SHORTEST PATH PROBLEM IN INTERMODAL GRAPHS
SOFTWARE AIDS FOR RESOLVING MOBILITY PROBLEMS IN BRUSSELS

Matsvei Tsishyn (IRIDIA, ULB)  matvei.tsishyn@gmail.com
Supervisor: Hugues Bersini
May 2019, VUB-ULB, Current Trends in Artificial Intelligence
Aim of my research:
Create **shortest path algorithms** on the city of **Brussels** that are:
1. **Intermodal** (may contain several modalities in a unique solution/path)
2. **User-adapted** (different users have different solutions)
3. Take into account **all the transports** on Brussels (there is more than 20 private and public transport’s actors on Brussels)

Tasks:
1. Centralize the **data** (roads map and transports) in a unique intermodal graph
2. Create **algorithms** that can make intermodal searches
I. Introduction to shortest path problem

II. Intermodal and user-adapted path search

III. Optimization methods
Shortest path problem is a well known problem of Combinatorial optimization.

From a theoretical point of view, almost all combinatorial optimization’s problems can be solved as following:

1. Compute all possibilities
2. Take the optimal one

Problem: Combinatorial explosion
Example:

For a complete graph with only 20 vertices, there are 17,403,456,103,284,420 possible path between two points.

So, the art of combinatorial optimization is to find « smart » ways to compute the minimal amount of combinations and still find the optimal solution, in order to make the computation possible.
Consider a directed weighted graph \([1] \ G = (V, E, l)\) where \(V\) is the set of vertices, \(E\) is the set of edges and \(l\) is a weight function \(l: E \rightarrow \mathbb{R}\).

For simplicity we can note an edge \(e\) as \((v, v')\) where \(v\) is its start vertex and \(v'\) its end vertex.

A path is a sequence \(P = \{e_1, e_2, ..., e_k\}\) of edges such that the end vertex of \(e_i\) is the start vertex of \(e_{i+1}\) for \(i = 1, ..., k - 1\).

For two vertices \(s\) and \(t\), a \(\{s, t\}\)-path is a path from \(s\) to \(t\).

We can define the length of a path \(P\) as,

\[
l(P) = \sum_{i=1}^{k} l(e_i)
\]
From now, let us suppose that the weight function $l$ is non-negative:

$$l(e) \geq 0 \ \forall e \in E$$

Now, we can define a notion of distance on $V$:
For two vertices $s$ and $t$,

$$d(s, t) = \min \{ l(P) | P \text{ is } \{s, t\} - \text{path} \}$$

**Remark:** from a strictly mathematical point of view, $d$ is not a distance.

**Question:** what condition should we add to be sure that $d$ is a distance?

→ **Shortest path problem:**

Given **two vertices** $s$ and $t$, find a $\{s, t\}$-path $P$ such that:

$$l(P) = d(s, t)$$

In other words, that $P$ is the **minimal length path between $s$ and $t$.**
SHORTEST PATH PROBLEM

Given a (directed) weighted graph $G = (V, E, l)$ with non-negative weights and two nodes $s$ and $t$, we have to find the shortest path from $s$ to $t$.


Intuition:

Explore the graph by growing a ball of nodes with increasing distances from the source node $s$.

Remark: In a more general case (weights can be negative), other theories and algorithms exist (Belleman’s equation [4]).
**SHORTEST PATH PROBLEM**

- Maintain a set $S$ of vertices to which we have found the shortest path

- Maintain a distance label $L(v)$ for each vertex that define the best distance we found so far.

