Tour Mate App

Alexandre Abreu
up201800168@fe.up.pt

Eduardo Correia
up201806433@fe.up.pt

Juliane Marubayashi
up201800175@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
May , 2020

Keywords

Tourism, Graph Theory, Network Graph

Abstraction

The goal of this project is to build the *TourMateApp* application.

This hypothetical application's purpose is to generate adaptable urban tourist routes, according to the personal preferences of its user.

The methodology must use an implementation of graphs to get the values.

Introduction

Problem Description

The problem discussed in this paper is how to evaluate the best tourist path to a client that is in a specific place and has limited time to spend, taking into account his preferences.

We must take into account the time spent in commutations and the expected time of visits.

For example reasons, we used the Porto city as a reference model to work on.

Model Formulation

Edge Weight

The maximum speed and the distance to travel along an edge are used to calculate the cost of moving between two nodes, assuming there will be no interaction with other vehicles or people.

Input Data

G = (V, E) - dense weighted directed graph:

- V - set of nodes representing points in the map:
  - ID - unique, the id of the node in the Open Street Map.
- E_i - edge representing a path between two nodes that is identified by the OSMid = i:
  - $OSMid$ - the id of the edge in the Open Street Map.
  - S_i - The id of the source node.
  - T_i - The id of the target node.
  - L_i - Length of the road.
  - M_i - The maximum speed of the road.

  - O_i - True if the street is just one way, False otherwise.

Input POI

- ID: Id of the Vertex
- $Name$: Name of the POI
- $Time$: Avarage time that a person spends in that place

Input extraction from csv

To initially obtain the map data from OSMnx [1] we used a Python library that retrieves, models, analyzes, and visualizes street networks and other spatial data from OpenStreetMaps [2].

On that way, we can easily obtain a visual representation of streets and locations much like a graph - in which the crossings/POIs would be vertexes and the streets would be edges - and export it to a file our algorithm could parse.

Furthermore, the user would inform about its preferences, according to its trip purpose (recreational, work...), such as the list of POIs to visit, the maximum total time available to spend, the type of circuit (by foot, bicycle, car, public transportation...).

The input data includes *.csv* files (obtained with OMSnx library [1] and cleaned with Pandas [3]) stores information about the edges and nodes.

Output Data

Output data available for the user

- The path to pass by in order to achieve the points of interest as the path line shown on the map.
- Name of the streets and directions needed to be followed in respective order.
- The total time to be spent by the tourist, including visiting the POIs, and in the way to get to the places.
- The total distance to go.

Output data after loading files

After reading the *.csv* file with edges and nodes information, a *Dense Direct Graph* $G(V, E)$ will be built with V vertexes and E edges.

- E_i
  - S_{ref}: source node reference
  - T_{ref}: target node reference
  - w_{ij}: calculated by $max\_velocity/road_{lenght}$
  - name: name of the street
- V
  - id: vertex number identification.
  - Adj_i : adjacent list for a vertex i.
  - t: average time spent in the POI.

Output after loading data about POIs

A class containing all the POI's will be built

- POI
  - ID: Id of the Vertex
  - $Name$: Name of the POI
  - $Time$: Avarage time that a person spends in that place
- $POIs$: Hashtable(string, POI) containing all the POIs.
- $CityName$: Name of the city with the POI's stored.

Objective Function

The optimal solution of the problem resides in minimizing the total time spent in commuting and the total traveled time and maximizing the number of points of interest visited.

- T_{total} is the amount of time the user has to spend
- d_{ij} is the distance between point i and point j
- v_{ij} if the allowed between point i and j
- POI is the set of points of interest visited (POI)
- t_i is the time spent in a POI

$$f = min(\sum_{i,j \in POI} \frac{d_{ij}}{v_{ij}}) \times min(\sum_{i \in POI} t_i) \wedge min(\sum_{i \in POI} t_i) <= T_{total}$$

Constraints

- $\lambda(i,j) > 0$, $\forall (i,j) \in E$
- $g > 0 \wedge f > 0$
- $T_i \neq NIL \wedge T_{ref} > 0$, $\forall T_{ref} \in V$
- $S_i \neq NIL \wedge S_{ref} > 0$, $\forall S_{ref} \in V$
- $L_i > 0$
- $M_i > 0$

Identification of Supported Cases

Considering the initial options given in the main menu of the application, the supported cases includes:

- Just one type of locomotion: car. However the algortihm could work for any time of locomotion.
- The client can choose the points of interesting

Conectivity of the maps implemented

- All the graphs implemented were heavy graph, since all the edges have weight.
- The maps used for the project are all connected, since from one node we can reach all the others.

