UNIVERSITÉ LIBRE DE BRUXELLES

Faculté des Sciences

Master thesis in Computer Sciences

# A*, Landmarks, and Triangle inequality (ALT) shortest path speedup technique for Bi-modal User-adapted road networks

Alexandre HENEFFE

*Promotor*
Prof. Hugues Bersini

*Supervisor*
Matsvei TSISHYN

Academic year 2020 - 2021

*"Le génie c'est 1% d'inspiration et 99% de transpiration."*

**EDISON**

*"Somewhere, something incredible is waiting to be known."*

**CARL SAGAN**

*"In the beginning there was nothing, which exploded."*

**TERRY PRATCHETT**

# Abstract

The shortest path problem on road networks has become a well-studied field and there exist many simple algorithms that solve it. The recent increasing amount of data forces us to find new techniques in order to considerably decrease their execution times. Several prominent speed-up techniques have emerged in the latest years. Besides, multi-modal routing has also been developing and brings new challenges compared to single-modal routing. In particular, there is an incentive to incorporate user preferences. In this thesis, we focus on ALT (A*, Landmarks and Triangle inequality) heuristic-preprocessing-based speed-up technique. We apply this speed-up technique on multi-modal user-adapted road networks and conduct some experiments on multiple scenarios using real road networks data. The goal is to determine if this technique is still applicable to such networks without drastically losing performance. The results show that ALT query times are not greatly impacted but at the expense of a little drop in performance in terms of preprocessing times. These findings are similar on bi-modal user-adapted road networks.

# Acknowledgements

I would like to thank all the people who supported and surrounded me throughout the creation of this master thesis, which concludes my master in computer sciences and 5 years of hard work.

First, I would like to thank Prof. Hugues Bersini who offered me the opportunity to work on this topic.

Secondly, I thank Matsvei Tsishyn, who supervised parts of my work and provided precious help as well as interesting leads.

Thirdly, I would like to thank my fellow comrades/colleagues/friends, namely Nicolas Jonas, Ricardo Gomes Rodrigues and Simão (Simon) Morais who have accompanied me during these 5 years at the university (ULB) and with whom I have worked during a tremendous amount of time. They gave me a lot of motivation.

Fourtly, I thank my family (my parents and my siblings) and friends outside the University (namely, Maxime Buelen, Alexandre Chapelle, Nathan Patesson and Jean-Baptiste Vauthier) who always supported me morally, especially during the realization of this thesis.

Lastly, but not least, I would like to thank my professors at the "Université Libre de Bruxelles" who taught me a lot of things and who helped me develop essential skills.

Special thanks to Victor Goossens who kindly provided the data I needed to conduct the experiments.

# Contents

# Chapter 1

# Introduction

In graph theory, the shortest path problem is an algorithmic problem that consists in finding a path in a graph so that the total sum of all edge weights/costs constituting this path is minimized. Consequently, it is considered as a combinatorial optimization problem since the number of possibilities is finite in finite graphs. (we can find a more detailed description of this approach in appendix A). The objective is to find the best possible path regarding a specific cost function. This problem is well studied and there exist many algorithms and techniques to solve it. The basic goal of the problem is to find the shortest path between two points. If we consider a graph, the two points would be a well-defined origin node and a destination node. Here, we consider graphs that contain a finite number of nodes, and thus a finite number of edges. The shortest path between two nodes is called the **single-pair** [1] shortest path problem. Of course, there exist several variants to this basic idea. The **single-source** shortest path problem [2] consists in finding the shortest path from one single source vertex to all other vertices in a graph. The opposite is the **single-destination** [3] shortest path problem in which we desire to find the shortest path from all vertices to a single destination vertex in a graph. We can easily see that we can invert the arcs in a directed graph to obtain the reduced problem, which is the single-source shortest path problem. Another variant is the **all-pairs** shortest path problem [4] that computes the shortest path between all pairs of vertices in a graph. All of these latter variants can be dealt with, with specific algorithms that are more efficient than simply running a single-pair shortest path algorithm on all relevant pairs of vertices. For example, the **all-pairs** can be solved using the so-called Floyd-Warshall algorithm that will be described later.

Amongst all algorithms, the well-known Dijkstra algorithm is the most popular. It solves the shortest path problem in any weighted graph with non-negative weights in polynomial time. If the weights are not non-negative, it can be solved using the Bellman-Ford algorithm which requires the condition that the graph does not contain any negative cycle reachable from the source node. Indeed, finding the shortest path in a graph containing negative cycles is known to be NP-hard, which means that it cannot be solved using a polynomial-time algorithm. (We prove this in Appendix B.1).