**Dijkstra Algorithms:**

(Step 0) $S = \emptyset, \quad L(s) = 0, \quad L(v) = \infty \forall v \in V \setminus \{s\}$

(Step 1) Chose next vertex to expand:

- Chose $v \in V \setminus S$ which minimizes $L(v)$ (use priority queue)

- Update $S = S \cup \{v\}$

(Step 2) Expand $v$:

- For each edge from $(v, v')$ with $v' \in V \setminus S$, if $L(v') > L(v) + l(v, v')$:

  Update distance: $L(v') = L(v) + l(v, v')$

- Stop if $t \in S$
SHORTEST PATH PROBLEM

(0) Initial graph

(1) Take the initial vertex

(2) Expand its edges

(3) Select next vertice and expand its edges
**SHORTEST PATH PROBLEM**

1. Select next vertex and expand its edges
2. Select next vertex and expand its edges
3. Tree of all shortest paths
4. Backtrack the shortest path
Idea of the proof [3]:
- Consider a subset $S$ of $V$ such that $s \in S$ and the labels $L(v)$ on $S$ are the length of the shortest path from $s$.
- Moreover, the labels $L(v)$ on $v \in S' = V \setminus S$ are the length of the shortest path from $s$ with the condition that all vertices of the path are in $S$ except $v$ itself.
- Then, if we take $v \in S'$ with the smallest label $L(v)$, the set $S \cup \{v\}$ still satisfy the wanted condition ($L(v)$ is the length of the shortest path from $s$).
- Indeed, if we consider $P$ the $\{s, v\}$-shortest path, it will have a first vertex in $S'$ and if this first vertex $v'$ have to be $v$ because $v$ has the smallest label $L(v)$ on $S'$.
- First we take $S = \{s\}$ and we expand it inductively until $t \in S$.
- Since all the labels of vertices in $S$ corresponds to shortest paths, we have found the shortest path from $s$ to $t$.

Question: why we need to suppose non-negative weights?
Since 1959, there were a lot of variations, generalisations and speed-ups for the Dijkstra algorithm:

For example:

- Find the $k$-shortest path between two nodes (Yen’s algorithm) [5]
- Find shortest path between all pairs of nodes (Floyd-Warshall algorithm)
- Various optimisation methods to speed-up the computation time of the algorithm

**Problem:** How can we solve a path search problem that is:

1) Intermodal
2) User-adapted
INTERMODAL PATH SEARCH

Data structure

How to build a model of the transports network of a city that supports intermodal and user-adapted path-search?

- The data structure is a **directed** graph \( G = (V, E) \) which is **embedded** in \( \mathbb{R}^2 \) (latitude and longitude).

- **Roads** → **Edges**  
- **Crosses** → **Vertices**

- Each edge \( e \) contain a **vector of parameters** (length, slope, maximal speed in car,...) instead of a unique weight
INTERMODAL PATH SEARCH

Data structure
The graph is organized by layers which represents different types of transports:
- Each node is included in one layer
- Two types of edges:
  - transition edges
  - internal edges
- But several mobility agent can share the same layer
**INTERMODAL GRAPH**

Data structure

- The *main layer* corresponds to ways by foot
- All transition links go **from the main layer** to another layer or conversely
- Each path **starts** and **ends** in the main layer
INTERMODAL GRAPH

Data acquisition

Layers data (vertices and internal edges):

1) **OpenStreepMap** with the API **Overpass Turbo** ([https://overpass-turbo.eu](https://overpass-turbo.eu))
   - OSM has tree types of objects: **nodes**, **ways** and **relations**
   - Objects have **parent/children relations** between them
   - Objects has **tags** organized in **key/values**

   → Export all **ways** with **highway tag** and their **children nodes** in a given region