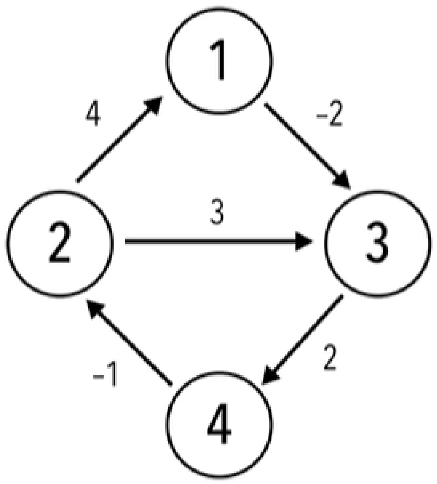
Solution Description

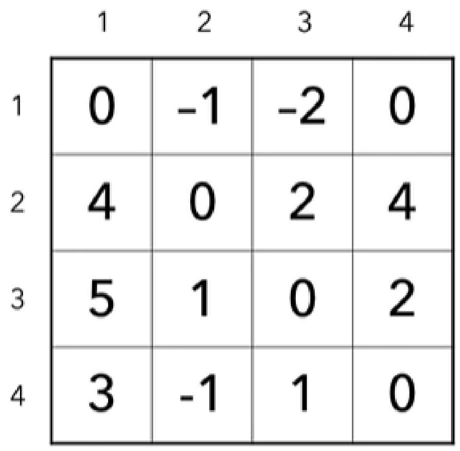
Pre-processing

The main algorithm to be used in the application is the *Floyd Warshall* [4] [5].

Algorithm 1 Floyd-Warshall

```
procedure SHORTEST PATH(G=(V, E))
    $dist \leftarrow N \times N$ matrix
    for each vertex $v \in dist$ do
        $dist[i][i] \leftarrow 0$
    for each edge $w \in dist$ do
        $dist[u][v] \leftarrow weight(u, v)$
        $pred[u][v] \leftarrow origin(w)$
    for k from 1 to V do
        for i from 1 to V do
            for j from 1 to V do
                if dist[i][j] > dist[i][k] + dist[k][j] then
                    dist[i][j] = dist[i][k] + dist[k][j]
                    pred[i][j] = pred[k][j];
```





	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

This algorithm considers the intermediate nodes to find and store all the shortest distance between nodes i and j with k as intermediate vertex ($i \rightarrow k \rightarrow j$, which can simply described by the recursion formula:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if k} = 0 \\ min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{ki}^{k-1}) & \text{if } k \ = 1 \end{cases} \tag{1}$$

Besides the $O(|V^3|)$ complexity, this is a good approach for dense graphs like city networks. However, the *Floyd Warshall* algorithm doesn't store the path between nodes. To do so, we can consider a new matrix called Π (*pred*).

For each d_{ij}^{k} element in the matrix of shortest path D^k the matrix Π (*pred*) will contain the respective shortest path for the pair of nodes i and j.

Formally, each element of Π can be described as it follows for $k > 1$:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{k-1} & \text{if } d_{ij}^{k-1} <= d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1} & \text{if } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases} \tag{2}$$

For $k = 0$ the definition will be:

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{if } w = \text{inf} \vee \text{i = j} \\ i & \text{if } i \neq j \wedge w < \text{inf} \end{cases} \tag{3}$$

To complete the algorithm analysis, the spacial complexity for this algorithm should be $O(|v^2|)$, since it uses two auxiliary N × N matrices.
In order to make the waiting time less significant, improvements to this algorithm will be considered.

Choose the next vertex

After calculating all the minimum distance between all the pairs of vertices, it's necessary to define a strategy to choose the next vertex.

Greedy approach

- **vPOI** is the **vector of pointers** containing the points of interest represented by the POI structure defined previously.
- **maxTime** is a variable containing the maximum time that a person can spend.
- **origin** parameter represents the position of the initial point at vPOI.
- Always consider the **vPOI[0]** as the initial point.

Remember that the index of a *origin* vertex in vPOI is not the same index of this same vertex in the vector of vertices in the graph.