Nevertheless, despite the fact that there exist several algorithms that solve the shortest-path problems, such as Dijkstra's algorithm or Bellman-Ford's algorithm, the recent increased amount of data forces us to find new techniques in order to considerably decrease their execution time. Several speed-up techniques have emerged in the latest years and among them,

there are : *heuristic search, bidirectional search, landmarks, reaches, short-cuts, and contractions*. Their efficiency depends on the topology of the graphs they are applied on. The term `topology` brings together several characteristics of a graph. Indeed, there are different kinds of graphs, such as road networks or social networks. The former contains way more edges than the latter for example. A graph can also be defined by other metrics such as the branching factor, the diameter, density and sparseness. Those metrics will be defined in chapter 2. For the experiments we will conduct later, we will only consider road networks as it is easy to get real-life data.

The problem with the existing speed-up techniques, as they are currently defined, is that they are not supported by graphs with more complex structures, such as multi-modal or user-adapted graphs. Indeed, multi-modal routing has become relevant in our society and is a field of research gaining more and more interest. Nowadays, many modes of transportation have been implemented and adopted by most people. Amongst them, we have public transports, shared bikes, shared cars and shared scooters. These modalities have been designed so that people can use them in combination with others and the freedom they offer can lead to complex traveling scenarios. As transportation networks become more complex and mobility in our society more important, the demand for efficient methods in route planning increases. Furthermore, implementing multi-modal systems imply bigger network structures, and applying a plain Dijkstra's algorithm on such structures is obviously more challenging than on single-modal networks. Hence the need to design and model adaptations of speed-up techniques to multi-modal routing systems. Besides, multi-modal routing brings another challenge. Indeed, multimodal graphs force shortest path algorithms to find compromises by offering tradeoffs between different metrics, which is not the case for intermodal graphs. Three main metrics can be measured in a travelling scenario : *time*, *cost* and *environmental impact*. While the relationship between these **3** measurements generally remains similar in a single-modal case, multi-modal shortest paths can offer a large range of tradeoffs in terms of time, cost, and ecological impact. Thus, there is an incentive to model user preferences in multimodal systems. Consequently, in the case of user-adapted graphs, different executions of the same algorithm will make sure each user has his own shortest path since each user has some kind of preference.

Given the existing speed-up techniques, several questions arise. Which optimization methods are the most efficient depending on the graph topology? How can we adapt these techniques to more complex graph structures? In this thesis, we will analyse several speed-up techniques and in particular, we will focus on *ALT* (*A*, landmarks, and triangle inequality*) speed-up technique algorithm and its applicability to multi-modal user-adapted road networks. It is a continuation of Victor's thesis ([5]) since he focuses on *Contraction Hierarchies* with time-dependent multi-modal road networks.

## Real world applications

Finding the shortest paths in a graph is a problem that applies in many real-world applications. For instance, if we represent a nondeterministic abstract machine as a graph where edges describe all transitions and vertices all states, we can use shortest path algorithms to find the sequence of actions needed to reach a given goal that is optimal in terms of time or any kind of weights. A simple game, represented by a finite state machine can be seen as a shortest path problem. Each state is a vertice of the graph and each move corresponds to a

directed edge. The algorithm can be used to minimize the underlying number of moves. The shortest path is also widely used in networking and telecommunications. And of course, it can be seen as a routing problem. As an example, a road network can be represented as a graph. The problem of finding the point-to-point shortest path in road networks has received great attention in recent years and even though Dijkstra's algorithm can solve this problem in almost linear time, there is a problem regarding the network size. Huge road networks require more efficient algorithms. In the road network itself, the nodes represent road junctions and the edges represent roads. The weight corresponding to an edge can be the length of a road or the time needed to walk from a road junction to another, or even the cost related to the difficulty of crossing a road. If we want to constraint some roads to be one-way streets, we can use directed edges.
Some edges could be more important than others regarding their weights. To illustrate it, we can consider Highways for instance. They are useful for long-distance travel and important to consider in priority when searching for the shortest path. Particularly, a property has been formalised with the notion of highway dimension [6], that we do not define here properly. It has been shown that low highway dimension gives guarantees of efficiency for several algorithms, such as Reach, contraction hierarchies, transit node, etc. The algorithms that exploit this property can thus find the shortest path a lot quicker. Note that the hierarchy of edges is constructed during the algorithm execution itself and without considering any additional information about an edge except its cost.

