Faculdade de Engenharia, Universidade do Porto

U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Algorithm Design and Analysis Project

Wagons: transportation of inmates

Academic year 2019/2020

Class 7 Group 2

José Miguel Afonso Maçães - up201806622@fe.up.pt

João Castro Pinto - up201806667@fe.up.pt

Afonso Maria Rebordão Caiado de Sousa - up201806789@fe.up.pt

Table of Contents

1. Introduction

This work is carried out within the scope of the course of Design and Analysis of Algorithms, of the second year of the Integrated Master in Informatics and Computing Engineering. Carrying out this work, which focuses mainly on the matter of graphs and graph algorithms, we are training our understanding not only on this matter, but also on other practices that will be used, such as dynamic programming, greedy, backward and backward algorithms. division and conquest.

2. Brief description of the problem

The transportation of inmates is done using vans adapted for the service (informally called “wagons”). These vehicles need to be highly resistant to ensure that inmates do not flee.

One of these inmate transport systems aims to optimize the route taken by vehicles in order to collect and deliver inmates to points of interest more effectively.

This problem can be broken down into three phases:

2.1 Considering that we only have one truck

In the initial phase, we will neglect the truck's capacity and thereby calculate the shortest route between all the collection and delivery points for inmates. The van starts and ends its trips in a garage, where the company is headquartered. All establishments that are involved with the company's activity are considered to be points of collection and delivery of passengers.

As a general rule, the collection of a inmate is more important than a surrender. This is because, we do not want a inmate to spend too much time in a police station or in a court, for example.

It is considered that while the van is traveling, the company may receive another transport request from one or more inmates, and that the route will have to be recalculated based on the tasks it has to do.

Note that it is necessary that all points through which the van passes belong to the same strongly connected component of the graph. If this condition is not met, it is not possible to calculate a route between all points where the van has to pass. Therefore, without analyzing the connectivity of the graph, it may happen that we try to calculate a path that is impossible to calculate.

Note that the connection between two vertices of the graph can be changed due to road works.

2.2 Considering the characteristics of the vans

In a second phase, we start to work with multiple vehicles, taking into account their different characteristics. Thus, it becomes necessary to divide the work of the different vehicles, taking into account their capacity, that is, having to check if a van has space for inmates or if a new van will be needed. Even if a van is not full, it may make sense to send another van because the area where you will be operating can be very different. When an

order is received, it is assigned to the van that causes the total distance traveled by all vans to be minimal.

In addition, the question of fuel arises. Whenever the van receives a new collection order, it is necessary to check whether the number of kilometers remaining after placing the order is greater than the distance between the delivery place and the company's garage, where the deposit is always filled. This also requires the van to be empty when refueling occurs. Thus, the possibility of impossible orders arises, due to the fuel consumption being too high and no van can do it. *Ditto* for the capacity of the vans.

2.3 Considering that we have specialized transport

In a third phase, there are specialized vans and vans. Specializations can be useful in the repetition of certain routes (for example, court - airport, squadron A, B, C and then passing in prison), optimizing the management of inmates.

At this stage, other types of vehicle specialization are also considered, not by the route itself, but by the areas covered by the van during the route. In this case, having, for example, smaller vehicles for areas with a higher population density. For different types of vans, it may make sense to have other garages, such as for smaller capacity vans that operate in a more densely populated area.

As a result of specializations, this solution minimizes the waiting time of some inmates on recurring journeys, by having vehicles ready on fixed lines. On the other hand, possible road problems are minimized.

In this iteration, special situations are considered (see point 5.2).

3. Formalization of the problem

3.1 Input data

Wagons - set of vans to transport inmates. Van attributes (attributes with * are not considered in the first iteration):

- position - Vertex where the van is located;
- * cap - number of inmates you can carry at the same time;
- * fuel - amount of fuel available;
- * maxFuel - maximum amount of fuel;
- * consumption - average fuel consumption at 100 km.

G = (V, E) - Heavy directed graph, composed of:

- Vertex Set V:
  - localType - specification of the type of location that the vertex represents: it can be a company garage, a inmate management establishment, a workshop or a different location;
  - P - number of inmates to be collected from this vertex;
  - Pmax - maximum number of inmates that can be in this location;
  - adj (E) - edges adjacent to the vertex.
- Edge Set E:
  - dist - edge weight, represents the distance between two points on the map;
  - dest - destination edge.

POI ∈ V - Points of interest in the van's journey.

3.2 Output data

Wagonsf - sequence of vans to be used (in the 1st iteration only contains one van), in addition to the input data they are also associated with the following object:

- Course - list of edges to be visited. With each edge, the number of inmates that the van transports Pcur is saved.

3.3 Restrictions

3.2.1 About theinput data

- Wagons:
  - cap> 0;
  - fuel> 0 && fuel <maxFuel;
  - maxFuel> 0;
  - consumption> 0;
- Vertices:
  - localType has 4 different instances, each representing a different location. It can represent a company garage, a inmate-related establishment, a workshop or other places not specified;
  - P> = 0;
  - Pmax> = 0.
- Edges:
  - dist> = 0;
  - dest! = NULL.
- | POI | > = 0.

3.2.2 About the output data

- | Wagonsf | <= | Wagons |;
- On each Course object, for all edges, Pcur <cap;
- In each Course object, its first and last elements must be vertices that represent places where there are garages;
- For each Course object, whenever a van is in a garage or workshop, the following applies: Pcur = 0.

3.4 Objective functionobjective

Theis to minimize the total distance traveled by the inmate delivery and collection vans. This time, we present the following function:

$$f = \sum_{wagons \in Wagonsf} \left(\sum_{e \in course} \text{dist}(e) \right)$$

By minimizing this function, we would be, in a real situation, minimizing the costs of the company in carrying out its tasks.

4. Description of the Solution

4.1 Data Structure

4.1.1 Representation of the Graph (Map)

For the representation of the Graph in program memory there are two possibilities to consider:

1. adjacency matrix
2. List of adjacenciesadjacency

Thematrix allocates space for all potential edges even though some will never exist. Thus, with this option there will be allocation of an amount of memory proportional to the square of the number of vertices of the graph [O (| V | ^ 2)]. However, this representation has the advantage that accessing, adding and removing an edge is very time efficient [O (1)].

In turn, the adjacency list only allocates memory for the edges used. Therefore, the memory allocated by this representation will be somewhat proportional to the number of vertices plus the number of edges [O (| V | + | E |)]. The disadvantage of this option is that access to an edge has linear time complexity [O (| V |)], but maintains the efficiency of adding an edge [O (1)]. The efficiency of an edge removal operation is also affected, changing to linear complexity [O (| E |)].