Algorithm 2 Greedy approach - Pre Processing

```
1:  procedure TRAJECTORYORDER(vPOI, maxTime, G=(V,E))
2:      $order \leftarrow$ vector                         ▷ Will store the final sequence of vertex
3:      $visited \leftarrow |V|$ vector
4:      $idNext \leftarrow int$                            ▷ Id of next POI to be visited
5:      $initalTime \leftarrow maxTime -$ time spend in the origin POI
6:
7:      $indexOrigin \leftarrow$ finds index of origin in V
8:      $visited[indexOrigin] = true$
9:
10:     for each poi $p \in vPOI$ do
11:         $idNext \leftarrow nextPoi$(indexOrigin, vPOI, visited, initialTime)
12:
13:         if $idNext = -1$ then return $order$         ▷ Don't have time to visit any other POI
14:
15:         $originPos \leftarrow$ finds origin index in V
16:         $nextPos \leftarrow$ find idNext index in V
17:         $FloydPath \leftarrow getFloydPath$(originPos, nextPos)
18:
19:         $order \leftarrow order[0...|order| - 1] + FloydPath[0...|FloydPath| - 1]$    ▷ Add path to the next POI
20:
21:         $visited[idNext] = true$
22:         $origin \leftarrow idNext$
23:     return order
```

Algorithm 3 Main Greedy Approach

```
procedure NextPoi(originIndexV, vPOI, visited, maxTime, G=(V,E))
    $actualVertexIndex \leftarrow originIndexV$
    $minWeight \leftarrow \infty$

    $selectedIndexVPOI \leftarrow -1$

    for i from 0 to |vPOI| − 1 do
        $nextVertexIndex \leftarrow$ finds vPOI[i] index in V
        $time \leftarrow$ finds the time to be spent in vPOI[i]

        if dist[actualVertexIndex][nextVertexIndex] + time < minWeight and visited[i] = false then
            $minWeight \leftarrow dist[actualVertexIndex][nextVertexIndex] + time$
            $selectedIndexVPOI \leftarrow i$

    $maxTime \leftarrow maxTime - minWeight$

    if maxtime < 0 then return -1                    ▷ Can't visit any other point

     return selectedIndexVPOI
```

Dynamic approach

Considering the objective function of achieving the **maximum number of nodes in a minimum time**, the following graph shows how the dynamic solution will approach the problem.
Considering the time as the weight for each node, the data for this demonstration is consisted of:

- A matrix M containing the time to go from one node to the other.
- A vector C with the time spent in each node.
- The maximum time (MT) of 46 units
- A function of Total Time Spent (TTS), where the TTS for one node k with father i is:

$$TTS = TTS[i] + C[k] + M[i][k]$$

Where the TTS for the root is

$$TTS = C[root]$$

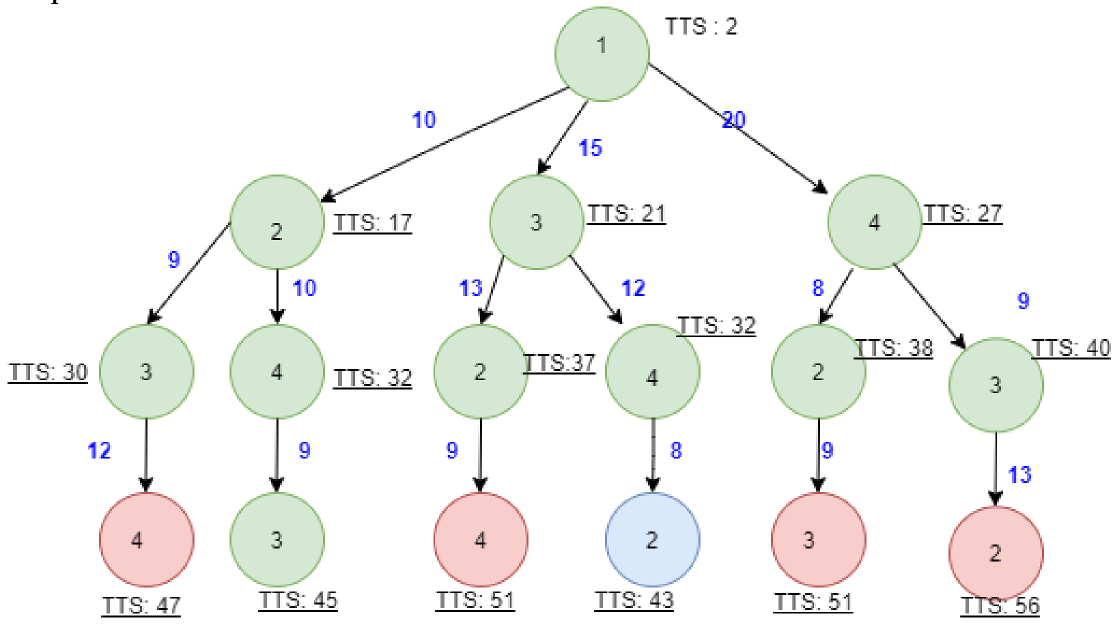
Matrix M

	1	**2**	**3**	**4**
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Vector C

1	**2**	**3**	**4**
2	3	4	5

Graph



Result: Node 2.
Note that the best solution is the deepest node in the Graph with minimum TTS < 46.

- **vPOI** is the **vector of pointers** containing the points of interest represented by the POI structure defined previously.
- **maxTime** is a variable containing the maximum time that a person can spend.
- **origin** parameter represents the position of the initial point at vPOI.
- Always consider the **vPOI[0]** as the origin POI.

Algorithm 4 Traveling salesperson pre preprocessing

```
1: procedure PREPROCESSING(vPOI, maxTime, G=(V, E))
2:     for i from 0 to |vPOI| − 1 do
3:         $vPOI[i]$ ← set visited false
4:
5:     $minDistance \leftarrow 0$
6:     $level \leftarrow 0$
7:     $initialTime \leftarrow time$ − find time spend in vPOI[0]
8:
9:     return $travelingSalesperson$(0, vPOI, |vPOI|, minDistance, initialTime, level)
```

- **actualPoint** is the index of the actual point in **vPOI**
- **vPOI** is the **vector of pointers** containing the points of interest represented by the POI structure defined previously.
- **available** how many not visited points of interest in vPOI
- **minDistance** The minDistance until now.
- **maxTime** The amount of time in minutes that a person still can spend
- **nodes** Quantity of nodes visited until now

Algorithm 5 Traveling salesperson dynamic approach

```
procedure TRAVELINGSALESPERSON(actualPoint, vPOI, minDistance, maxTime, level, G=(V, E))
    $maxLevel \leftarrow level$
    $path \leftarrow$ vector
    $vPOI[actualPoint] \leftarrow$ set visited true
    $minDistance \leftarrow$ find time spent in this poi

    if time $< 0$ then
        $minDistance \leftarrow \infty$
        return $emptyVector$, level, minDistance        ▷ There is no time left to visit this POI

    $nodes \leftarrow nodes + 1$
    if available = 1 then return $emptyVector$, level, minDistance        ▷ There is no more POIs to visit

    for i from 0 to |vPOI| − 1 do
        if vPOI[i] not visited then
            $actualDistance \leftarrow 0$
            $auxLevel \leftarrow level$
                                        ▷ Calculating the available time left to spend
            $srcPointIndex \leftarrow$ index of vPOI[actualPoint] in V
            $destPointIndex \leftarrow$ index of vPOI[i] in V
            $timeActualPOI \leftarrow$ find time spent in vPOI[i]
            $timeLeft \leftarrow maxTime - dist[srcPointIndex][destPointIndex] - timeActualPOI$

            $tempVector, nodesReached, actualDistance \leftarrow travelingSalesperson(i, vPOI,$
    $actualDistance, timeLeft, auxLevel)$

                                        ▷ Update the actual distance
            $actualDistance \leftarrow actualDistance + pred[srcPointIndex][destPointIndex] + timeActualPOI$

            if (minDistance > actualDistance and auxLevel >= maxLevel)
    or (auxLevel > maxLevel and actualDistance $\neq \infty$ ) then
                $path \leftarrow tempVector$
                $minDistance \leftarrow actualDistance$
                $maxLevel \leftarrow auxLevel$
                $nextPOI \leftarrow i$

        if nextPOI ! = -1 then
            $srcPointIndex \leftarrow$ index of vPOI[actualPoint] int V
            $destPointIndex \leftarrow$ index of vPOI[nextPOI] in V
            $level \leftarrow maxLevel$

            $floydPath \leftarrow getFloydPath(srcPointIndex, desPointIndex)$
            $floydPath \leftarrow floydPath[0...|floydPath| - 1] + path[0...|path| - 1]$        ▷ Join the two vectors

             return $floydPath, level, minDistance$

         return $emptyVector, level, minDistance$
```

Algorithm Analysis

Floyd-Warshall

The time complexity for Floyd-Warshall is $O(|V^3|)$ and its space complexity is $O(|V^2|)$ as discussed before.

