Inter-Modal Transportations

João Carvalho, José Machado, Renato Campos

***Abstract*—This paper aims to address the problem of finding the best route between two points in a inter-modal transport system. By inter-modal we mean, a transport system which can use Metro, Bus and also walking. Later in this paper we will focus on the efficiency of the algorithms used.**

I. Preface

We are students from FEUP, Porto, doing an integrated five years master degree in Informatics and Computing Engineering. This project was developed to be presented in the course unit: Algorithm Design and Analysis, from the second year of the course.

II. Introduction

Nowadays there are already transport companies that offer to their customers services that enable them to find the best way to get from on place to another, using their transport net, like for example STCP from Porto.

In this project we will try to create a robust system, capable of giving to the user a good pack of "best ways" to go from A to B. The user will be able to get any of this services:

In order to accomplish this tasks, we decided to create a graph structure using C++; Our graph contains a collection of Nodes, Edges, and Transport Lines. A Node represents a subway station or a bus stop and has coordinates in the plane xOy, and a collection of Edges that start in that node. An Edge represents a connection between two nodes and has it's destination node, a weight that will always be the distance in meters, and the transport line associated with the edge. A Transport Line represents a set of the real life lines that cross for example a street. It holds both the initial and the final edges. Furthermore, it contains a name (for example: street name), as well as a set of the real lines that use that Transport Line. That way we can know for example which bus to catch in a certain Node. Finally a Transport Line has information about the type of transports that use it, if it is bidirectional or not, and an average waiting time (seconds);

III. Problem Formal Definition

A. Input

The input will be a graph that abstracts the transport network. We will use real-life scenarios to test our implementations of the algorithms. The data is obtained from [1], and parsed to .txt files that are then read by our application.

B. Output

The output of the graph, will be a set of nodes that are considered to belong to the optimal path. In this project, we aimed to allow the user to define the optimal path between n_i to n_j as:

- Shortest Path;
- Fastest Path:
  1) Using the awt_i;
     a) Select a favorite transport;
     b) Select maximum cost for the route;
  2) Not using the awt_i
     a) Select a favorite transport;
- Minimize the swapping between different transports.

We decided to not include the cost calculation when calculating the fastest path without average waiting time, because it does not simulate very well a real case scenario, so there's no need to introduce complexity. With our implementations we can also select to do not favor any mean of transport or to not use the maximum cost in the calculations. We have designed our code so that, when using the awt_i, selecting a favorite transport still allows to select a maximum cost for the route.

Therefore, there we can define 3 objective functions.

C. Objective Functions

- Minimize the traveled distance:

$$\sum_{n_{start}}^{n_{end}-1} d_{k,k+1}, k \in A_{n_{start},n_{end}}$$

- Minimize the traveled time with a favorite transport and a maximum cost:

$$\sum_{n_{start}}^{n_{end}-1} t_{k,k+1}, k \in A_{n_{start},n_{end}}$$

- Minimize the swapping between different transports:

$$\sum_{n_{start}}^{n_{end}-1} c_{k,k+1}, k \in A_{n_{start},n_{end}}$$

D. Simplifying the problem

1. We wanted to allow the traveler to walk both through roads and bus lines (as these are just roads), but not through metro rails. In order to take into consideration the average waiting times, we first make a copy of the original graph, then we preprocess the copy, making a directed edge between every node of every transportation line. Then, after getting the best path, we destroy the copied graph. We have to do this, because when we are