Dada the nature of the problem, knowing the graph that we insert in our program - the map of Portugal - such as orders of magnitude the number of nodes and the number of edges are equal and they are very high (the map of Portugal has 1,803,214 knots and 1874296 edges), our priority will be to “save” as much memory as possible.

Therefore, we opted for the second hypothesis, the list of adjacencies, sacrificing part of the temporal efficiency for spatial efficiency. Note that in this situation, the solution of using an adjacency matrix to represent this graph for a common computer would not be feasible, since we would have to allocate space for 1803214 * 1803214 edges. This would result in terms of allocating an amount of memory in the order of terabytes, whereas through the list of adjacencies we allocate about a dozen megabytes.

4.1.2 Company manager

In the context of the proposed problem, the application will be used to obtain the routes that each van will take. For this, an interface will be implemented, consisting of a series of menus that will allow an accessible navigation by the application. Through this interface, the input data will be introduced which will allow the program to produce the results expected by the user.

Company data will be stored in an object-oriented manner. There will be a central object, the “system” that contains all other data, such as vans. In turn, vans and other more specific objects will also contain their attributes.

4.2 Algorithms

4.2.1 Connectivity Analysis

Graph connectivity is extremely important in the context of any problem that involves determining optimal paths, since the connectivity between two vertices determines whether it is possible to reach one from the other. Thus, connectivity, in problems that use graphs to represent street maps, can be useful to symbolize possible real difficulties in passing between two points, such as, for example, works on public roads. Thus, it will be necessary to assess the connectivity of the graph to identify possible locations with little accessibility, which will allow to optimize the calculation of paths between two points.

An undirected graph is said to be connected if a*depth-first search*starting at any vertex visits all vertices. The vertices that, if removed, render the graph disconnected are called Points of Articulation.

In the case of the problem in question, the graph is directed, so the analysis of connectivity will involve dividing the graph into strongly connected components, components where, from any vertex, any other can be reached. For this, an in-depth research will be done, which will determine multiple expansion trees (expansion forest), also numbering the vertices in post-order (reverse order of pre-order numbering). Then, the edges will be inverted, resulting in a new graph. A new in-depth search will be made on this new graph, starting with the highest-numbered vertex yet to be visited. The resulting expansion trees will indicate the strongly connected components. For this algorithm to work, the graph cannot be non-connected.

These components will be fundamental in the choice of paths, as they allow to associate each vertex to the component in which it is located, knowing that, if both the starting and ending vertex belong to the same component, it is mandatory to arrive from one to the other.

4.2.2 Shortest distance calculation

For this, we have several algorithms to consider, such as the Dijkstra Algorithm or the Bellman-Ford algorithm. In our case, when dealing with a heavy directed graph (see point 3), we will therefore use the Dijkstra algorithm.

The Dijkstra algorithm is recommended in our case to calculate the shortest distance between two points of interest *s* and *t*, ending the algorithm when the next node to process is node *t* (more detailed explanation below).

This algorithm then begins by marking the starting point as having distance 0, and all other vertices of the graph (intersections in the practical case of a road network) as having distance ∞ (infinite). Then, proceed the vertices with minimum distance and repeat the method successively. In this case, the edges may represent the roadways. It may be necessary to review the distance to a vertex already reached, which has not yet been processed. By having a heavy directed graph, the distance is obtained by adding the weight of the edges.

At each iteration, new vertices are processed, updating the minimum distance calculated (or not) to some of the vertices that have not yet been processed.

Dijkstra's algorithm is a greedy algorithm, minimizing the distance at each step. Its execution time is $O((|V| + |E|) * log|V|)$, being | V | the number of vertices and | E | the number of edges.

An optimization to this algorithm is to use the bidirectional Dijkstra. Compared to Dijkstra's regular algorithm, the bidirectional search algorithm performs a significantly lower number of steps.

The bidirectional search consists of performing the Dijkstra algorithm in the direction of *s* to *t* (see point 4.2.2), as well as in the direction of *t* to *s*, simultaneously. The unidirectional search, until it reaches from vertex t, will have searched for vertex t in an area about π * (d / 2) ^ 2, being the distance between ose t. In turn, bidirectional search will search for an area about 2 * π * (d / 4) ^ 2 until it finds the desired vertex. As the area surveyed by bidirectional Dijkstra is half the area surveyed by unidirectional Dijkstra, we are approximately doubling the efficiency of the algorithm.

With this optimization, instead of finishing the search when the next vertex to be processed is the arrival point *t*, the algorithm ends when it processes a vertex *x* that has already been processed in the other direction.

4.2.3 Path calculation between two vertices

For this algorithm we use the bidirectional Dijkstra described in the previous point. The only difference is that in addition to keeping the distance from one vertex to all the others, we also keep the vertices that make up the paths of one vertex to all the others. When the algorithm ends, the paths of the first Dijkstra are concatenated with the inverted path of the second Dijkstra. Hence the path between two vertices.

4.2.4 Calculating a path to pass through more than two vertices

In cases where we have to calculate a path that passes through more than two points of interest, it is necessary to repeat the algorithm for calculating a path between two vertices only (see point 4.2.3). This can happen, for example, when a van is requested to pick up a inmate from location A and deliver him to location B. Assuming that the van is initially in one of the garages, the route to be calculated will be: garage -> location A -> location B -> garage.

Therefore, we will execute the path calculation algorithm between two vertices, presented in the point above (see point 4.2.3), for each “sub-path” that needs to be done. In the example given above, it would be necessary to execute the algorithm 3 times, once for the route “garage -> location A”, once for “location A -> location B” and once for “location B -> garage”. This example only has 3 points of interest, but it could have more. For a path with *n* points of interest, it is necessary to calculate *n* paths that pass through two vertices.

Thus, the calculation between a path through more than two vertices will depend on the algorithm to calculate the path between two vertices.

Note that the temporal and spatial complexities are equal to the complexities of calculating the path between two vertices (see point 4.2.3).

5. Use Cases

5.1 Regular Routes

The most common use of the application will be to obtain the optimum route for a common service, that is, pickups and deliveries at certain locations on the map. Typically, these locations on the map are related to inmate management.

Note that a location is represented by a vertex of the graph.

5.2 Special Situations

5.2.1 Regular Route, specifying one or more roads

A situation of extraordinary possible use is the calculation of an optimal route in which the user specifies one or more roads that the route must include or exclude. Note that what changes in this situation is that instead of working only with “vertices of interest”, we also consider “edges of interest”.

