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
April, 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 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. In order to describe the output graph, consider $\lambda(i, j)$ as the direct cost between vertexes i and j and w_{ij} as the weight between these two, with the following formal definition:

$$\lambda(i,j) = \begin{cases} \text{inf} & \text{if i and j are not directly connected} \\ w_{ij} & \text{if i and j are directly connected} \end{cases} \tag{1}$$

- 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, $Adj_i \subseteq E$, where $(j \in V \wedge j \neq i \wedge \lambda(i,j) < \text{inf}) \rightarrow (j \in Adj_i)$.

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.

- $f = \sum_{t_{POI}}^{T} \Delta\lambda$, the total time spent in the POIs
- $g = \sum_{t_{travel}}^{T} \Delta\lambda$, the total time spent travelling

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:

- Different types of locomotion: bus, on foot and car.
- The client can choose the points of interesting

Solution Description

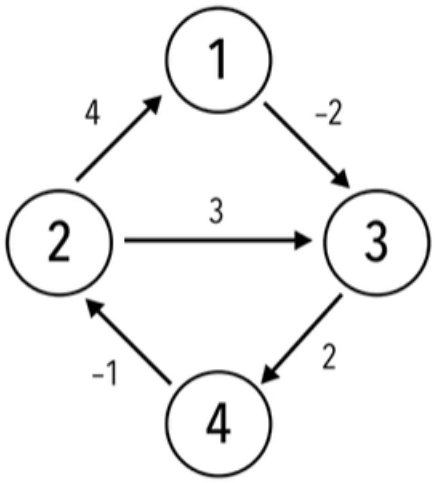
Data Treatment

Algorithms

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

Algorithm 1 Floyd-Warshall

```
1:  procedure SHORTHEST PATH(G=(V, E))
2:      dist ← N × N matrix
3:      for each vertex v ∈ dist do
4:          dist[i][i] ← 0
5:      for each edge w ∈ dist do
6:          dist[u][v] ← weight(u, v)
7:          pred[u][v] ← origin(w)
8:      for k from 1 to V do
9:          for i from 1 to V do
10:             for j from 1 to V do
11:                 if dist[i][j] > dist[i][k] + dist[k][j] then
12:                     dist[i][j] = dist[i][k] + dist[k][j]
13:                     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 ($i \to k \to j$, which can simply described by the recursion formula:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if k} = 0 \\ min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{ki}^{k-1}) & \text{if } k \ = 1 \end{cases} \tag{2}$$

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 $k > 1$:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{k-1} & \text{if } d_{ij}^{k-1} <= d_{ik}^{k-1} + d_{kj}^{k-1} \\ \pi_{kj}^{k-1} & \text{if } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1} \end{cases} \tag{3}$$

For $k = 0$ the definition will be:

$$\pi_{ij}^{(0)} = \begin{cases} NIL & \text{if } w = \text{inf} \vee \text{i} = \text{j} \\ i & \text{if } i \neq j \wedge w < \text{inf} \end{cases} \tag{4}$$

To complete the algorithm analysis, the spacial complexity for this algorithm should be $O(|v^2|)$, since it uses two auxiliary N $\times$ N matrixes.

In order to make the waiting time less significant, improvements to this algorithm will be considered.

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).

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.

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