The fastest known algorithm to compute the shortest path on road networks is called hub labeling [7] and can find the shortest path of road networks of Europe or the USA in a fraction of a microsecond. This algorithm and other algorithms of this kind usually work in two phases. The first phase consists of a `preprocessing` on the graph without taking into account the source and target node. The second phase is a `query`, where the source and target node are known. Since the road network is static, the preprocessing can be done once and can be used for several queries on the same network. The preprocessing part is important since it reduces the computation time and allows algorithms to run in sub-linear time [8].

## Outline

We start by introducing some definitions and notations in chapter 2. In particular, we focus on definitions related to graphs as they will be extensively involved in all the concepts described in this thesis. Then, in chapter 3, we describe some well-known algorithms that tackle the shortest path problem in a graph, namely Dijkstra's algorithm, Bellman-Ford algorithm, Floyd-Warshall algorithm, and Johnson's algorithm. In section 3.2, we briefly introduce search algorithms and in particular, we focus on A* algorithm. We will finish this chapter by comparing the performance of all algorithms. In chapter 4.3, we present an overview of different prominent speed-up techniques. We start from small Dijkstra's algorithm improvements to well-studied complex algorithms as well as a combination of them. Chapter 5 introduces the notion of intermodality and user preferences. It highlights the motivations behind multimodal routing and how we can adapt single-modal graphs to multi-modal graphs. We also present how to take user preferences into account. Following, we conduct some experiments on single and multimodal networks in chapter 6. Last but not least, chapter 7 concludes our work and brings new challenges for future work.

# Chapter 2

# Preliminaries

We now define the shortest path problem mathematically in order to have some notations and precise formulations for the algorithms that we will describe later. We define the problem using graphs notations.

## 2.1 Definitions

In graph theory, the shortest path problem consists in finding the path between two nodes (vertices) such that the sum of weights of the edges constituting that path is minimal.

The shortest path problem can be defined for undirected, directed or mixed graphs. The only variant we are defining here is the undirected graph. In the case of a directed graph, the definition of path requires that consecutive vertices must be connected by an appropriate directed edge (meaning that each node of a path must have one in-going arc and one out-going arc). Moreover, mixed graphs contain both undirected and directed edges. We can thus transform an undirected graph into a mixed graph by adding some directed edges. In this case, it also requires the same condition as the directed graph.

### 2.1.1 Graph

**Graph** Let us consider a graph $G = (V, E)$ with the set of nodes/vertices $V$ and the set of edges $E$. We define an edge as a pair of vertices $(u, v)$, where $u \in V$ and $v \in V$ and $u \neq v$. Let $|V|$ be the cardinality of $V$ and $|E|$ be the cardinality of $E$.

**Undirected graph** A graph $G = (V, E)$ with bidirectional edges. That is, any edge $(u, v)$ is equivalent to $(v, u)$. $E$ is set of unordered pairs of edges from $V$. The maximal number of edges in $G$ is $\frac{|V|^2-|V|}{2}$.

**Directed graph** A graph $G = (V, E)$ where $E$ is a set of ordered pairs of edges (arcs) from $V$. That is, an edge $(u, v)$ is composed of an source vertex $(u)$ and a destination vertex $(v)$ and is not necessarily equivalent to $(v, u)$. The maximal number of edges in $G$ is $|V|^2 - |V|$

Given a directed graph $G = (V, E)$, the neighbour of a vertex $u$ is $v$ so that $(u, v) \in E$. In an undirected graph, $u$ is the neighbour of $v$ and $v$ is the neighbour of $u$.

**Weighted graph** A graph $G = (V, E, \omega)$ where $\omega : E \to \mathbb{R}$ is a weight function that associates a weight/cost $\omega(u, v)$ to each edge $(u, v)$. We use $\omega$ instead of $w$ in order to dissociate the weight from the node notation. Also, we can use $\omega(e)$ or $\omega(u, v)$ for $e = (u, v) \in E$.


Figure 2.1: An undirected graph and a directed graph

- *The diameter* : the number of edges in the longest shortest path,
- *The density* : a graph is *dense* whenever the number of edges is close to the maximum number of edges possible,
- *The sparseness* : a graph with only a few edges (the opposite of dense graphs). A graph is *sparse* when $|E|$ is way smaller than the maximal number of edges. (e.g. road networks)

### 2.1.2 Connected graph

**Reachable** A vertex $v$ is *reachable* from $u$ is there exist a path $P$ between them.

**Connected** An undirected graph is connected if each vertex of $V$ is reachable from any other vertex. In a directed graph, a connected graph is said to be **strongly connected**. A directed graph is **weakly connected** if its corresponding undirected graph is connected.