Algorithmically, to exclude an edge, we temporarily remove the edge from the graph before executing the algorithm, and put the edge back on the graph after calculating the path. To include an edge, just consider that its starting and destination vertices are points of interest, which the route must contain.

In a real situation, this special calculation can be interesting to obtain a route that avoids motorway tolls, for example.

5.2.2 Extraordinary

service A special service is a service that can have points of collection and delivery on the map, vertices that do not represent inmate management places.

Algorithmically, the calculation of the route is done normally.

A real situation in which this service can be useful is a requirement for transporting inmates to do community service.

5.2.3 Maintenance of a vehicle

It is also possible that a vehicle may need to be taken to a workshop for maintenance. In this situation, the vehicle must remain in a garage until a certain date set for when the van will be repaired.

This situation does not require changes to the route calculation. The only difference is that we did not add any more points of interest after the workshop.

Note: It is possible to combine special situations

SECOND PART

6. Implemented Use Cases

6.1 Graph View

It is possible to view all available graphs, using the GraphViewer API.

6.1.1 Visualization of the connectivity of the graphs

It is possible to see the connectivity of each graph, each component being strongly connected distinguishable by the color of its vertices. Since there are a limited number of colors, there may be repetitions.

6.1.2 Visualization of the graphs with the *tags*

For each graph there is a set of *tags* that represent the various points of interest related to our work, represented in the graph with different colors for each type of point of interest.

6.1.3 Visualization of paths in graphs

When calculating a path, the program gives the possibility to show the path in graphical or non-graphical mode. If shown in graphical mode, the calculated path will be represented by green edges.

6.2 Calculating the shortest path

There are several possibilities for calculating the shortest path on all available forks. For each graph there are also suggestions of vertices that represent points of interest that can be part of this path.

6.2.1 Between two points

Using one of the A * or Dijkstra algorithms, it is possible to directly calculate the distance between two vertices.

6.2.2 Between all points

The Floyd-Warshall algorithms and an adapted version of A * are used to calculate the path between all pairs of vertices, saving them for future access. They also serve as pre-processing. We tried to implement the Dijkstra algorithm for all pairs, but we were unable to do so due to a bug.

6.2.3 Passing through the points of interest chosen by the user

The central part of our work involves, having a set of vertices associated with points of interest chosen by the user, calculating the shortest path to pass at all these points.

6.2.4 Passing through inmate pickup and delivery locations

Taking into account the context in which this work was proposed to us, we implemented an input of inmate pickup and delivery locations taking into account the times at which they should be done. To better understand how this and other menus work, simply access the instructions for each menu when running the program.

6.3 Distance

calculation The distance of each trip required by the user is calculated, “converted into time”, using the average speed of the van chosen for the task. This algorithm is not available directly to the user.

6.4 Addition and removal

operations These operations can be performed on points of interest and inmates. These are the operations that allow you to add or remove points of interest that you must pass on the trip, as well as add or remove inmates you want to collect and deliver. This way, it facilitates the manipulation of the path that the vans will take to the user.

6.5 Analysis of performance of algorithmsalgorithms

The possibility of analyzing the performance of the implementedis contemplated.

The running time for the same task that is spent by the chosen algorithms is shown, helping the comparison.

It is possible to compare:

1. A * vs Dijkstra
2. A * for all pairs vs Floyd-Warshall
3. Tarjan (only connectivity analysis algorithm implemented)
4. BFS vs DFS

6.6 Use cases not implemented, present in Delivery 1

- Starting at the level of algorithms, the main algorithm referenced in the first delivery that was not implemented in our project was the Bidirectional Dijkstra Algorithm. This being approximately 2x more efficient than the regular Dijkstra algorithm, we ended

up not implementing it, due to the increased difficulty and lack of knowledge to do so.

- Vehicle Maintenance (referred to in 5.2.3) has not been implemented, as we have not implemented any functionality related to the operation of the van.
- Some characteristics of the vans (such as their capacity, or their deposit) were also not implemented (2.2), essentially because, at the time of the first delivery, we did not have a fixed idea of how the project would be implemented and developed. We then thought that these were not essential features for our project.
- We had also mentioned a feature that involved a specialized route in which the user specified one or more roads where the van would have to pass during its route (5.2.1). We are then left to work with “Vertices of interest” instead of also considering “Edges of interest”.
- We also ended up discarding the use of specialized routes (2.3), where we would take into account population density, placing vehicles with less capacity in densely populated locations, and in the development of the project we realized that it would not be a functionality understood in the scope of the project.

7 . Data structures used

We chose to have a central object (class *System*) that manages the program's general operations and functionalities. The main data structures in this object are:

1. Object *map* of the class *Map* in which are contained the objects:
    a. Graph, which as explained in the first part of the report, is represented by an adjacency list. Vertices and edges are represented by theclasses *Vertex* and *Edge*, respectively.
    b. An *unordered_map* that maps location IDs (Class *Location* to be presented later) to location descriptions (enum Tag)

   c. Another *unordered_map* that maps a location pair (templateobject *pair<Local *, Local *>*) to an edge ID that links the pair of objects.
2. Vector of pointers for inmates (class *Prisioner)* called *prisioners*
3. Vector of pointers for points of interest (class *POI* to be explored further on) called *POIs*

To represent a location in the graph, as already mentioned, we use theclass *Local*. Objects in this class have an ID, x and y coordinates, and a description of what kind of location they represent. The description is represented by an *enum* called *Tag*. *Tag* can be 5 different types: *HQ, PRISON, POLICE, COURT, DEFAULT. HQ* represents the company's headquarters on the map. *PRISON* represents a prison, *COURT* represents a court, *POLICE* represents a police station and *DEFAULTundisclosed* represents an/ irrelevant point.

The representation of a point of interest is done by theclass *POI*. Points of interest are used for route calculation. A point of interest has a location (*Location*) and a date and time associated with it (class *DateTime*).

The algorithms that operate on graphs that we implement, all need auxiliary data:

- For the Dijkstra algorithm we addto theclass *Vertex* two attributes. We add an attribute of type *double* that represents the distance from that vertex to the origin vertex, and we also add one pointed to a *Vertex* that points to the previous vertex of the path (once the algorithm has been calculated). For this algorithm we also use a priority queue.
- The A * algorithm uses the same attributes that Dijkstra uses. *Ditto* for the priority queue. The priority queue is ordered taking into account the distance from the vertex to the “destination” of the algorithm.
- The Floyd-Warshall algorithm uses two matrices, *dist* and *pred*. Both have size *v*v*, where *v is* the number of vertices of the graph. Thematrix *dist* is amatrix *double* and thematrix *pred* is a matrix of vertex pointers. *Dist* is used to store distances between all pairs of vertices, and *pred* stores the predecessor on the path between all pairs of vertices.
- Tarjan's algorithm uses a stack and three new attributes at the vertices. The stack is used to know which vertices are currently being processed by the algorithm. Of the attributes added to the vertices, the most relevant is the *tarjanLowlink,* from which we distinguish strongly connected components.