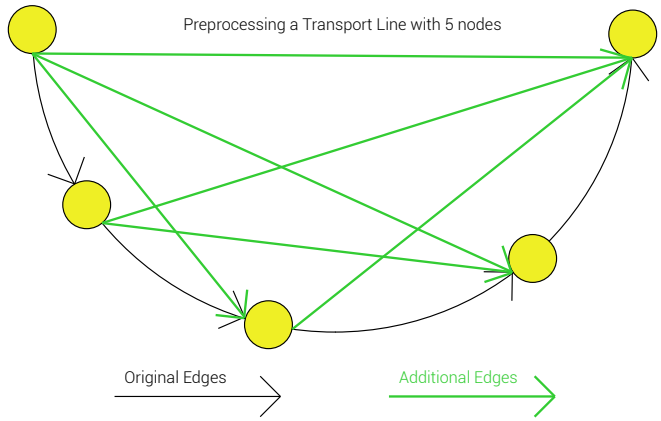

Fig. 1. Preprocessing a transportation in order to use average waiting times

trying to find the best path, we save in each node which type of transport line was used to get to it. By type of transport line, we mean : Bus, Metro, or Walk; This way, we can detect when a traveler is changing between transport lines and add an additional cost to the traveling time - the average waiting time;

IV. Algorithms

Algorithm 1 Preprocess Graph

```
1:  procedure Graph::preprocess()
2:      Let tlNodes_{i,j} be the set of nodes associated with a transport line
3:      for all tl_{i,j} ∈ TL do
4:          tlSize ← Size_{tlNodes_{i,j}}
5:          if type_{tl_{i,j}} = bus And tlSize ≥ 3 then
6:              for all s = 0 To tlSize − 2 do
7:                  n_src ← tlNodes_{i,j}.at(s)
8:                  weight ← 0
9:                  for all d = s To tlSize − 1 do
10:                     weight += weight_{Edge(initEdge+d)}
11:                     if s ≠ j then
12:                         highestEdgeId + +
13:                         Createe_{s_d}
14:                         SetTransportLine_{e_{s_d},tl_{i,j}}
15:                         B ← B ∪ e_{s_d}
```

A. Complexity Analysis

- Preprocess Graph: $\Theta(|TL| * |N|^2)$
- Dijkstra: $\Theta(|E| * \log(|N|))$
- SPFA: $\Theta(|E| * \log(|N|))$

B. Improvements

- Ellipse - "Cutting the Graph" 2. We only used the ellipse technique to "cut" the graph when the objective function is to minimize the distance, because in other situations we may be loosing valuable information.

Algorithm 2 Minimum distance Dijkstra

```
    procedure dijkstraMinDistance(n_s)
2:      Let PQ be an empty heap
        Let Adj be a set of adjacent edges
4:      for all n_i ∈ N do
            path_{n_i} ← null
6:          processing_{n_i} ← false
            dist_{n_i} ← ∞
8:      dist_{n_s} ← 0
        PQ ← n_s
10:     while PQ ≠ ∅ do
            n_s ← PQ.front; PQ.pop_back
12:         Adj ← Adj_s ∪ Foot_s
            for e_{s,w} ∈ Adj.dest do
14:             if dist_{n_w} > dist_{n_s} + w_{s,w} then
                    dist_{n_w} ← dist_{n_s} + w_{s,w}
16:                 path_{n_w} ← n_s
                    if !Processing_{n_w} then
18:                     Processing_{n_w} ← true
                        PQ.push_back(n_w)
20:             n_i.processing ← false
                n_i.dist ← ∞
22:         Reorder_PQ
```

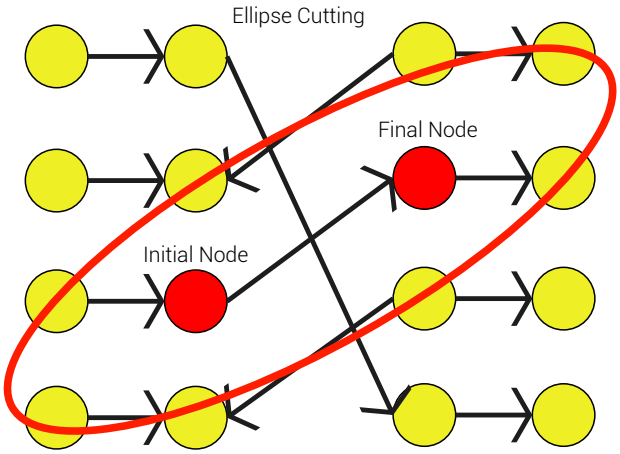

Fig. 2. "Cutting" the graph using an ellipse

In order to use the ellipse, we first need to send the destination node as an argument to the algorithm. Secondly, we calculate the rotation matrix to multiply by the nodes Coordinates. Lastly, if :

$$\frac{xRotated - xCenter}{semiH} + \frac{yRotated - yCenter}{semiV} \leq 1$$

Then the node is within the ellipse and can be added to the priority_queue.

