm picks the nearest house available and once the car is full of trash, finds the closest Garbage Collection Facility, amongst other details which we explain in its own section.

Despite our algorithm choice and considering we went thought the trouble of researching the Held-Karp algorithm, we decided to keep it in the report.

3.5.2 Cluster analysis

The vehicle routing problem (VRP) is a combinatorial optimization that generalizes the TSP problem.
As we are considering the existence of several drivers and that the vehicles have a limited capacity we can't solve a simple TSP,since we must split our graph in different trips. This problem can be solve in different ways and we analysed different possibilities to choose our routes.
As we were searching for different methods for doing our clustering, we realized that this would be difficult to implement since our VRP has a lot of restrictions : the vehicles have limited capacity (CVRP), there are multiple depots (MDVRP), the vehicles can do more than one trip (VRPMT) and the fleet of vehicles is heterogeneous (HVRP).
For example, the **sweep algorithm** is a cluster-first route-second algorithm. We assume that each vertex i is represented by its polar coordinates (θ_i, ρ_i), where θ_i is the angle and ρ_i is the ray length. It executes the following steps:

- **Step 1**: Route Initialization. Choose an unused vehicle k.
- **Step 2**: Route Construction. Starting from the unrouted vertex having the smallest angle, assign vertices to the vehicle k as long as its capacity

or the maximal route length is not exceeded. If there remain nodes that remain unrouted vertices go to step 1.

- **Step 3**: Repeat from Step 1 if there are nodes left to visit
- **Step 4**: Route Optimization. Optimize each vehicle route separately by solving the corresponding TSP (we would execute the Held Karp).

The problem with this algorithm is that it has been created to solve a homogeneous VRP, so if this approach is chosen there hast be some minimal changes, as choosing always the car with smaller capacity possible (in order to optimize the numbers of cars used).
However there are no guarantees that using the sweep algorithm we will obtain a viable solution : even if the capacity of all cars together is bigger than the capacity needed to collect all the garbage, it is possible that the combination of houses (routes) turns out on an impossible situation.
We considered using other algorithm, for examples : **Fisher and Kaikumar** (This solution isn't adequate since it demands a fixed number of vehicles and that value isn't fixed in our app since there can be always new users).
All the possible solutions that we have found would compromise some restriction of our "real -world" problem so we concluded that, for now, this isn't the best approach.

3.5.3 Held-Karp algorithm[7]

The Held-Karp algorithm is a dynamic programming approach to problems similar to the TSP.
The costs of the final solution, the distance, can be modeled as the sum of individual steps and is denoted as $D(V, v)$.
For a set $S \subset V$ that contains v and a node $w \in S \backslash \{v\}$ let $D(S, w)$ be the costs of the shortest path that starts at the origin point, visits all locations in S and ends at location w. The shortest path of this subproblem consists of two parts: the last arc on the path that goes from some location $u \in S$ to w, and a shortest path that starts at the origin point v, visits all locations in $S \backslash \{w\}$, and ends in u.
If the path for the full set S does not contain the shortest subpath for $S \backslash \{w\}$, we can obtain a better solution by replacing this subpath with the shortest one, contradicting the fact that the path for S was shortest.
As a consequence, the subproblem $D(S, w)$ can be solved by considering all

arcs (u, w): $u \in S \backslash \{w\}$, and adding each of those arcs to the solution for the smaller subproblem $D(S, w)$. This structure, in which the optimal solution of a subproblem can be expressed in terms of smaller subproblems can be formalized by means of a recursive formula as follows:

$$\begin{cases} \infty & \text{if } w \notin S \\ c(v, w) & \text{if } S = \{w\} \\ \min_{u \in\ S} D(S \backslash \{w\}, u) + c(u, w) & \text{otherwise} \end{cases}$$

As there 2^n possible subsets of N and n locations in n, we can see that there are at most $n * 2^n$ subproblems to solve here. For each subproblem in the recurrence relation we only consider at most n smaller subproblems, and as a result this algorithm can be run in $O(n^2 * 2^n)$ time. A possible implementation of an algorithm that exploits this structure in a bottom up fashion is presented in Algorithm 5.

