Intelligent Garbage Collection

FEUP – MIEIC – CAL
Mário Gustavo Gomes Rosas de Azevedo Fernandes – up201201705@fe.up.pt
Bruno Miguel Faustino Moreno – up201504781@fe.up.pt
Francisco Maria Fernandes Machado Santos – up201607928@fe.up.pt

Abstract—**The main objective of this project is to develop solutions in order to deal with the problems of finding the best path to reach from point A to point B. The proposed theme is an automatic garbage collection system and its aim is to shorten the distance traveled by the vehicles. In order to do this, both Dijkstra's and Floyd Warshall's algorithm were used. The theme also included a lot of combinatory analysis which made finding the optimal solution very time consuming so the algorithms developed are greedy and only find good solutions.**

Keywords—graph, garages, containers, stations, dijkstra, floydwarshall, shortest path

I. Introduction

The background of the paper is a program that implements a garbage collection system that was implemented in a city. This system's objective is to shorten the travel distances between garages (start points) and waste treatment stations (finish points). The system only detects filled containers as points that a vehicle needs to go to to collect its garbage. It takes also into account the fact that there can exist multiple garages, stations, containers that need to be collected at the same time and have different garbage types, and vehicles (inside the same or different garages). The vehicles have limited capacity and can collect different types of garbage (one or many per vehicle).

The program represents all of this information within an abstract graph, which holds vertexes and edges. Each vertex holds location's information which generalizes garages, containers and stations. Each edge corresponds to streets between those vertexes. These also have weight which is the actual distance between two vertexes and can either be specified or automatically calculated using the coordinates of the source and destination vertexes.

The rest of the paper is structured as follows. Section II addresses the preliminary definitions that are used throughout the whole paper. Section III explains the program and the algorithm used and is subdivided into Section III a), Section III b) and Section III c) that explain the algorithm used and the use of Dijkstra's and Floyd Warshall's algorithm, respectively. Section IV explains the connectivity analyzation function and Section V explains the conclusions of the study involved and the program implemented. Finally, Section VI and VII show the use cases diagram and the class UML model, respectively.

II. Preliminaries

An external application was used in order to view the graph in a user-friendly way, and using this visualization, colors were attributed to symbolize each different location: garages are yellow, containers are either blue or red if they are full, stations are green and generic locations are blue; locations are circles and streets are black curved arrows.

The main definitions are:

- G represents the abstract graph which represents the network
- Vi represents a node with the index i on the graph
- Ei represents an edge with the index i on the graph
- A valid garage is a garage that has at least one vehicle capable of collecting garbage from an end of a filled containers set.
- An end of a vertexes set is one of the two vertexes which are the furthest away from any other vertex on that set (optimal start and exit)

A. Input data

The expected outcome is a set of nodes which correspond to an either optimal or good solution depending on the complexity of the graph at hand to collect the garbage of the filled containers. The path solution is easily interpreted by the user with the aid of the custom animation available each time the Collect Garbage option is selected, on the viewer application used.

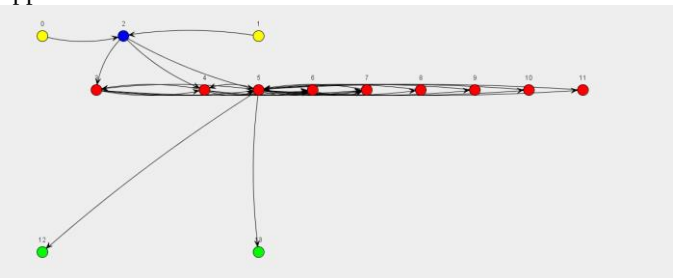

Fig1 – Example of a graph (used on statistics)

B. *Expected outcome*

The expected outcome is a set of nodes which correspond to an either optimal or good solution depending on the complexity of the graph at hand to collect the garbage of the filled containers. The path solution is easily interpreted by the user with the aid of the custom animation available each time the Collect Garbage option is selected, on the viewer application used.

C. *Objective function and restrictions used*

The aim is to minimize the travel distance, so a few restrictions were needed to be assumed before implementing the algorithm:

1. The distance between a garage or station and a set of containers is always considerably higher than the distance between each container.
2. A vehicle will always have considerably higher capacity than any container.
3. The graph is always not directed (apart from the garages and stations vertexes).

Using these restrictions, it can safely be assumed that reducing the number of trips and vehicles used will result in a shorter total distance traveled and the first container to go to should be an end of a vertex set.