- Multi-threading Several implementations of multi-threading in Dijkstra algorithms can be found by searching for "multi-threading Dijkstra". Although we tried to implement it, our lack of knowledge, but

Algorithm 3 Minimum time Dijkstra

procedure DIJKSTRAMINTIME($n_s, max_cost, favoriteT$)
  Let PQ be an empty heap
  Let Adj be a set of adjacent edges
  for all $n_i \in N$ **do**
    $path_{n_i} \leftarrow null$
    $processing_{n_i} \leftarrow false$
    $dist_{n_i} \leftarrow \infty$
    $wayToGetHere_n n_i \leftarrow Walk$
  $dist_{n_s} \leftarrow 0$
  $PQ \leftarrow n_s$
  while $PQ \neq \varnothing$ **do**
    $n_s \leftarrow PQ.front; PQ.pop_back$
    $Adj \leftarrow Adj_s \cup Foot_s$
    for $e_{s,w} \in Adj.dest$ **do**
      $cost \leftarrow 0$
      $n_w \leftarrow destNode_{e_{s,w}}$
      $tl \leftarrow tLine_{e_{s,w}}$
      $typeTransport \leftarrow type_tl$
      $onTransport \leftarrow true$
      if $wayToGetHere_{n_s} = Walk$ **then**
        $onTranports \leftarrow false$
      $cost \leftarrow$ CALCULATECOST$(e_{s,w}, tl)$
      $summedCost \leftarrow cost_{n_s} + cost$
      if $summedCost > max_cost$ **then**
        $deltaTime \leftarrow \infty$
      else if $typeTransport = Walk$ **then**
        $deltaTime \leftarrow w_{s_w}/W_{speed}$
      else if $typeTransport = Bus$ **then**
        if $onTransport$ **then**
          $deltaTime \leftarrow w_{s_w}/B_{speed}$
        else
          $deltaTime \leftarrow w_{s_w}/B_{speed} + awt_tl$
        if $w_{s_w}/W_{speed} < deltaTime$ **then**
          $typeTransport \leftarrow Walk$
      else if $typeTransport = Metro$ **then**
        if $onTransport$ **then**
          $deltaTime \leftarrow w_{s_w}/M_{speed}$
        else
          $deltaTime \leftarrow w_{s_w}/M_{speed} + awt_tl$
      $realTime \leftarrow deltaTime$
      if $typeTransport = favoriteT$ **then**
        $deltaTime \leftarrow 0.01(deltaTime)$
      if $dist_{n_w} > dist_{n_s} + deltaTime$ **then**
        $dist_{n_w} \leftarrow dist_{n_s} + realTime$
        $path_{n_w} \leftarrow n_s$
        $wayToGetHere_{n_w} \leftarrow typeTransport$
        $cost_{n_w} \leftarrow summedcost$
        if $!Processing_{n_w}$ **then**
          $Processing_{n_w} \leftarrow true$
          $PQ.push\_back(n_w)$
      $n_i.processing \leftarrow false$
      $n_i.dist \leftarrow \infty$
    $Reorder_{PQ}$

Algorithm 4 Short Path Faster Algorithm

procedure SPFAMINDISTANCE(n_s)
  The graph should be stored in a Adjacency List
  Let vi_{dist} be a vector<int> to store the distances for each node
  Let q be a queue<int> to store the vertex to be processed
  Let $vi_{in.queue}$ be a vector<int> to the denote if a node is the queue or not
  $vi_{dist}(n, \infty)$
  $dist[n_s] \leftarrow 0$
  $queue < int > q; q.push(n_s);$
  while $PQ \neq \varnothing$ **do**
    $intu = q.front; PQ.pop\_back; in_{queue}[n_u] = 0;$
    $Adj \leftarrow Adj_s \cup Foot_s$
    for $j = 0; j < AdjList[n_u].size; j++$ **do**
      int $v = AdjList[u][j].first;$
      int $weight_{u,v} = AdjList[u][j].second;$
      if $dist[v] > dist[u] + weight_{u,v}$ **then**
        $dist[v] \leftarrow dist[u] + weight_{u,v}$
        if $!in_{queue}[v]$ **then**
          $q.push(v)$
          $in_{queue}[v] = 1$

