WasteApp - Selective Garbage Collection

Miguel Azevedo Lopes - up201704590
Rafael Fernando Ribeiro Camelo - up201907729
Sofia Ariana Moutinho Coimbra Germer - up201907461

May 2021

Contents

1. Introduction

We were tasked with the 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.
In this second report we will discuss how we went about developing the App, the data structures we used, what algorithms we implemented and their complexity and finally we will also review the connectivity of our Graph.

2. Used Data Structures

2.1 Graph portrayal

The graph portrayal was implemented based on the *Graph* class, which can be found in the Graph.h file. The *Graph* class has a single attribute, a vector of *Vertex* pointers called *VertexSet* (`vector<Vertex *>` VertexSet). It also has a variety of different methods, which we use to manipulate the *VertexSet*.

2.2 Auxiliar Data Structures

As well know, data structures influence significantly the code's time efficiency. In this way, while developing our app we were faced with several scenarios where we need to choose the best data structure for what we pretended.

2.2.1 Pre-Processing

- **Stack**: In Tarjan's Algorithm and Kosaraju's Algorithm we used a stack to store all nodes that were visited
- **Unordered-Set**: In Kosaraju's Algorithm we used an unordered-set to store all visited nodes, and search for them.

  - An unordered-set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized.

  - All operations on the unordered-set takes constant time **O(1)** on an average which can go up to linear time **O(n)** in worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

2.2.2 Routing

- **Mutable Priority Queue**: We used the Mutable Priority Queue provided in class to improve the time efficiency of our Dijkstra's and A* algorithms

- **Map**: We used a map to store where each vertex came from, when creating a path
- **Stack, Queue and Vector**: We also used stacks and queues for path construction and to transverse paths

2.2.3 Other

- **Vector**: Vectors were our main choice for storing information such as the Users, Cars, Drivers, Garbage Collection Facilities, etc. Some of these elements had a pointer to a graph Vertex to situate them in the map, others have just their x and y coordinates to situate them.

3. Implemented Algorithms

3.1 Pre-Processing

Both the algorithms were implemented as planned in the first report, however we will also review them here. They allowed us to characterize the given maps in terms of their connectivity, identifying how many strongly connected components there were, and most importantly allowing us to use the other algorithms we planned.

Tests to Pre-Processing Algorithms

If the user selects the Programmer option in the Menu, he can choose to run some tests to check if our Pre-Processing Algorithms are working correctly. We chose to create this functionality since these are relatively complex algorithms that weren't lectured in classes. By creating this option we can prove that our algorithms are correct (we created a small graph and tested it by comparing it to the expected results).

Notes:

While developing this algorithms, we made the decision about how to store the results : we chose to store the graph's multiple strongly connected components in a 2 dimensional vector (by storing the index of it's nodes in each subvector). This way, every position of the main vector is a strongly connected component and we chose the biggest one for our app.
It is also important to refer that in the Programmer Option shows how many SCCs are in the original Graph (we thought that this functionality was very interesting since it gives us a better understanding of the graph's connectivity).

3.1.1 Tarjan's Algorithm

Tarjan's pseudocode

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

3.1.2 Kosaraju's Algorithm

Kosarajus's pseudocode

Algorithm 2 Kosaraju's algorithm

Main function

```
 1: $S = \emptyset$
 2: $L = $ STACK()
 3: function KOSARAJU($G(N, E)$)
 4:     for $u \in N$ do $SCC(u) \leftarrow$ NULL
 5:     end for
 6:     for $u \in N$ do DFS_K($G$, $u$)
 7:     end for
 8:     while !$L$.EMPTY() do
 9:         $u \leftarrow L$.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 $u \in S$ then return
3:     end if
4:     $S \leftarrow S \cup \{u\}$
5:     for $v \in$ 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 $SCC(u) \neq$ NULL then return
3:     end if
4:     $SCC(u) \leftarrow root$
5:     for $v \in$ ADJ($G^T, u$) do ASSIGN($G$, $v$, $root$)
6:     end for
7: end function
```

3.1.3 Complexity

Theoretical:

- Tarjan:
  - Temporal: $O(|V| + |E|)$
  - Spacial: $O(|V|)$
- : Kosaraju:
  - Temporal: $O(|V| + |E|)$
  - Spacial: $O(|V|)$

3.2 Dijkstra's and A*'s algorithms

In the first report we mentioned the possibility of pre-calculating the distance between all nodes, however it became clear soon enough that doing such an operation would involve a lot of waiting when running the program for the calculations to be done. For that reason we discarded that possibility and moved on to the two other possibilities we proposed in our first report using either Dijkstra's or A* to compute the distance between 2 nodes when necessary. We implemented both methods with a Mutable Priority Queue to speed up minimum value extractions. We arrived at the conclusion that A* was consistently faster, as we predicted in our first analysis.