Algorithm 6 Held-Karp algorithm

Data **:** Set of locations N, an arbitrary location $n \in N$ and a cost function c

Result **:** The shortest path between all locations in N

```
1: Initialize D with values ∞
2: Initialize a table P to retain predecessor locations
3: Initialize n as an arbitrary location in N
4: for w ∈ V do
5:     D({w}, w) ← c(v, w)
6:     P({w}, w) ← v
7: end for
8: for i = 2, ..., |N| do
9:     for S ⊆ N where |S| = i do
10:        for w ∈ S do
11:            for u ∈ S do
12:                z ← D(S\{w}, u) + c(u, w)
13:                if z < D(S, w) then
14:                    D(S, w) ← z
15:                    P(S, w) ← u
16:                end if
17:            end for
18:        end for
19:    end for
20: end for
21: return path obtained by backtracking over locations in P starting at
    P(N, n)
```

3.5.4 Nearest Neighbour[8]

This is a very simple to understand algorithm. We go through a set of predetermined nodes and in each one we calculate the closest node to it that has not been visited thus far. In our version of this algorithm, if the car is full we add a stop at the nearest Garbage Collection Facility. Nearest neighbour is a greedy approach to this problem and isn't guaranteed to find the shortest path, however it provides a good approximation.

Pseudocode

Algorithm 7 Nearest Neighbour

```
1: function NEAREST NEIGHBOUR(G(N, E), NodesToGoBy)
2:     Result ← ∅                          ▷ vector of Nodes to go through in order
3:     N ← NodesToGoBy[0]
4:     Result.add(N)
5:     NodesToGoBy.remove(N)
6:     while |NodesToGoBy| > 0 do
7:         Astar(G, N)
8:         NodesToGoBy.Sort()    ▷ Sort the Nodes by their dist attribute
9:         N ← NodesToGoBy[0]
10:        if Cf < N.Trash then          ▷ Cf is the capacity left in the car
11:            N ← ClosestGarbageFacility()
12:            Result.add(N)
13:            Cf ← Cc        ▷ Cf is reset after the car goes to the Garbage
   Collection Facility
14:            continue
15:        end if
16:        Result.add(N)
17:        NodesToGoBy.remove(N)
18:    end while
19:    return Result
20: end function
```

Efficiency

Focusing on Time Efficiency: Given the Pseudocode above uses a $Sort()$ algorithm, let $O(S)$ be the time complexity for it, and $A*$'s algorithm as well, let the time complexity for it be $O(A)$, we can conclude the time complexity for Nearest Neighbour is $O(|N| * S * A)$ where $|N|$ is the size of the vector of $NodesToGoBy$.
Focusing on Space Efficiency: In this algorithm we use a simple container for the Nodes, therefore the space efficiency amounts to $|NodesToGoBy|$

3.6 Additional Functions

In this section we will analyse secondary problems, namely, finding the closest trash container that isn't full **Closest Trash Container** and a function that determines how the drivers get paid **Payment Calculator**.

3.6.1 Closest Trash Container

This algorithm first reduces the number of trash containers to analyse by establishing a radius, R, around the Node that defines a person's location. Let's say R=5km, the algorithm will use Euclidean distance to create a priority queue of trash containers that have a Euclidean distance that is smaller than 5km, from the user's current location.
The priority queue orders the nodes base on distance (from smallest to biggest), using the previously defined algorithm (A*). It then chooses the first trash container of the queue, that is now the closest one, and checks whether or not it is full. If it's full it checks the next element of the set, and so on. If it's not full it returns that node.

Algorithm 8 Closest Trash Container

```
1: function ClosestTrash(G(N, E), CurrentNode,Radius)
2:     max ← ∞
3:     for each n in G do
4:         if Euclidean(CurrentNode, n)<= Radius then
5:             ClosestTrashQueue.Add(n)
6:         end if
7:     end for
8:     while ClosestTrashQueue not empty do
9:         if ClosestTrashQueue.Front().IsEmpty() then
10:            ClosestTrashQueue.Pop()
11:            continue
12:        end if
13:        return ClosestTrashQueue.Front()
14:    end while
15: end function
```

3.6.2 Payment Calculator

The payment calculator will determine how much a driver gets paid. It is a simple algorithm that considers two factors, the number of houses a driver visits, and the amount of kilometers he had to travel.