**tree** A (*directed*) tree is a (*weakly*) connected graph without cycles.

For the experiments that we conduct in this work, we only consider strongly connected weighted directed graphs with non-negative edge-weights and simple paths.

**Incidence** A vertex is incident to an edge if the vertex is one of the two vertices the edge connects. A pair $(v, e)$ is a incidence where $v$ is a vertex and $e$ is an edge that is incident to $u$.

**degree** The **degree** of a vertex $z \in V$ is the number of edges incident to $v$. In the case of directed graphs, the degree of a vertex $v$ is the sum of in-going and out-going degrees. That is, $|(u, v) \in E : u = z| + |(u, v) \in E : v = z|$.

**branching factor** The number of "children" at each node (i.e. the out-going degree) in a directed tree.

**Adjacency** Two vertices are *adjacent* when they are both incident to a common edge. Two distinct incidences $(u, e)$ and $(v, f)$ are adjacent if and only if $u = v$, $e = f$ or the edge $(u, v) = e$ or $(u, v) = f$.

### 2.1.3 Path

**Path** [9] [10] Let $G = (V, E, \epsilon)$ be a undirected graph. $\epsilon(e_i)$ is the set of vertices connecting the edge $e_i$. Let $e_1, e_2, \ldots, e_{k-1}$ be a sequence of edges of $E$ and $v_1, v_2, \ldots, v_k$ be a sequence of distinct vertices of $V$ such that $\epsilon(e_i) = \{v_i, v_{i+1}\}$ for $i = 1, 2, \ldots, k$, where $k$ is the number of vertices in the sequence. As defined in [10] , the sequence $e_1, e_2, \ldots, e_{k-1}$ of edges is a `path` $P$ of size $k - 1$ from $v_1$ to $v_k$ in G.

The sequence of vertices $v_1, v_2, \ldots, v_k$ constituting this path $P$ is the **vertex_sequence** such that $v_i$ is adjacent to $v_{i+1}$ for $1 \leq i < k - 1$. We can thus define a path by its sequence of edges $(e_1, e_2, \ldots, e_{k-1})$, or its sequence of vertices $(v_1, v_2, \ldots, v_k)$. We stick with the first definition :

$$P = (e_1, e_2, \ldots, e_{k-1})$$

**Simple path** A path with distinct vertices is a **simple-path**. [11]

In the undirected graph of Figure 2.1, the sequence of edges a,c,d is a path with the vertex sequence A,C,B,D. The same sequence of edges is a directed path in the directed graph of Figure 2.1.

If we use a directed graph, we must ensure that there is a directed edge from $v_i$ to $v_{i+1}$ and not from $v_{i+1}$ to $v_i$ for every pair of adjacent edges in the path $P$ for $1 \leq i < k - 1$. For mixed graphs, a path is constituted of both undirected edges and directed edges. Therefore, as mentioned above, each pair of successive adjacent vertices $v_i$ and $v_{i+1}$ have to satisfy the same property as in directed graphs.

**Trail** A sequence of edges $e_1, e_2, \ldots, e_{k-1}$ in a graph $G$ is considered as a *trail* if all edges are distinct, but all vertices $v_1, v_2, \ldots, v_k$ are not necessarily distinct. In the directed graph of Figure 2.1, the sequence of edges d,c,a,b,e is a directed trail, but not a directed simple path since it contains duplicate vertices.

**Walk** A *walk* is a finite or infinite sequence of edges $e_1, e_2, \ldots, e_{k-1}$ in $G$ which connect a finite or infinite sequence of vertices $v_1, v_2, \ldots, v_k$, which means that edges and vertices of this sequence do not necessarily have to be distinct. If a walk is infinite, there is no starting and ending vertex. A walk with a first vertex but no last vertex is a semi-infinite walk.

The definition of walk is general and both definitions above can be derived from it. Indeed, a trail is a walk with distinct edges and a path is simply a trail with distinct vertices. In the directed graph of Figure 2.1, the edge sequence a,b,a is a walk, but not a directed trail, nor a directed path since there are duplicate arcs and vertices.

For the shortest path problem, we will only consider *simple Paths*. That is, finite sequences of edges and distinct vertices. Besides, we also assumed in chapter 1 that graphs will be finite. Otherwise, some algorithms that we will describe later cannot terminate.