3.2.1 Dijkstra's pseudocode

Algorithm 3 Dijkstra's algorithm

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

3.2.2 A*'s pseudocode

Algorithm 4 A* algorithm

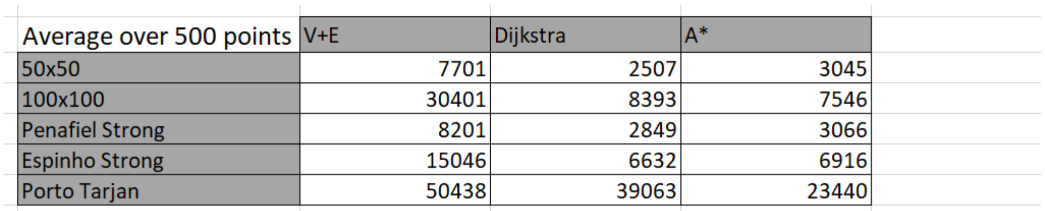
```
1: function ASTAR($Start, Goal$)
2:     $openSet \leftarrow \{Start\}$
3:     $cameFrom \leftarrow$ empty map
4:     $costMap \leftarrow$ map initialized with $\infty$
5:     $costMap[0] \leftarrow 0$
6:     $bestCostMap \leftarrow$ map initialized with $\infty$
7:     $bestCostMap[0] \leftarrow H(Start)$
8:     while $costMap$ is not empty do
9:         $current \leftarrow$ 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:            $tentative \leftarrow costMap[current] + d(current, neighbour)$
16:            if $tentative < costMap[neighbour]$ then
17:                $cameFrom[neighbour] \leftarrow current$
18:                $costMap[neighbour] \leftarrow tentative$
19:                $bestCostMap[neighbour] \leftarrow costMap[neighbour] + H(neighbour)$
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:    $x \leftarrow (n.longitude - goal.longitude)^2$
31:    $y \leftarrow (n.latitude - goal.latitude)^2$
32:    return $\sqrt{x + y}$
33: end function
```

3.2.3 Complexity

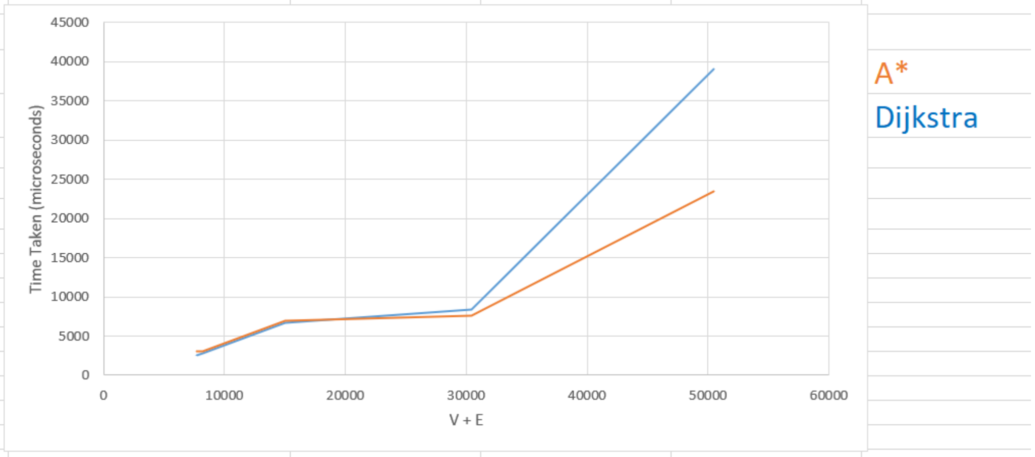
Theoretical: The space and time complexity of A* and Dijkstra's algorithm are very similar, partially because A* is simply a variation of Dijkstra's algorithm, or if you prefer Dijkstra's is the equivalent of an A* with an heuristic function where $h(x) = 0$.