III. Collect Garbage Program

The program's main classes are the Graph, Edge and Vertex ones, as well as the GarbageManagement which holds information about every variable of the program.

The superclass Location generalizes the Garage, Container and Station classes, and the graph used is a Graph<Location>. There also exists auxiliary classes such as Auxiliary, Menu and Interaction (which holds almost all of the communication with the user) and MyExceptions which holds custom declared exceptions which are used when an invalid input of the user is made.

An UML diagram is attached to this report which represent the classes in an easier to understand manner.

A. *Algorithm used*

The algorithm needs an integer input defining what path calculating algorithm to use (1 for Dijkstra's and 2 for Floyd Warshall's). It can be divided into 4 distinct parts:

This part simply sets up the algorithm by resetting all vehicles current capacity to 0 and checking which containers require collection.

In this part, a starting point is selected. This selection focuses on calculating the two ends of the container set by getting the sum of all of the shortest paths between each vertex: the two vertexes with the highest total distance are the ends. Once that is calculated, the best possible garage is calculated by checking which garage is closest to one of the ends and has a vehicle capable of collecting the garbage on that point. The algorithm chooses the best vehicle of that garage (a valid one and the one with the most capacity and garbage types), moves it to the starting point and starts a cycle to collect the garbage.

This part is the collection of the garbage itself, which for every new location the vehicle goes to, its current capacity is diminished due to garbage collection. This part follows a greedy algorithm which calculates the closest container in need of collection to the vehicle's current position. It then moves the vehicle to that point and repeats until there are no other points to go to apart from a station or the vehicle is full. This part could be improved by using a heuristic component which could take into account how full the vehicle is getting and comparing it to how further away from a station the vehicle is traveling in order to make a more optimal choice of the next container to go to.

This is the last part which, once a vehicle is filled or there are no containers left to pick up, the current vehicle is sent to the closest station to its current location. Finally, the full cycle (starting on part 2) is repeated until there are no more containers to collect from or there are no capable vehicles of picking up any more garbage.

This whole algorithm is a bit far from optimal since it can even show that the current containers that need collection are not possible to be collected when they're actually are. This may happen because vehicles that can collect multiple types of garbage can be led to pick up more of a certain type and then there will be no vehicle left to pick up the other types.

B. *Dijkstra's Algorithm*

Dijkstra's algorithm is an algorithm used to calculate the shortest distance between two nodes in a graph. It starts by setting every distance of every vertex to infinity except the starting vertex which is 0. The current vertex is inserted into a priority queue. The algorithm then gets the set of adjacent vertexes and iterates through them assigning their distance and putting them onto the queue. Once all of the adjacent vertexes are considered, the initial vertex is now "visited" and is extracted from the queue. If the target vertex has been reached, the algorithm is finished and the path is saved in each of the vertex, otherwise, repeat for the closest vertex (top of the queue).
In the program's version of the algorithm, a vertex is passed as an argument and for the whole graph, the algorithm defines the shortest path from that vertex to every other one.
The original algorithm (without queues) has a time complexity of O(|V|^2) and the program's algorithm's complexity is of O (|E|*log |V|) if |E| > |V| and a space complexity of O(V^2).

C. *Floyd Warshall's Algorithm*

Floyd Warshall's algorithm is an algorithm used to calculate the shortest distance between every node in a graph. It starts by creating two matrixes of the size of VertexSet^2 and filling them, one with the edge's weights and the other with -1. It then iterates through the matrixes and for each cell of a matrix, the value of the edge's weight is updated if a lower one to the current vertex is found. The second matrix simply holds the current vertex that it is being iterated.

In the program's version, this algorithm is accompanied by a custom getPath function (getfloydWarshallPath) which returns the shortest path between two vertexes.

This algorithm has a time complexity of O(|V|^3) and a spatial complexity of O(|V|^2) due to both matrixes.

D. *Algorithm chosen and Dijkstra vs Floyd Warshall*

A challenge that was encountered while making this project was choosing which shortest path algorithm to use. Apart from Dijkstra's and Floyd Warshall's, other algorithms were considered such as Bellman–Ford's which would be good if the graph had negative weights and even a minimum spanning tree algorithm such as Prim's or Kruskal's. The first one did not apply to this project since none of the edges are considered to have negative weights, and the later ones also did not apply since besides not needing to pass every location, it enforces a fixed path which vehicles that do not have corresponding garbage types might be uselessly traveling through.

Both Dijkstra's and Floyd Warshall's algorithms were used, giving the choice to the user of which he sees fit to use and also allowing an empiric comparison between each one of them.