Algorithm 9 Payment Calculator

1: **function** PAYMENTCALCULATOR(TotalDistance,NumberHouses)
2: $HousePay \leftarrow PayPerHouse \times NumberHouses$
3: $DistancePay \leftarrow PayPerKm \times TotalDistance$
4: $TotalPay \leftarrow HousePay + DistancePay$
5: **return** $TotalPay$
6: **end function**

4. Use Cases

4.1 Identification

The Application developed will contain a simple text based interface built with menus to interact with the user. It will be divided into two main interfaces, dependent on which user is going to use the App, the $User$ interface, UI, and the $Driver$ interface, DI.
It will also allow to save information about a Map ($G(N, E)$), which will contain $Trash\,Containers$ and $Garbage\,collection\,facilities$ in predetermined locations. Essentially all functionalities of our application will revolve around operating on the Map.

4.2 Functionalities

- Viewing the Map
- Inputting User's Houses locations.
- Shortest path between two or multiple locations.
- Checking if a location is reachable.
- Attributing efficiently a path of Houses to pick up trash from for the Driver ending in a garbage collection facility.
- Showing the expected pay for a given pick up route.
- Giving a set of houses which trash fits in a given car.

5. Conclusion

To end this report we decided to review our work and then formulate our conclusions.
We started by analysing the problem statement and formulating an iterative analysis. This analysis proved to be crucial as it helped structure the way we thought about this problem and all its conundrums. After this we worked on the problem formalization where we defined both the input and output data as well as all the restrictions that we thought of at this stage of the project. Finally before starting to solve the problems at hand we stated our objective functions. All these initial steps helped us reach our ultimate solution which we will now review.
While developing this report, we faced several moments we needed to make choices in order to optimize our solution. We had to research several different algorithms and understand every possible approach in order to choose the best one. From determining that Tarjan's algorithm was the best option for pre-processing of the graph, to having to choose between pre-calculation of the distance between nodes and immediate calculations using the A* algorithm and finally settling for the Nearest Neighbour algorithm after an extensive research process where we considered Held-Karp, and several different options for doing the clustering of the graph.
This was a challenging project, however we feel that it was the preparation we needed for the next part of the project.

6. Contribution

Miguel Azevedo Lopes - Objective Functions, Floyd-Warshall algorithm, Minimal path between two nodes (A* algorithm), Exact solution versus approximate solutions,Held-Karp algorithm, Nearest Neighbour, Conclusion, Additional Functions
Rafael Fernando Ribeiro Camelo - Dijkstra's algorithm, Nearest Neighbour, Use Cases
Sofia Ariana Moutinho Coimbra Germer - Restrictions , Pre-Processing of Input Data, Graph connectivity analysis(Brute Force, Kosaraju's algorithm, Tarjan's algorith), Exact solution versus approximate solutions, Cluster analysis, Conclusion, Additional Functions
All - Problem Statement, Iterative Analysis, Input Data, Output Data

Bibliography

[1] Kosaraju's algorithm:
https://iq.opengenus.org/kosarajus-algorithm-for-strongly-connected-components/
https://www.geeksforgeeks.org/strongly-connected-components/
https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm

[2] Tarjan's algorithm:
https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm
https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/

[3] Dijkstra's algorithm:
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html
https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[4] Floyd-Warshall algorithm:
https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

[5] A* algorithm:
https://en.wikipedia.org/wiki/A*_search_algorithm
https://www.geeksforgeeks.org/a-search-algorithm/
http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html

[6] Clustering
https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68
https://neo.lcc.uma.es/vrp/bibliography-on-vrp/RHG93

[7] Held-Karp algorithm:
https://developers.google.com/optimization/reference/graph/hamiltonian_path/
https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

[8] Nearest Neighbour:
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm
https://en.wikipedia.org/wiki/Travelling_salesman_problem