8. Implemented algorithms

8.1 Dijkstra's

algorithm This algorithm seeks to find the shortest path from one point to all others. For that, it uses a minimum priority queue (*min_priority_queuestores*), where it temporarilythe vertices that it will have to analyze.

The row extraction criterion is the minimum weight of the edge that leads to the vertex to be chosen. The candidate weight (*candidateWeight*) is simply the current weight plus the weight of the edge to the candidate vertex and it is based on it that the position of the priority row will be the vertex. The lower the weight, the sooner we are going to expand the expanded path from that vertex.

Pseudo-code:

```
function Dijkstra (Graph, source):
    min_priority_queue Q
    for each vertex v in Graph:
        v-> dist ← INFINITY
        v-> pred ← UNDEFINED

    add v to Q
    source-> dist ← 0

    while Q is not empty:
        u ← extract Min from the priority_queue

        for each edge and of u:
            candidateWeight ← v-> dist + length (u, v)
            if candidateWeight <v-> dist:
                v-> dist ← candidateWeight
                v-> pred ← u
                if ! e-> dest-> visited:
                    Q.insert (e-> dest)
                else:
                    Q.givePriority (e-> dest)
    return
```

8.2 Algorithm A *

This algorithm is similar to Dijkstra, with the exception of the ordering of the priority queue. Neste algoritmo, vamos determinar a posição na fila de prioridade tendo também em conta a distância euclidiana do vértice atual ao vértice de destino, bem com a alteração o critério de paragem, que é inexistente no Dijkstra. O critério de paragem verifica-se quando processamos o vértice de destino retirado da fila de prioridade.

Pseudo-código:

```
function dist(source, dest):
    return sqrt((source->x - dest->x)² + (source->y - dest->y)²)

function A*(Graph, source, dest):
     min_priority_queue Q

     for each vertex v in Graph:
         v->dist ← INFINITY
         v->pred ← UNDEFINED

     add v to Q
     source->dist ← 0

     while Q is not empty:
         u ← extract Min from the priority_queue

         if (u == dest) break

         for each edge e of u:
             v ← e.dest
             candidateWeight ← v->dist + e->weight + dist(v, e->dest)
             if candidateWeight < v->dist:
                 v->dist ← candidateWeight
                 v->pred ← u
     return
```

8.3 Algoritmo Floyd-Warshall

Este algoritmo pré-processa os caminhos e distâncias entre todos os pontos do grafo.

Pseudo-código:

```
function Floyd-Warshall(Graph):
    for each vertex u in Graph:
        for each edge in u->adj:
            dist[u][edge->dest] ← edge->weight
            pred[u][edge->dest] ← u
    for each vertex v in Graph:
        dist[v][v] ← 0
        pred[v][v] ← NULL

    for k from 1 to |V| do // standard Floyd-Warshall implementation
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j] :
                    dist[i][j] ← dist[i][k] + dist[k][j]
                    pred[i][j] ← pred[i][k]
```

8.4 Algoritmo de Tarjan

Neste algoritmo é dividido o grafo em Componentes Fortemente Conexo [*Strongly Connected Components* (SCC)], sendo para isso utilizada uma *Depth-First Search* (DFS).

Para isso, procura identificar os diferentes SCC utilizando um identificador único, que vai sendo guardado no *tarjanLowLink*.

Serve-se também de uma pilha onde são colocados os vértices visitados.

Pseudo-código:

```
function dfsTarjan(Graph, v):
    Graph->tarjanStack.push(v)
    v->onStack = true
    v->tarjanID = v->tarjanLowlink = Graph->nextTarjanID
    v->visited = true
```

```
 for each edge in v->adj:
      if (!edge->dest->visited):
            dfsTarjan(Graph, v)
      if (edge->dest->onStack):
            v->tarjanLowLink=min(v>tarjanLowlink,
aaaaaaaaaaaa                              edge->dest->tarjanLowlink)

  if (v->tarjanId == v->tarjanLowlink):
       for each vertex v = Graph->tarjanStack.extractTop():
            v->onStack = false;
            v->tarjanLowlink = at->tarjanId;
            if (v->tarjanId == at->tarjanId) break;

        this->sccCount++;

 return

   function tarjan(Graph):
      Graph->tarjanStack.clear()
      Graph->nextTarjanID = 0
      Graph->sccCount = 0
      for each vertex v in Graph:
           if !v->visited:
              dfsTarjan(Graph, v)

      Graph->tarjanStack.clear()
      return
```

8.5 Algoritmo de Dijkstra para todos os pares

Este algoritmo é uma tentativa de melhorar o algoritmo de Floyd-Warshall para pré-processar o grafo. No programa real, não conseguimos fazer com que funcionasse.

Pseudo-código:

```
function allPairsDijkstra(Graph):
      Graph->dist.fill(INF)
      Graph->pred.fill(NULL)

      for each vertex u in Graph:
            Dijkstra(Graph, u)
            for each vertex v in Graph:
                  if (dist[u][v] > v->dist):
                        dist[u][j] = v->dist
                        pred[u][j] = v->path

      return
```

8.6 Algoritmo A* para todos os pares

Este algoritmo é uma tentativa de melhorar o algoritmo de Floyd-Warshall para pré-processar o grafo. No entanto, acabamos por perceber que este algoritmo não tem melhor eficiência que o algoritmo de Floyd-Warshall.

Pseudo-código:

```
function allPairsA*(Graph):
      Graph->dist.fill(INF)
      Graph->pred.fill(NULL)
      for each vertex u in Graph:
            for each vertex v in Graph:
                  A*(Graph, u, v)
                  if (dist[u][v] > v->dist):
                        dist[u][j] = v->dist
                        pred[u][j] = v->path

     return
```

8.7 Algoritmo de resolução do problema do *caixeiro viajante*

Para resolver o problema de passar em vários em pontos numa só viagem utilizamos a técnica do problema do *caixeiro viajante*. Há várias formas de resolver esse problema. Optamos pelo algoritmo "ingénuo" que calcula todas as permutações de pontos de interesse por onde queremos passar.