Generally, the usage of both algorithms follow this rule: if (|E| ≥ |V|^ 2 / 2) Floyd Warshall's algorithm should be used, otherwise, Dijkstra's is faster. Although this is true, an adjustment should be made considering how many times the algorithm is called: Floyd Warshall's is called once on a single graph and never again; Dijkstra's algorithm needs to be called every single time a path needs to be calculated.

On the first time the garbage collection algorithm is run, unless (|E| ≥ |V|^ 2 / 2), Dijkstra's will always be faster. If the garbage collection algorithm is run a second time and (using dynamic programming) the matrix of the adjacencies is the same, the algorithm will be faster with Floyd Warshall's algorithm since the time complexity will only be O(n) (due to getfloydWarshallPath function).

All of the referred information has been tested using the "stress test" functionality of the program which runs the algorithms multiple times and prints the time it took to do each algorithm those numbers of times.

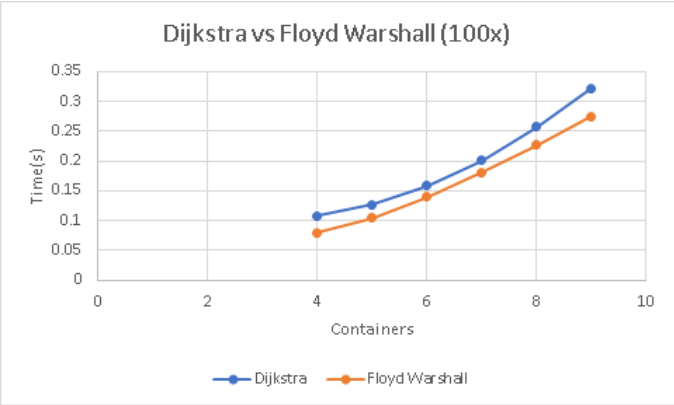

Fig2 – Graph comparing Dijkstra algorithm vs Floyd Warshall algorithm (stress test of 100 times each).

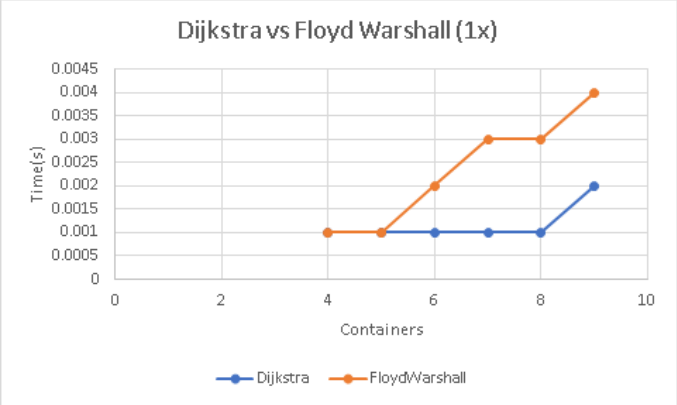

Fig3 – Graph comparing Dijkstra algorithm vs Floyd Warshall algorithm (single iteration).

Note: all of the above data was collected through the stress test functionality of the program and was using the graph of the figure 1 as reference, varying the different containers to filled or clear. On this graph, there are present 14 vertexes and 29 edges at all times which proves that the dynamic programming method of the Floyd Warshall algorithm is having a big effect (since (|E| < |V|^ 2 / 2), which are optimal conditions for Dijkstra's algorithm to prevail). The raw data can be seen on Fig4, on the next page of this paper.

IV. Connectivity Analyzation

The program also allows the user to make a connectivity analyzation of the current graph. This analyzation is separated into two different types of graphs: directed and non-directed graphs.

Directed graphs are graphs in which a depth search on a generic point doesn't return every single node and can have their vertexes analyzed in order to check if points are not strongly connected. This is done by reversing every edge on the graph and analyzing if every point can be reached; if they cannot be reached, they are not strongly connected to the graph.

Non-directed graphs are graphs in which a depth search on a generic point returns every single node and can be analyzed for articulation points. This is done by removing a

vertex from the graph and checking if the graph is still connected and then adding it (and its respective adjacencies); this is done to every vertex ad if at any point, the graph stops being connected, the current vertex being iterated is an articulation point.

V. DIFFICULTIES AND CONCLUSIONS

During this project, some difficulties were encountered as the algorithm was being made namely choosing a starting point. This was overcome by discussion and examples presented by each member of the group. The choice between using the Dijkstra or the Floyd-Warshall algorithm was also had different opinions, so we decided to implement both and compare them at the same time.

