WasteApp - Selective Garbage Collection

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

April 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.

3. Implemented Algorithms

3.1 Tarjan's and Kosaraju's Algorithms

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.

3.1.1 Pseudocode

Tarjan's pseudocode

Algorithm 1 Tarjan's algorithm

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

Main function

```
function TARJAN(G(N, E))
    for u ∈ N do
        id(u) ← NULL
        SCC(u) ← NULL
    end for
    for u ∈ N do
        if id(u) = NULL then
            DFS_T(G, u)
        end if
    end for
    return SCC
end function
```

Auxiliar function 1

```
nid ← 1,  L ← STACK()
function DFS_T(G(N, E), u)
    L.PUSH(u)
    id(u) ← nid++
    low(u) ← id(u)
    for v ∈ ADJ(G, u) do
        if id(v) = NULL then
            DFS_T(G, v)
            low(u) ← min{low(u), low(v)}
        else if v ∈ L then
            low(u) ← min{low(u), id(v)}
        end if
    end for
    if low(u) = id(u) then
        while (v ← L.POP()) ≠ u do SCC(v) ← u
        end while
        SCC(u) ← u
    end if
end function
```

Kosaraju's pseudocode

Algorithm 2 Kosaraju's algorithm

Main function

```
S = ∅
L = Stack()
function Kosaraju(G(N, E))
    for u ∈ N do SCC(u) ← NULL
    end for
    for u ∈ N do DFS_K(G, u)
    end for
    while !L.Empty() do
        u ← L.Pop()
        Assign(G, u, u)
    end while
    return SCC
end function
```

Auxiliar function 1

```
function DFS_K(G(N, E), u)
    if u ∈ S then return
    end if
    S ← S ∪ {u}
    for v ∈ Adj(G, u) do DFS_K(G, v)
    end for
    L.Push(u)
end function
```

Auxiliar function 2

```
function Assign(G(N, E), u, root)
    if SCC(u) ≠ NULL then return
    end if
    SCC(u) ← root
    for v ∈ Adj(G^T, u) do Assign(G, v, root)
    end for
end function
```

3.1.2 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

```
function DIJKSTRA(G(N, E), origin)
    Q ← ∅                                        ▷ Creating an empty Priority Queue
    for node ∈ N do                              ▷ Initialization
        dist(node) ← ∞
        prev(node) ← NULL
        Q.insert(node)
    end for
    dist(origin) ← 0
    while |Q| > 0 do                             ▷ Main cycle
        cn ← Node of Q with least dist(cn)
        Q ← Q\{cn}
        for adjn ∈ ADJ(G, cn) do
            if dist(adjn) > dist(cn) + w(cn, adjn) then
                dist(adjn) ← dist(cn) + w(cn, adjn)
                prev(adjn) ← cn
                Q.update_Priority(adjn, dist(adjn))
            end if
        end for
    end while
    return dist, prev
end function
```

3.2.2 A*'s pseudocode

Algorithm 4 A* algorithm

```
function ASTAR(Start, Goal)
    openSet ← {Start}
    cameFrom ← empty map
    costMap ← map initialized with ∞
    costMap[0] ← 0
    bestCostMap ← map initialized with ∞
    bestCostMap[0] ← H(Start)
    while costMap is not empty do
        current ← the node in costMap having the lowest bestCostMap[] value
        if current = goal then
            return path obtained by backtracking over locations in cameFrom
        end if
        openSet.Remove(current)
        for each neighbor of current do
            tentative ← costMap[current] + d(current, neighbour)
            if tentative < costMap[neighbour] then
                cameFrom[neighbour] ← current
                costMap[neighbour] ← tentative
                bestCostMap[neighbour] ← costMap[neighbour] + H(neighbour)
                if neighbour not in costMap then
                    openSet.add(neighbor)
                end if
            end if
        end for
    end while
    return failure - goal never reached
end function

function H(n)  ▷ Calculates the Euclidean distance between the node n and the goal node
    x ← (n.longitude − goal.longitude)^2
    y ← (n.latitude − goal.latitude)^2
    return √(x + y)
end function
```

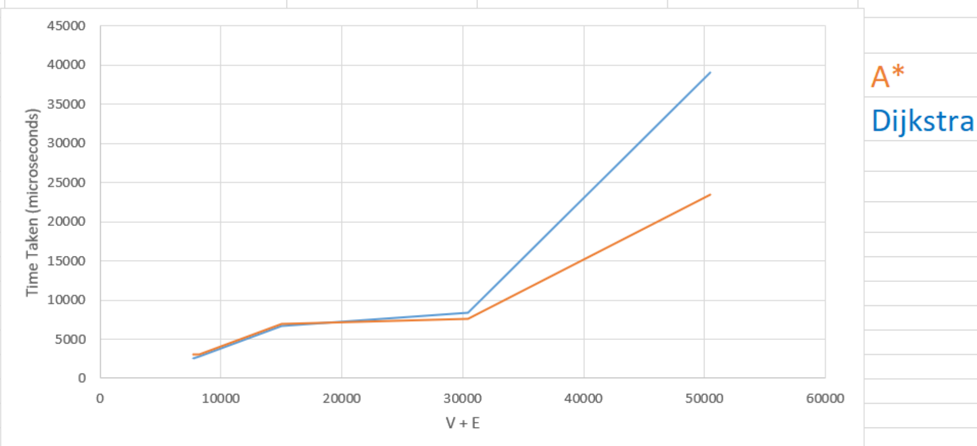
3.2.3 Complexity

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

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.

Set/Change House Address

We ask the user which location is going to be the new house address.

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.

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.

Set House Address

We ask the user which location is going to be their house address.

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.

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.

Get pick up route

With the current driver position and car capacity we will call the shortest path algorithm for the driver to get an efficient route that through the best houses considering their trash that is going to be picked up.

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 Garbage collection facility.

Check amount of money earned

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

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 will create a Driver and save them to our App, telling the user what's their user ID.

Get pick up route

With the current driver position and car capacity we will call the shortest path algorithm for the driver to get an efficient route that through the best houses considering their trash that is going to be picked up.

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 Garbage collection facility.

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.

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 pseudocode.

We encountered a lot of problems with the GraphViewe and it also brought some compatibility problems, because we were developing code in Linux and windows.

The interface is very simple and straight-forward, for the context we think that it should be enough and it also allows us to use almost every function implemented.

The Empirical analysis was not that extensive, only using 500 cases for the average time taken for each function which can me misleading we got very lucky (or unlucky). It also consumed a lot of time and computing power since, for the results to be more accurate the computer should remain in the same state from the begging to the end. Therefore our time was channelled to the parts of the project.

To sum up, in spite of our difficulties we managed to get it done, we implemented everything we wanted and we did it with a good code structure, we are proud our project.

6. Contribution

Miguel Azevedo Lopes - 40%
Rafael Fernando Ribeiro Camelo - 20%
Sofia Ariana Moutinho Coimbra Germer - 40%
All - Data Structures, Implemented Algorithms