Pseudo-código:

```
function tspNaiveSolution(Graph, POIs): // POIs is an array
    minCost = INF
    cost = 0.0

    best_solution = {}
    do {
        cost = Graph->getTotalWeights(POIs)
        if cost < minCost:
            minCost = cost
            best_solution = POIs

    } while (permutate_order(POIs))

    return best_solution
```

Nota: No programa real, este algoritmo foi utilizado de duas formas: esta versão que só tem a distância em conta e combinando com a verificação de se cada permutação consegue chegar pontualmente a todos os pontos de interesse. Portanto, é possível que um este algoritmo não encontre nenhum caminho possível.

9 . Análise de complexidade

9.1 Análise teórica

9.1.1 Algoritmos de caminho mais curto Single Source

Temos dois algoritmos implementados com o intuito de calcular a menor distância entre dois vértice especificados: Dijkstra e A *. Sendo então esses que vamos comparar.

Teoricamente, a nível da sua complexidade temporal, no pior caso, os dois algoritmos teriam uma eficiência relativamente equivalente, tendo ambos um tempo de execução teórico de (|V| + |E|) * log |V|. No entanto, no caso médio, o algoritmo A* seria mais eficiente.

A complexidade espacial é semelhante entre os dois, proporcional ao número de vértices (V) do grafo utilizado, no pior caso. No entanto, no caso médio, raramente é utilizada a totalidade do espaço.

9.1.2 Algoritmos de caminho mais curto All Pairs

Para os algoritmos que calculam o caminho entre todos os pares existentes de vértices, vamos comparar os seguintes algoritmos: A* para todos os pares e Floyd-Warshall.

Teoricamente, o tempo de execução é menor para o algoritmo de Floyd-Warshall : V ^3, comparativamente ao de A* : (V ^3 + E * V ^2) * log(V).

A complexidade espacial é semelhante entre os dois algoritmos. Tendo uma complexidade de V ^2, no pior caso.

Inicialmente, tencionamos também implementar o algoritmo de Dijkstra para todos os pares, sendo este o mais eficiente dos três a nível temporal, com uma complexidade temporal de (V ^2 + E * V) * log(V). No entanto, após várias tentativas, acabamos por não o conseguir fazer com sucesso, deixando de qualquer forma o código desenvolvido no nosso projeto (Graph.h).

Para obter caminhos entre dois vértices, se pre-processarmos os grafos com estes algoritmos, vamos ter eficiência temporal constante. Isto acontece porque passamos a ter a informação sobre os caminhos guardada em memória e só precisamos de aceder a esses dados.

9.1.3 Algoritmos de análise de conectividade / Algoritmo de Tarjan

Implementamos igualmente o algoritmo de Tarjan, que encontra os componentes

fortemente conexos de um grafo direcionado.

A complexidade temporal deste algoritmo é, no pior caso, V + E, prevendo-se então um algoritmo eficiente e de rápida execução.

A complexidade espacial, será no pior caso V. O pior caso acontece quando todo o grafo tem apenas um componente fortemente conexo, em que vai haver um momento que o na pilha utilizada pelo algoritmo (ver acima) estão todos os vértices.

9.1.4 Algoritmo para o problema do caixeiro viajante

Implementamos a solução naive deste problema (implementação detalhada em 8.7).

Esta, tem um complexidade temporal na ordem de N!, sendo N o número de pontos de interesse que passamos ao algoritmo. A complexidade espacial é proporcional ao tamanho do caminho resultante, sendo então em última análise proporcional ao número de vértices do grafo.

Acabamos por não testar a performance temporal deste algoritmo como fizemos para os restantes, por se tratar de uma funcionalidade relativamente complicada.

9.2 Análise Temporal Empírica

Os tempos calculados foram todos obtidos fazendo a média de 5 execuções do respectivo algoritmo.

9.2.1 Algoritmos de caminho mais curto

Algoritmos de caminho mais curto são uma família de algoritmos com o intuito de resolver o problema do caminho mais curto. Este problema pode tomar várias formas, havendo então dois principais tipos de algoritmo para este problema:

9.2.1.1 Algoritmos Single Source

Os nossos algoritmos de single source a analisar encontram a distância mais pequena entre um certo vértice especificado e um outro vértice de destino.

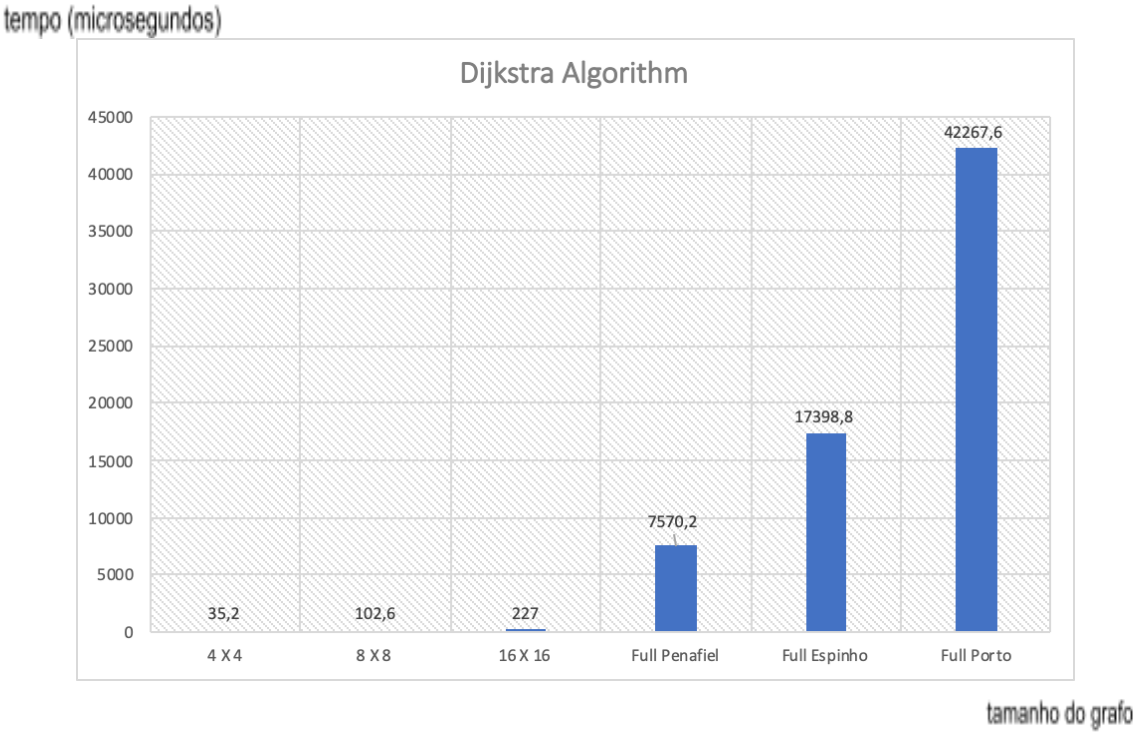
Para os algoritmos de Single Source, sendo possível testar a performance temporal entre um par de vértices à escolha do grafo, e para manter a coerência na comparação entre os diferentes grafos, decidimos comparar o tempo de execução tendo sempre como ponto inicial o Id dos nossos Headquarters e como destino a estação de polícia mais próxima

deste último:

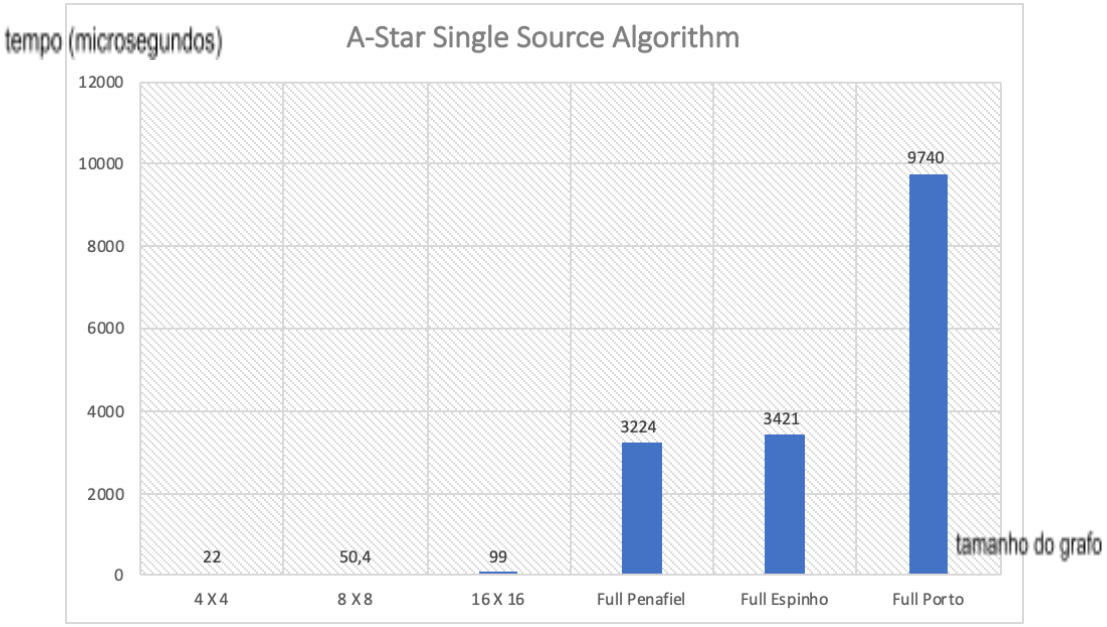
HQ Ids Usados: {4 X 4 ; 0 }, {8 X 8 ; 40 }, {16 X 16 ; 155 }, { Full Penafiel ; 159 }, { Full Espinho ; 6861 }, { Full Porto ; 19548 } ;

Destiny (Police Station) Ids Usados: {4 X 4 ; 10 }, {8 X 8 ; 24 }, {16 X 16 ; 204 }, { Full Penafiel ; 787 }, { Full Espinho ; 4019 }, { Full Porto ; 6011 } ;

9.2.1.1.1 Algoritmo Dijkstra



*9.2.1.2.1 Algoritmo A * para dois vértices definidos*



9.2.1.2.1 Comparação final de algoritmos

Tendo em conta que neste caso estamos a calcular o caminho entre dois pontos do grafo, esses pontos variando consoante o tamanho do grafo, é mais interessante comparar para cada grafo, a diferença de execução dos dois algoritmos. Constata-se então que o algoritmo de Dijkstra tem um tempo de execução mais elevado que o algoritmo A*.

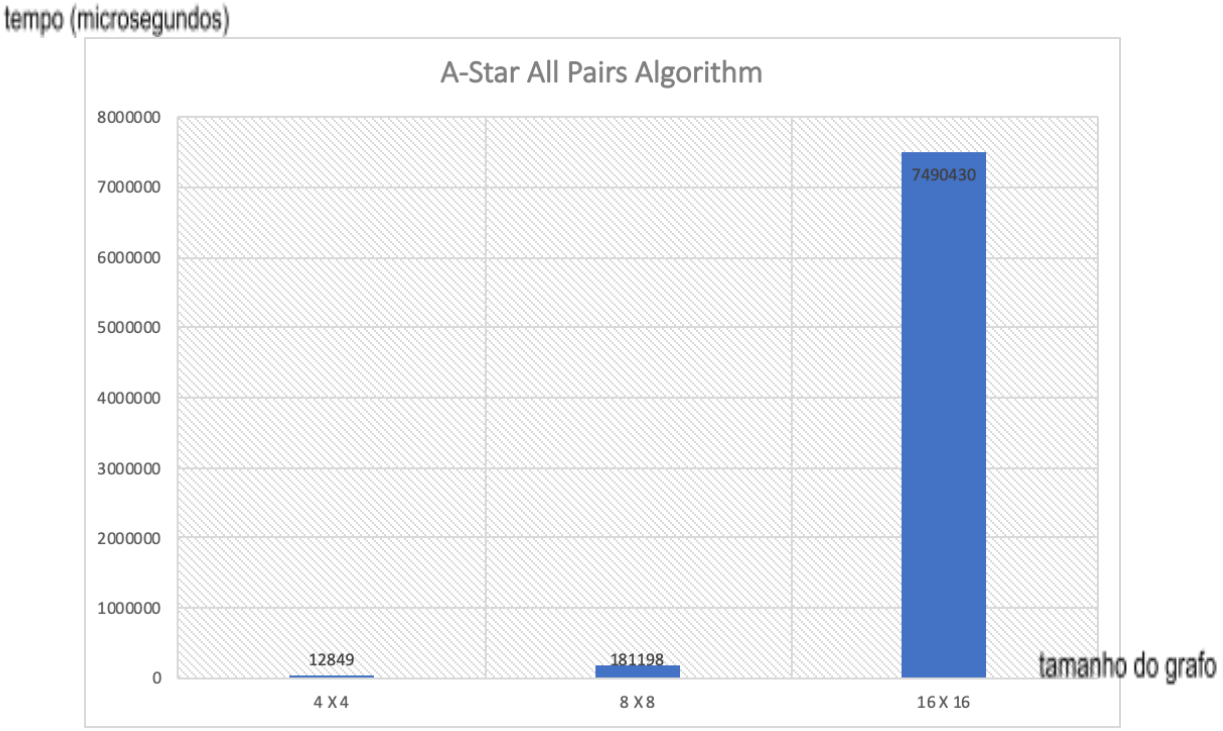
Como tínhamos previsto na análise teórica, no caso médio, verifica-se que o algoritmo A* é então mais eficiente que o algoritmo de Dijkstra.