2) **Local Public database**: Uribs ([http://urbisdownload.gis.irisnet.be](http://urbisdownload.gis.irisnet.be))
   - Some data is more accurate but do not have efficient API
**Data acquisition: OSM Overpass Turbo, example of query**

```plaintext
// STEP 5: Initial Settings -----------------------------------------------
// 0.1) Define output format
// 0.2) Maximal time of execution
// 0.3) Rectangle area (lat/lon) where the query is performed

[out:json][timeout:180] [bbox:50.63,4.33,50.86,4.37];

// STEP 1: find all ways in the Region -----------------------------
// 1.1) Find all 'way' objects with the tag 'highway' of any value
// except the one that have the tag 'area'
// Inside the polygon defined due to its lat/lon coordinates

// 1.2) Stock all these objects in an object .myWays

{ way[highway] [area] { poly: 50.634931 4.3402105 50.8499552 4.3355559 50.0591947 4.3450474; 50.0536597 4.3686861 50.8470796 4.3669975 50.8403232 4.3671914 50.6326145 4.3467327; }; }

// STEP 2: expand their child nodes -----------------------------------
// Remark: way objects are not located in space if we do not refer to the nodes that
// they process

// 2.1) Recall the object .myWays
// 2.2) Access all of its children objects that are of type "node"
// 2.3) Stock them in the object .myNodes

.myWays: node(w) -> .myNodes;

// STEP 3: take the union ---------------------------------------------
// 3.1) Take the union of .myWays and .myNodes
// 3.2) Save it in the object .all

{ .ways: .nodes:1 -> .all; }

// STEP 4: Print/export the data ---------------------------------------
// 4.1) Export the object .all
// 4.2) Precise how detailed is the exported data with 'meta'

.all out meta;
```
**INTERMODAL GRAPH**

*Data acquisition*

OSM data is **not** organized as a graph => need **preprocessing** algorithms

Row OSM data

Graph from OSM data
Data acquisition

**Transition** between layers (transition edges):

Data from each **actor of transports**
- Some data is on OSM or Urbis (as the Villo stops)
- Use different API for each actor of transports (open or private)
INTERMODAL GRAPH

Data acquisition

**Reunite** all data in a **unique multi-layer graph** structure
- All informations are stocked in the **parameter vector** of edges

**Challenge:**
How to **reunite** the data in the best/more realistic way?
How to **link a Villo station** to the rest of the network?
→ need algorithms of **computations geometry**
INTERMODAL GRAPH

Algorithm
Use any classical shortest path algorithms (any variation of the Dijkstra algorithms) with some additional details:

The cost of a edge is not a unique constant weight, but is defined by a cost function depending on the parameters of the link, the layer, the user, the environmental condition and the parameters of the search.

- This allows us to find shortest paths depending on,
  - User’s preferences and possibilities
  - Environmental conditions

- This also allows us to:
  - Find several shortest paths that use different modalities
  - Find shortest path with different balances of time, cost and ecological impact
INTERMODAL GRAPH

Challenges:
1) How to measure the «real cost» of a road
   - Theoretical model
   - Learning
2) How to make the graph geometrically accurate?
   - Data from different sources may be slightly different
   - Some data may be missed
   - All the reality of a road network can not be stored in data bases
3) How to optimize this kind of algorithms on this type of graph topology?
OPTIMIZATION METHODS

Heap
To choose the next vertex to expand, the Dijkstra algorithm have to take the vertex with the minimal distance tag $L(v)$ which is a priority queue that is constantly updated and may become very long.

The use of heap (binary or Fibonacci [6], [7]) can largely reduce the computation time of the algorithms.

Complexity:
- Dijkstra with list: $O(V^2)$
- Dijkstra with binary heap: $O(V \log_2(V))$

Idea:
To extract the minimum value of a priority queue, we only have to partially order the queue (not totally)

<table>
<thead>
<tr>
<th>data structure</th>
<th>insert</th>
<th>remove maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>heap</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Order of growth of worst-case running time for priority-queue implementations

Image from [7]
OPTIMIZATION METHODS

Binary heap:
A structure of binary tree where each child is greater or equal to its parent. So the largest element is found on the root of the tree.

Remark: the structure works as well with the reversed order: largest => smaller
**Binary heap:**
The binary heap structure can be stored in an **array**. For that we store the elements starting at 1 represented in **depth-level order** in the array. This allows easy **access to parents and children**:

**Children** of element at position $p$: $2p$ and $2p + 1$

**Parent** of element at position $p$: $\lfloor p/2 \rfloor$

**Exercise:** prove it
OPTIMIZATION METHODS

Binary heap: Preliminary operations

- Swim:
When an element violates the heap order because it is larger than its parent (it is too low in depth-level of the tree).
  - Exchange it with its parent element
  - Repeat until the heap order is restored