mainly time, didn't allowed us to get any to work. Anyway, it would be interesting in future years to see second year students implementing multi-threading in this project.

V. Empirical analysis

A. Optimizing Distance - Dijsktra and Ellipse "Cutting"

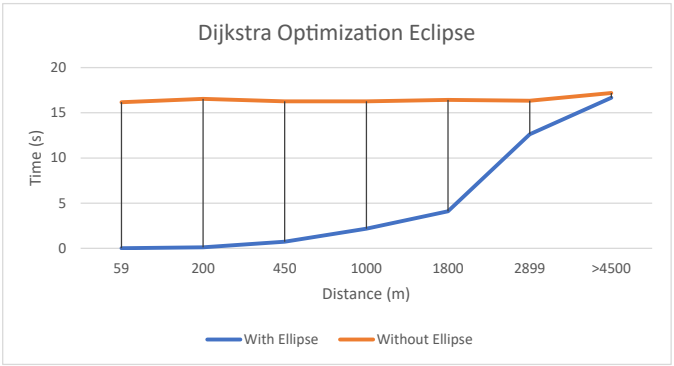

Fig. 3. "Dijkstra Optimized by Ellipse"

In the Fig. 3. there is a big difference between with and without the ellipse optimization for small distances because the graph to analyze it is much smaller because only an ellipse area with the source and destination Nodes as extremes is analyzed but for a bigger distance the difference will be less noticed or even null because from a value the graph will be all analyzed.

B. Optimizing Distance - Dijkstra vs SPFA

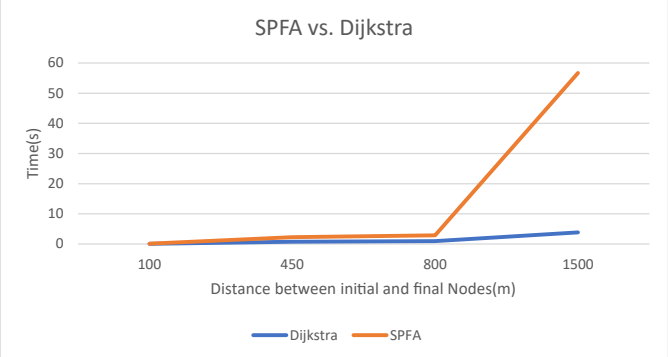

Fig. 4. "Comparing SPFA vs Dijsktra's Running Time"

In the Fig. 4. we can see that the SPFA have about the same time of the execution of the Dijkstra, for small distances between initial and final Nodes, but at certain point the difference between SPFA and the Dijkstra's is huge, and SPFA is a lot, lot slower that the Dijkstra's.

C. Optimizing Time - Dijkstra minimizing time

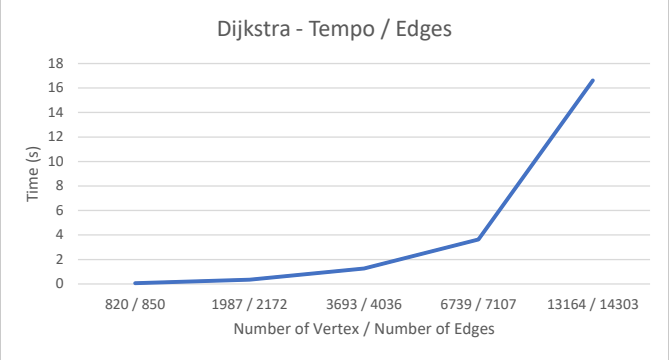

Fig. 5. "Dijkstra running time when calculating favorite transport, max cost and waiting times"