9.2.1.2 Algoritmos All Pairs

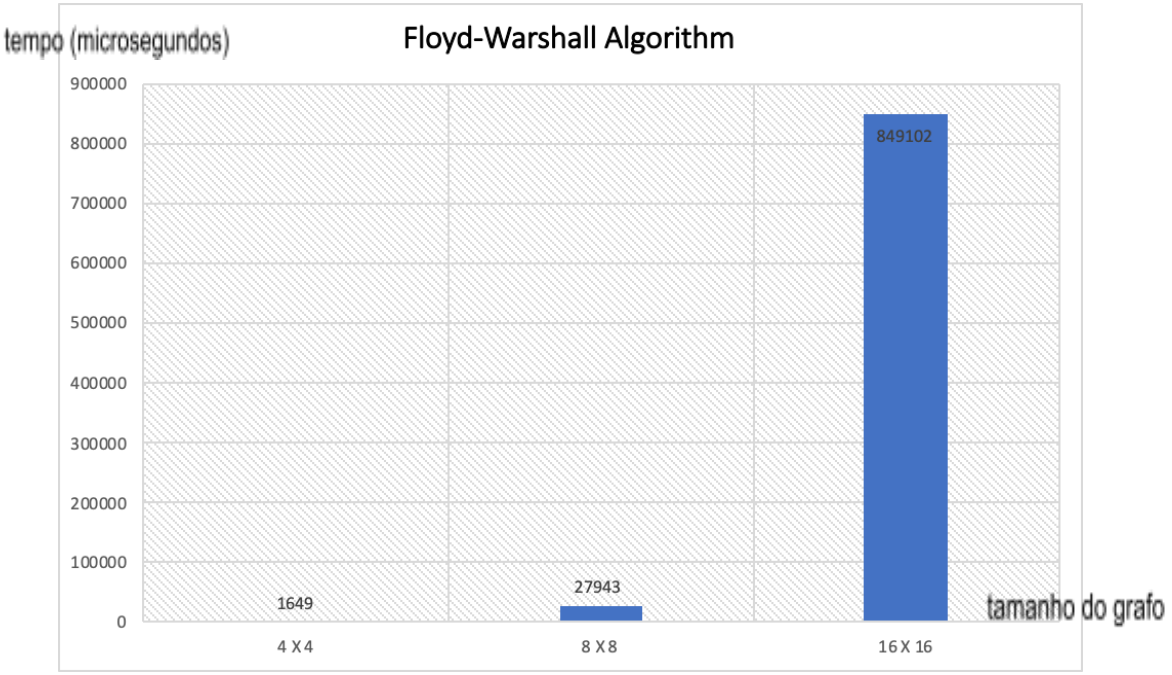
Os algoritmos All Pairs que implementamos encontram o caminho mais curto entre qualquer par de vértices.

Os algoritmos de pesquisa do menor caminho para todos os pares existentes, quando aplicados a grafos muito grandes como é o caso dos mapas das cidades de Penafiel, Porto ou Espinho, são muito demorados, chegando mesmo a ser não recomendável a utilização, por isso, vamos fazer apenas a análise temporal para graphs com grids de tamanho máximo 16 x 16, neste caso.

*9.2.1.2.1 Algoritmo A * para todos os pares*



9.2.1.2.2 Algoritmo Floyd-Warshall



9.2.1.2.3 Comparação de Algoritmos

Como seria expectável, o tempo de execução dos algoritmos aumenta consoante o tamanho do grafo.

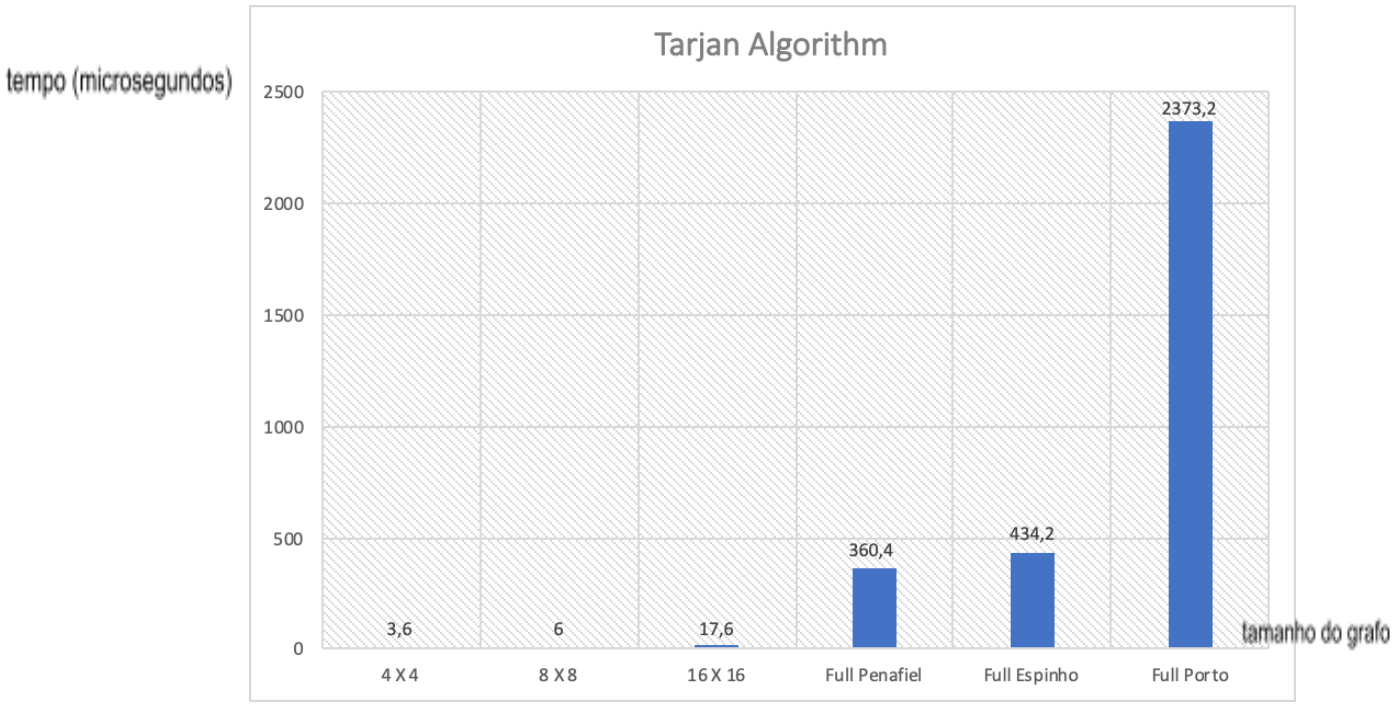
Em comparação entre eles, o algoritmo de Floyd-Warshall termina nos 3 diferentes caso de grafo com um tempo de execução inferior ao tempo de execução do A*, na ordem de 10^1.