The results concluded were surprising and very interesting. By doing this project the group confirmed their theoretical study of different path calculating algorithms as well as recursive and dynamic programming techniques.

Every member of the group worked equally and felt satisfied with the final product.

	Containers	Dijkstra (total)	Floyd Warshall (total)	Dijkstra (iteration)	Floyd Warshall (iteration)	Floyd Warshall dynamic (iteration)
1x	1	0	0	0	0	0
1x	2	0	0	0	0	0
1x	3	0.001	0.001	0.001	0.001	0
100x	4	0.108	0.08	0.001	0.001	0
100x	5	0.127	0.104	0.001	0.001	0.001
100x	6	0.158	0.14	0.001	0.002	0.001
100x	7	0.201	0.181	0.001	0.003	0.001
100x	8	0.257	0.226	0.001	0.003	0.001
100x	9	0.322	0.275	0.002	0.004	0.001

Fig4 – Raw statistics data

VI Use Case Diagram

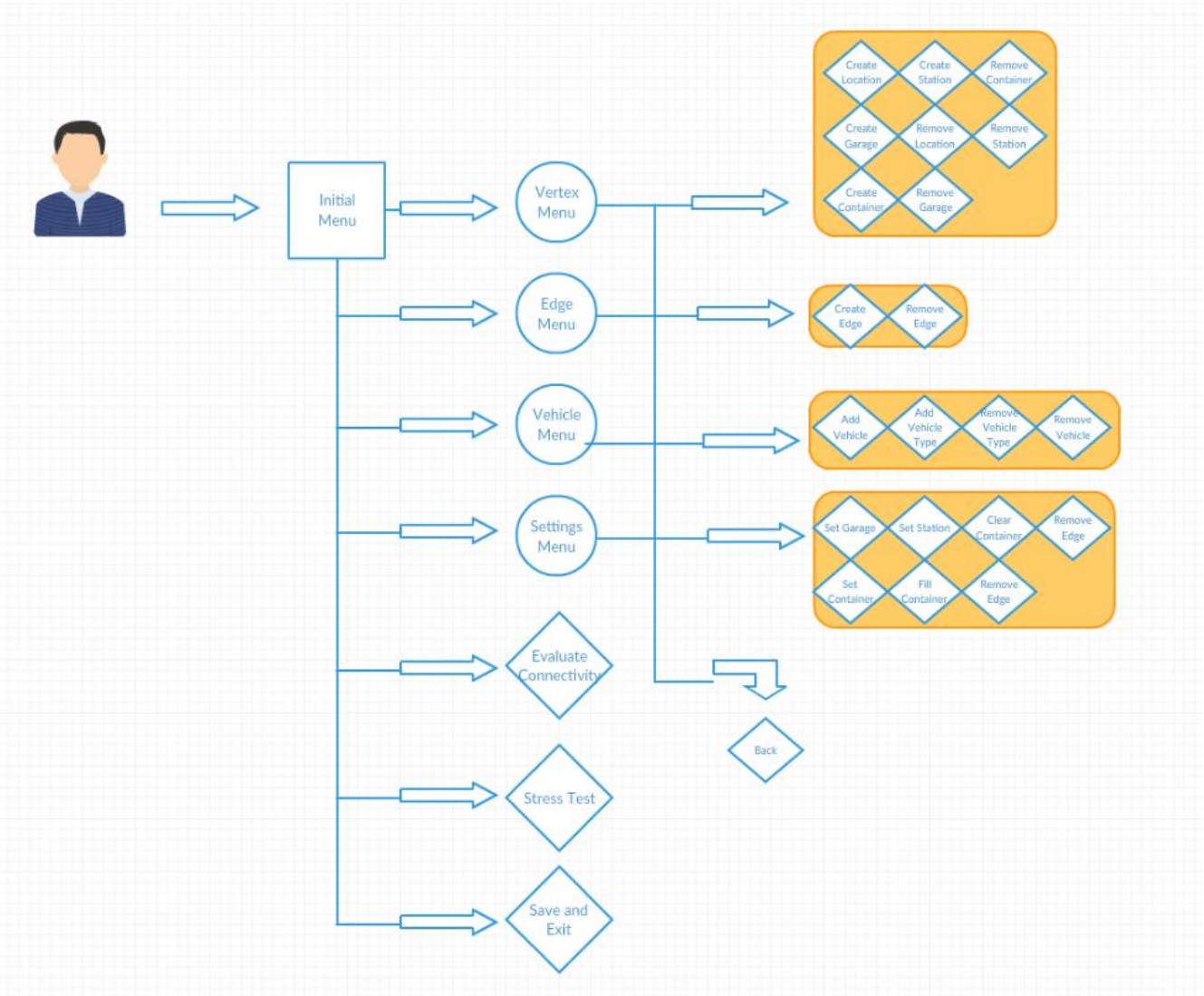

Fig 5 – Use Case Diagram

VII UML Class Model

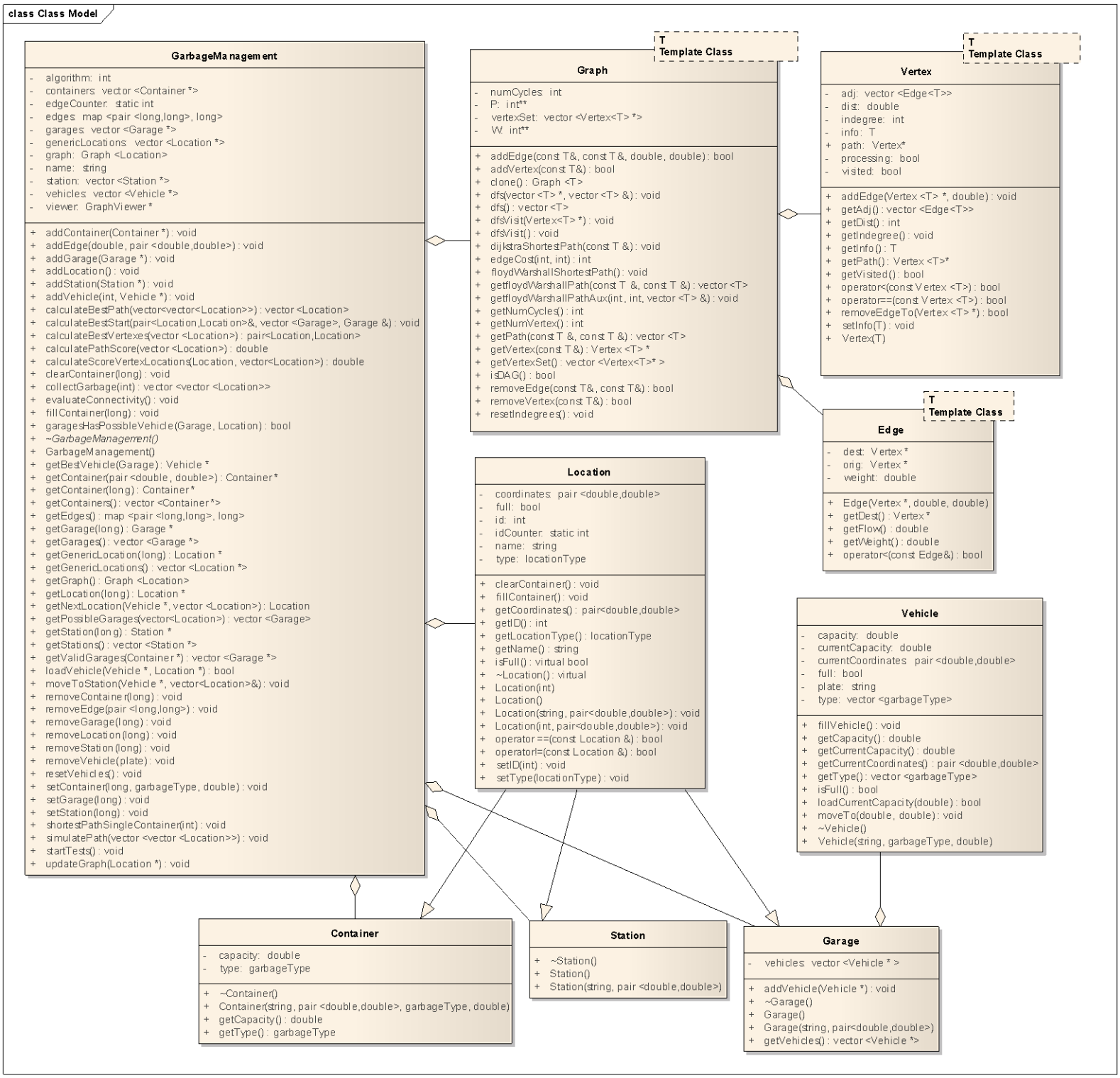

Fig 6 – Class UML Model