- Sink:
When an element violates the heap order because it is smaller than (at least) one of its children (it is too high in depth-level of the tree).
  - Exchange it with the largest child
  - Repeat until the heap order is restored

Question: Why swim and sink restore the heap order and do not create other heap order violations?
**OPTIMIZATION METHODS**

**Binary heap:**

**Insertion:**
- **Insert** the new element in the **end** of the array
- « **Swim** » the element until the heap order is restored

**Remove maximal element:**
- **Remove** the **first element** of the array (**root** of the tree)
- Put the **last element** of the array on the **first place** (on the **root**)
- « **Sink** » the element until the heap order is restored

**Question:** See that the complexity of these operations is \( O(\log(n)) \) where \( n \) is the length of the queue.

**Question:** In Dijkstra algorithm, what to do when the distance label \( L(v) \) is updated?
OPTIMIZATION METHODS

Bidirectional search ([9], [12])
- One **forward** search from the **source** node $s$
  $\rightarrow$ On the **original graph**
- One **reversed** search from the **target** node $t$
  $\rightarrow$ On the **reversed graph**
- **Alternate** the two searches until the two searches « **meets** » at a vertex $m$.

- **Reconstruct the path:**
  - $P_f$: shortest $\{s, m\}$-path
  - $P_r$: shortest $\{t, m\}$-path
  - $P$, the shortest $\{s, t\}$-path is reconstructed as $P_f + rev(P_r)$
Bidirectional search

Problem:
When the two sets $S_f$ and $S_r$ of the respectively forward and reversed searches intersects, the optimality is not guaranteed.

→ Need for a stopping condition
OPTIMIZATION METHODS

Bidirectional search

Stopping condition:
Keep track of the forward and reversed distance label as $L_f(v)$ and $L_r(v)$

Define $\mu$ as the length of the minimal path founded so far as:
When an edge $e = (v, w)$ is scanned (when $v$ is included in $S_f$) in the forward search and $w \in S_r$, we update $\mu$ if $L_f(v) + l(v, w) + L_r(w) < \mu$ and conversely.

Stop when $top_f + top_r \geq \mu$

Where $top_f$ and $top_r$ are the minimal value of the priority queue of respectively the forward and the reversed search.
Bidirectional search

Stopping condition: Idea of the proof

Suppose we have searched until stopping condition but there is a path $P$ of length $\mu' < \mu$.

So there is an edge $e = (v, w)$ in $P$ such that:

$$d(s, v) < \text{top}_f \quad \text{and} \quad d(w, t) < \text{top}_r$$

Indeed, $\mu' = d(s, v) + l(v, w) + d(w, t) < \text{top}_f + \text{top}_r$

This means that $v$ is already scanned in the forward search and $w$ is already scanned in reversed search.

So, when the second of them was expanded, the edge $e$ was scanned and so the path $P$ should have been found.
OPTIMIZATION METHODS

Bidirectional search

For road networks:
Search space: \[ D_1 \sim \pi R^2 \implies D_2 \sim 2\pi (R/2)^2 \]
→ Decrease the search space by 2

For search with constant branching factor \( b \):
Search space: \[ D_1 \sim b^d \implies D_2 \sim 2 b^{d/2} \]
→ Decrease the search space by \( \frac{1}{2} b^{d/2} \)

The efficiency of the optimization method largely depends on the topology of the graph!
OPTIMIZATION METHODS

The following optimization methods use **preprocessing** techniques or **external informations** about the graph.

**Naive preprocessing**: Compute **all shortest pathes** between all points and then just give the answer by accessing it in the memory.

**Problem**: in most of the **modern** applications handle **networks** are **too large** to allow it in terms of **memory** and **computational time**.
A* [8]

Idea:

Use **heuristic information** to determine **which vertex explore first**

Intuitively, if the vertex $t$ is on the right of the vertex $s$, it may **not be a good idea** to search far on the left of $s$.

In this case, the **heuristic** can be considered as a **prediction for the distance** from a vertex $v$ to the target vertex $t$. 
Consider a potential function \( \pi : V \to \mathbb{R} \) such that \( \pi(t) = 0 \).