Confirma-se então de novo a boa implementação dos algoritmos, ao ter os resultados que correspondem à análise teórica efetuada previamente (9.1).

9.2.2 Algoritmos de análise de conectividade / Algoritmo de Tarjan

O nosso principal algoritmo de conectividade implementado e utilizado é o algoritmo de Tarjan, que tem como objetivo encontrar os componentes fortemente conexos de um grafo direcionado.

Após a análise temporal (página 23) constata-se que é um algoritmo com tempo de execução relativamente rápida.



10 . Conectividade dos grafos utilizados

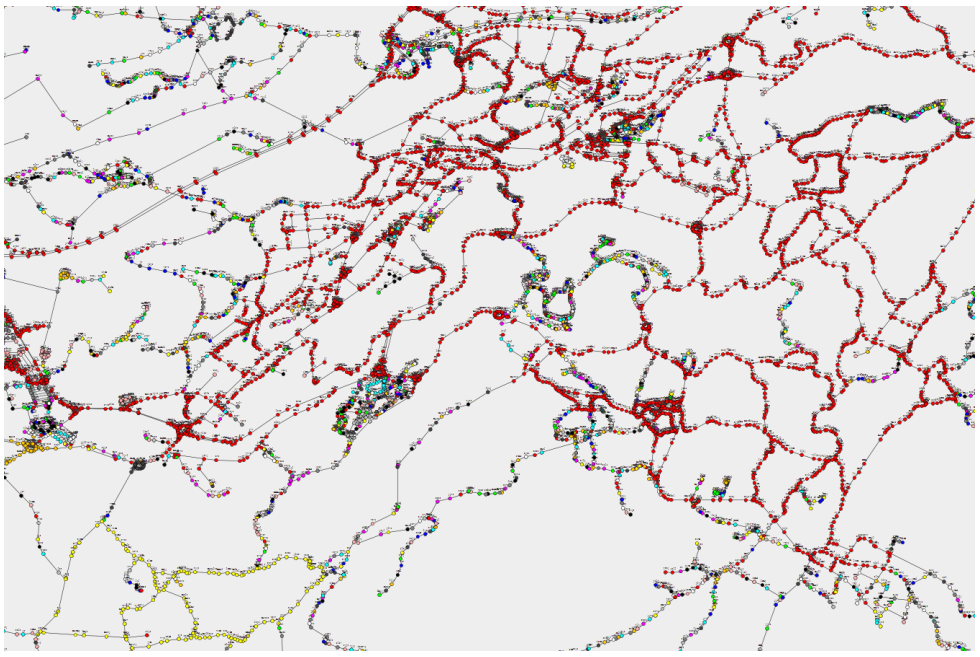
Para todos os grafos disponibilizados foi efetuada uma análise de conectividade a seguir a ler o grafo para memória.

10.1 Algoritmo Tarjan

Para toda a análise de conectividade foi utilizado o algoritmo de Tarjan. Sucintamente, este algoritmo divide o grafo em componentes fortemente conexos.

10.2 Representação da conectividade

Está presente a possibilidade de visualizar uma representação da conectividade para o grafo escolhido. Esta representação serve-se do leque de cores presente na API GraphViewer para atribuir cores iguais a vértices pertencentes ao mesmo componente fortemente conexo. Como o número de cores é limitado, pode haver repetições.



Conectividade do Grafo Completo de Espinho

10.3 Aplicação da análise da conectividade

Para este trabalho, a análise de conectividade é utilizada para determinar se é possível a partir dum ponto do grafo, ir e voltar a um segundo ponto. Resumidamente, se os 2 pontos pertencerem ao mesmo componente fortemente conexo, deve ser possível calcular um caminho entre eles.

Assim, antes do cálculo de caminho entre dois pontos, da adição dos pontos de interesse e da adição dos locais de origem e destino associados aos prisioneiros, é verificada esta condição para saber se é possível efetuar o cálculo ou adicionar, respetivamente. Esta verificação tem de ser feita, uma vez que para calcular um caminho que comece e termine no mesmo ponto, é necessário que todos os pontos do caminho pertençam ao mesmo componente fortemente conexo.

Para além disso, quando são apresentadas sugestões de pontos de interesse para grafos que não são completamente conexos, essas sugestões pertencem ao mesmo componente fortemente conexo para ser possível calcular sempre caminhos com os vértices sugerido.

11 . Conclusão Geral

Em suma, o trabalho correu como esperado, tendo sido atingidos grande parte dos objetivos relativos à proposta da docência.

A distribuição do esforço não foi propriamente igualitária, tendo o João Pinto trabalhado mais (fez aproximadamente 50% do trabalho) que os outros dois membros do grupo, entre os quais o trabalho restante foi bem repartido: 25% - José Maçães, 25% - Afonso Caiado. Foi intercalado o trabalho em simultâneo, com reuniões online, com o trabalho em paralelo, com cada um a fazer a sua parte.

Foi um trabalho na generalidade bem conseguido, que despertou o interesse dos três membros por se tratar de implementar uma aplicação que apresenta caminhos minimais para chegar de um local ao outro, como vemos constantemente na vida diária presente no GPS.

12. Referências Bibliográficas

- “Algoritmos em Grafos: Caminho mais curto (Parte I)”, R. Rossetti, L. Ferreira, HL Cardoso, F. Andrade. CAL, MIEIC, FEUP;
- “Algoritmos em Grafos: Caminho mais curto (Parte II)”, R. Rossetti, L. Ferreira, HL Cardoso, F. Andrade. CAL, MIEIC, FEUP;
- “Algoritmos em Grafos: Conectividade”, R. Rossetti, L. Ferreira, HL Cardoso, F. Andrade. CAL, MIEIC, FEUP;
- Fiset, W., 2019, *Algorithms Course - Graph Theory Tutorial from a Google Engineer*, video, Youtube, visto 15 de Abril 2020, <https://youtu.be/09_LlHjoEiY>.
- Wikipédia para consulta de formas de pseudo-códigos