- Temporal: $((V + E) * logE)$
- Spacial: $O(|V|)$

Empirical: To test the algorithms we ran them using 3 strongly connected components and 2 graphs created by us. The results are presented below:



Average over 500 points	V+E	Dijkstra	A*
50x50	7701	2507	3045
100x100	30401	8393	7546
Penafiel Strong	8201	2849	3066
Espinho Strong	15046	6632	6916
Porto Tarjan	50438	39063	23440



3.3 Nearest Neighbour

In our initial report we first considered and exact solution such as Held-Karp's algorithm, however we were dissuaded by it's time complexity and the need to also use clustering algorithms. For that reason, we ended up using the most common solution to this sort of problems - the Nearest Neighbour algorithm. This is a greedy approach that chooses at each point in the path creation process the node that is closest to it.

4. Menu

According to the functionalities that we aimed to construct, we divided the Menu int three options : The regular user option, the drivers option (both require a password) and the programmer option. This last one isn't exactly a part of the APP, but we thought it was important to include because this project is mainly focused on algorithms and optimization.

4.1 User's Menu

4.1.1 Log in

We ask the user to input their user ID and their password, if there is not any user with combination in our App we will ask the user to create an account.

4.1.2 Create account

We ask the user to input their name and their password, then we will create an User and save them to our App, telling the user what's their user ID.

4.1.3 Options

These are the options the user has once he is logged in:

Set House Address

We ask the user which location is going to be their house address (we ask his x,y coordinates and check if it is a vertex in the graph). Initially the amount of trash will be set to 0 and that indicates if the user want's his trash to be collected as false.

Search for closest trash container

Currently there are 4 trash options, Paper, Plastic, Glass and Regular. With the current user position we can tell the user which trash container is the

closest one for that specific trash type.
A graph will be shown with the users and the trash nodes colored.

4.1.4 Notes:

We didn't check if there was already a house in that node, because we thought that in one Apartment to different users can have the same coordinates and be represented by the same node but still be different Houses.
Since not every user has a House associated to him, when the program ends and we store the users data in files, we defined it's House ID as -1, which means that currently it has no House associated to him.

4.2 Driver's Menu

4.2.1 Log in

We ask the driver to input their user ID and their password, if there is not any driver with combination in our App we will ask the user to create an account.

4.2.2 Create account

We ask the driver to input their name, their password, their car trash capacity and its license plate, then we create a Driver and save them to our App, telling the user what's their user ID.

4.2.3 Options

These are the options the driver has once he is logged in:

Get pick up route

With the current driver position and car capacity we use an adaptation of the Nearest Neighbour algorithm that considers car capacity limitations when creating an optimized route that passes through the houses that want their trash collected.
The route will by shown using the GraphViewer.

Get route to facility

With the current driver position we will call the shortest path algorithm for the driver to get an efficient route that ends in the closest Garbage Collection Facility.
The route will by shown using the GraphViewer.

Check amount of money earned

Shows the driver the amount of money they have earned so far with the App

4.3 Programmer's Menu

4.3.1 Preprocessing Time Efficiency

Check Tarjan's time efficiency

Checks the time efficiency for the Tarjan's algorithm with a pre-built function

Check Kosaraju's time efficiency

Checks the time efficiency for the Kosaraju's algorithm with a pre-built function

4.3.2 Routing Time Efficiency

Check A*'s time efficiency

Checks the time efficiency for the A*'s algorithm with a pre-built function.

Check Dijkstra's time efficiency

Checks the time efficiency for the Dijkstra's algorithm with a pre-built function.

Compare Dijkstra's algorithm with A*'s algorithm

Checks the time efficiency for the A*'s and the Dijkstra's algorithm with a pre-built function and compares their values.

4.3.3 Test Algorithms

Test Tarjan's Algorithm

Tests the Tarjan's Algorithm and shows what the results should be.

Test Kosaraju's Algorithm

Tests the Kosaraju's Algorithm and shows what the results should be.

4.3.4 Show Graph

Shows entire Graph (through GraphView)

5. Conclusion

The previous problem analysis was very useful, since it allow us to structure our thought process and helped implement the methods since we already had the pseudo code.
One of the biggest difficulties over the course of the project was working with Graphviewer. Two of our members are mainly Windows users and unfortunately we couldn't get the Graphviewer to work there, only on Linux. We regret spending so much time focused on this part.
Despite this difficulties we felt this project was a great learning opportunity and we feel like we learned a lot about algorithms of all sorts, from clustering algorithms to preprocessing, path creation, etc.
To conclude, we would say this project was a success, considering we implemented all the algorithms we initially though we would need and they all worked as expected. We created an APP that fulfills all the requirements that were asked of us.

6. Contribution

Miguel Azevedo Lopes - 40%
Rafael Fernando Ribeiro Camelo - 20%
Sofia Ariana Moutinho Coimbra Germer - 40%