We say that \( \pi \) is feasible if:

\[
\forall u, v \in V \quad d(u, v) - \pi(u) + \pi(v) \geq 0
\]

In other words, that \( \pi \) never overestimates \( d \).

Then, redefine a new edge weight

\[
l_\pi(u, v) := l(u, v) - \pi(u) + \pi(v).
\]

A* algorithm: Dijkstra algorithm on the same graph \( G = (V, E) \) but with new weights \( l_\pi \).
**OPTIMIZATION METHODS**

**A*: Idea of the proof**

By the fact that the heuristic $\pi$ is **feasible**, the new weighted graph $G_\pi = (E, V, l_\pi)$ has **non-negative weights** (why?), which is required for the optimality of the Dijkstra algorithm.

All the $\{s, t\}$-paths lengths are **changed** by the same **constant** value,

$$l_\pi(P) = l(P) - \pi(s) + \pi(t) = l(P) - \pi(s)$$

So a shortest $\{s, t\}$-path in $G_\pi$ is also a **shortest path** in $G$.

Since the A* algorithm on $G$ is a Dijkstra algorithm on $G_\pi$, it give the shortest path on $G$. 
OPTIMIZATION METHODS

A*

To note:
- The A* algorithm **reduce the search space** $S$ of a run and so reduce the computational time.
- The aim of the heuristic $\pi(v)$ is to give a **lower bound** to $d(v, t)$.
- The **closer** is the **lower bound** the **better** the A* will work

Remark:
In **roads networks** for routing algorithms, we can for example take as heuristic $\pi$ the **straight-line distance** to $t$ with the **maximal** possible **speed** on this region. (Why maximal?)
OPTIMIZATION METHODS

A* with Euclidian distance as $\pi$ vs. Dijkstra

Dijkstra: 126 explored vertices
A*: 17 explored vertices
**OPTIMIZATION METHODS**

**Landmarks** [10]

- Select $k$ landmarks $(l_1, ..., l_k)$: vertices for which we compute distances to and from all other vertices.
- For each landmark $l_i$, each vertex $v$ and the target vertex $t$, we can compute two independent triangle inequalities:
  
  $$d(v, t) \geq d(l_i, t) - d(l_i, v) \quad \text{and} \quad d(v, t) \geq d(v, l_i) - d(t, l_i)$$

- Use it as a **lower bound for the distance** $d(v, t)$:
  
  $$d(v, t) \geq \max_{i=1,\ldots,k}\{d(l_i, t) - d(l_i, v), \; d(v, l_i) - d(t, l_i)\}$$

- Use this lower bound as **heuristic for A***
Landmarks

Landmarks can **considerably reduce the search space** $S$. The **better** is the **lower bound** (for $d(v, t)$), the **more** they it **reduce the search space**.

$\rightarrow$ We have to found landmarks that give good **lower bounds**.

At one **run**, select only a **few landmarks** and not all, otherwise the computations of the maximal bound will be too long.

**Non-trivial question**: how to **find** the **best possible landmarks** (the one who reduce in average the most the search space)
OPTIMIZATION METHODS

There are many others optimization methods for shortest path problem and it remains an active field of research.

Two things to note:

1. Their efficiency depends on the topology of the graph
2. They often require preprocessing
Problem:
- Speed-up techniques require **preprocessing**
- The graph is **intermodal** and **user-adapted**

→ **Preprocessing is very challenging** (different users have different shortest paths)

How can we make preprocessing for each user and in intermodal graph?
- **Several preprocessings**?
- **Generalize preprocessing** that keep flexible information?
FURTHER QUESTIONS AND RESEARCH

1) **Which** speed-up use?
   - Their **efficiency** depends on the **topology** of the graph
   - The intermodal graph has its **specific topology**
     → **Which speed-up technique** is the better to use?

2) **How** to use it?
   - **Preprocessing** is very **challenging**
     → How to **adapt current speed-up** techniques to intermodal graph?
THANK YOU
REFERENCES