VI. Main Difficulties

- The parser was way to complex and it took too long to modify to correspond to our necessities;
- Initially it was difficult to define the design of our program because the bus stops were independent nodes that were not connected to anything, so we have considered that every node in a bus or metro transportation line is a bus stop or metro station;
- Reduce the read times of the files because initially, if the map had thousandths of nodes it would take more than 5 seconds to read, even in release mode;
- Report in LaTeX, because it was the first time;

VII. Conclusion

The objective with this work was to try to solve the real problem of planning a trip in an inter modal transportation network. using a graph. The optimal path to go from a Node to another one depends on the variable that we want to minimize which in our approach can be the distance, the time and the number of lines' changing. there are multiple options of minimization that were implemented such as: minimization by time, minimization by time considering the waiting time in each line that is a random number between 3 and 7 minutes, minimization by distance, minimization by distance using an ellipse as improvement, minimization by time having a favorite transport, minimization by time having a favorite transport and a max_cost, etc. The problem of the shortest path was solved with Dijkstra's algorithm, to all the minimizations specified above there is a different algorithm with the necessary changes, it was also implemented SPFA(Shortest Path Faster Algorithm) which in theory should be take as much time as Dijkstra but that was not verified in fact it was multiples times slower. In Dijkstra's algorithm in the case of the shortest path by distance an improvement was introduced to reduce the time of calculation, we tried to introduce another improvement with the same purpose as the ellipse, multi-threads but with the lack of time we could not manage to get it working. It would be interesting implement this work using a neural network.

VIII. Recommendations

After some reflection we recommend to use the parser, although it took too much time to put it 100% working, because it is much more satisfying to test the algorithms in real situations and we recommend trying to use multi threads on Dijkstra's algorithm because it has a lot of potential to reduce the processing time and it allows to use knowledge from other course unit.

IX. Symbols and Notations

Here is a brief explanation of the symbols used in this report:

- G - designates the Graph which abstracts the network;
- N - designates the set of nodes contained by the graph,
- E - designates the set of edges contained by the graph;
- TL - designates the set of transport lines contained by the graph;
- n_i - designates a node which abstracts a metro station, a bus stop or a simply a point in a road;
- x_i, y_i - designates the coordinates of node n_i in the plane xOy;
- $e_{i,j}$ - designates the directed edge that connects n_i to n_j;
- Adj_i - designates a set of all the edges that start in n_i;
- $Foot_i$ - designates a set of edges that are calculated to allow the traveler to walk from n_i to close bus stops or metro stations.
- $w_{i,j}$ - designates the cost to get from n_i to n_j;
- $tl_{i,j}$- designates a transportation line that starts in node n_i and ends in node n_j;

- awt_i - designates the average waiting time associated with transportation line i;
- W_{speed} - designates the velocity chosen to simulate the walking process;
- B_{speed} - designates the velocity chosen to simulate the bus speed;
- M_{speed} - designates the velocity chosen to simulate the metro speed;
- $t_{i,j}$ - designates the time in seconds necessary to cross the edge $e_{i,j}$;
- pm_x - designates the ratio $\frac{meter}{pixel}$ in the x axis.
- pm_y - designates the ratio $\frac{meter}{pixel}$ in the y axis.
- $d_{i,j}$ - designates the length of the edge $e_{i,j}$. It is calculated by : $\sqrt{(pm_x(x_i - x_j))^2 + (pm_y(y_i - y_j))^2}$
- $c_{i,j}$ - designates the number of times that the passenger changed lines, when crossing edge $e_{i,j}$, can be $0||1$;
- $path_{v_i}$ - designates the node which precedes v_i in the optimal path;
- A - designates the set of nodes of an optimal path;
- n_{start} - designates the initial node in a path;
- n_{end} - designates the destination node in a path;

References

[1] "Open Street Maps", http://openstreetmaps.org
[2] Halim, Steven, and Felix Halim.*Competitive Programming 3.* 3rd ed. 2013. The new lower bound of programming constests.