Greedy Approach

- The time complexity of the algorithm is:

$$O(|V^3|) + O(|V|) * O(|V * log(V)|)$$

  giving the final complexity of $O(|V^3|)$

- The space complexity of the algorithm is:

$$O(|V^2|) + O(|V|) * O(1)$$

  giving the final complexity of $O(|V^2|)$

- The empirical analysis provided the following minimum, maximum and average times in ms after running the algorithm with 4x4, 8x8 and 16x16 grids passing by all nodes:

	4x4	8x8	16x16
Maximum	180	342	1503
Average	83	164	605
Minimum	36	78	332

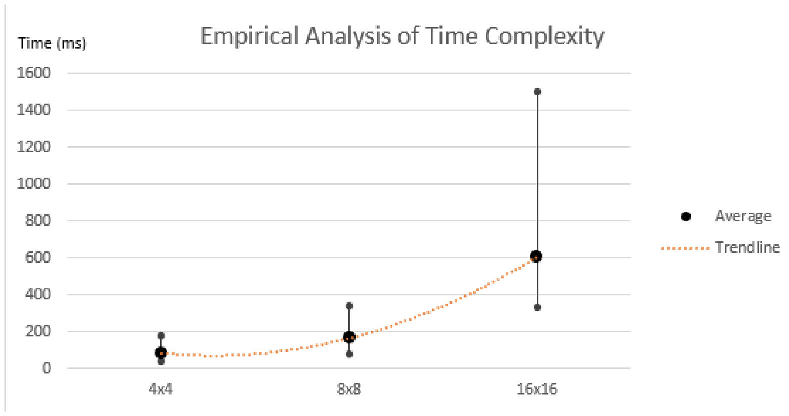


Figure 1: A graphic done with the table information showing also the trendline.

Dynamic Approach

- The time complexity of the algorithm is:

$$O(|V^3|) + O(|V|) + O(|V * V!|)$$

  giving the final complexity of $O(|V * V!|)$

- The space complexity of the algorithm is:

$$O(|V^2|) + O(1) * O(1)$$

  giving the final complexity of $O(|V^2|)$

Data Structures

Nodes.csv

The *nodes.csv* file is a simple table of one column with all the nodes in the graph identified by it's id.

Edges.csv

- S_i: Source, the id of the source node.
- T_i: Target, the id of the target node.
- L_i: Length of the road (in meters).
- $OSMid$: the id of the edge in the Open Street Map.
- $Name$: name of the street in which the node belongs.
- O_i: Oneway, if the street is just one way.
- M_i: Maxspeed, the maxspeed of the road (in kilometres per hour).

POI_cityName.txt

Every city has a file called POI_<cityName>.txt

- $Name$: Name of each point of interest.
- ID: ID of the POI in the vector V of the Graph.
- $Time$: Time in minutes spent in the POI

Conclusion

The goal of this project was to develop a solution of the problem of route planning in a city sightseeing context.
For that purpose we resort to shortest-path and optimization algorithms as well as dynamic programming.
Finding a solution for choosing the next POI was hard and the greedy approach contains the fastest solution, however it's not the best one. By the other side, the dynamic approach finds the best solution, but it's very slow .
So, we decided that the best approach for our scenario would be: use the dynamic approach when the number of possible POI's is not much and the greedy approach when it's number is high.
The participation of each person is:

- Alexandre Abreu - 33%
  - Algorithm
  - Performance Tests
- Eduardo Correia - 33%
  - Input Data
  - Graphical Interface
- Juliane Marubayashi - 33%
  - Input Data
  - Algorithm
  - Data Storage

References

[1] Boeing, Geoff, *OSMNnx*, Github repository, https://github.com/gboeing/osmnx

[2] *Open Street Maps*, https://www.openstreetmap.org/

[3] *Pandas*, https://github.com/pandas-dev/pandas

[4] Cormen, Thomas H., and Thomas H. Cormen. 2001. *Introduction to algorithms*. Cambridge, Mass: MIT Press.

[5] Sambol, Michael, *Floyd-Warshal Algorithm in 4 minutes*, https://www.youtube.com/watch?v=4OQeCuLYj-4