### 2.1.4 Shortest Path

**Path length** Given a real-valued weight function $\omega : E \to \mathbb{R}$ and an undirected (simple) graph $G$, the length of a path $P = (e_1, e_2, \ldots, e_{k-1})$ in G with respect to $\omega$ is the sum of the weights of its edges:

$$\mathcal{W}(P) = \sum_{e \in P} \omega(e)$$

**Shortest path** [1] Given a set $P_{st}$ of paths from $s$ to $t$, where $s$ represents the starting node and $t$ represents the destination node, the shortest path is the path $p^*$ with minimum total weight that lies in $P_{st}$ (as described in [12]) such that :

$$p^* = arg \min_{p \in P_{st}} \mathcal{W}(p)$$

When all edge in the graph have a unit weight (the same constant value) or $\omega : E \to \{1\}$. This is equivalent to finding the path with the fewest edges since each edge has the same importance. In this case, the path cost is defined by the number of edges constituting it.

**Distance** We denote the distance between 2 vertices $u$ and $v$ such as the length of the shortest path of a weighted graph. That is, given the shortest path $p*$

$$d(u, v) = \mathcal{W}(p*) = \mathcal{W}(arg \min_{p \in P_{st}} \mathcal{W}(p))$$

$d(u, v)$ must respect the **triangle inequality** property : $d(u, v) \leq d(u, w) + d(w, v)$

**Single pair** Shortest path between 1 source node $s$ and 1 destination node $t$ :

$$p^* = arg \min_{p \in P_{st}} \mathcal{W}(p)$$

**Single source** Shortest paths from a given node $s$ to all other nodes $v$ of the graph : with fixed $s$ and $\forall v \neq s \in V$,

$$p^* = arg \min_{p \in P_{sv}} \mathcal{W}(p)$$

**Single destination** Shortest paths from all nodes to a single destination node : with fixed $t$ and $\forall v \neq t \in V$,

$$p^* = arg \min_{p \in P_{vt}} \mathcal{W}(p)$$

**All pairs** Shortest paths between all pairs of nodes : $\forall s \forall t s \neq t \in V$,

$$p^* = arg \min_{p \in P_{sv}} \mathcal{W}(p)$$

# Chapter 3

# Shortest Path Algorithms

We now describe some well-known algorithms that tackle the shortest path problem in a graph. We first talk about *classical algorithms* and give some examples, followed by a small complexity analysis (time complexity and space complexity). Afterward, we will talk about *search algorithms*. These algorithms share the same goal as the classical algorithms, and are often very similar, but deal with different issues. We will see that some search algorithms are just some variant of the so-called Dijkstra's algorithm. We will analyse their optimality, time and space complexity, as well as a "topology" complexity. The goal here is to have a global overview of different algorithms that can be used in order to find the shortest path in a graph. Of course, their main purpose is not to be as efficient as possible, but to obtain an algorithmic result that solves the problem.

## 3.1 Classical algorithms

The shortest path problem can be solved using many algorithms. Each one of them deals with different versions of the shortest path problem. Dijkstra's algorithm solves the *single-source* shortest path problem with non-negative edge weights. Bellman–Ford algorithm solves the *single-source* problem with potentially negative edge weights. A* search algorithm solves for *single pair* shortest path using heuristics to try to speed up the search by reducing the search. Floyd–Warshall algorithm solves *all pairs* shortest paths. Johnson's algorithm solves *all pairs* shortest path, and may be faster than Floyd–Warshall on sparse graphs.

### 3.1.1 Dijkstra

The well-known Dijkstra's algorithm **(Edsger W. Dijkstra, 1956, [13])** [14] finds the shortest path from one node to all other nodes in a graph (single-source shortest path), producing a spanning tree. However, it is mostly used to solve the single-pair shortest path problem and can therefore be adapted accordingly. The condition for the algorithm to work properly is to work on non-negative edge weights graphs. Here, we describe Dijkstra's algorithm solving the single-pair shortest path problem.

The algorithm itself works with a *priority queue* and is quite simple. A priority queue is an abstract data type in which we attribute to each element a priority. An element with a higher priority than the others will be selected first. If two elements have the same priority, they are chosen according to the order in which they entered the queue.

- At the initialisation phase, each node in the graph is marked as "unvisited". A first set is created, containing all the unvisited nodes, and a second set is created, containing all the visited nodes (initially empty). Then, every node receives a *tentative* distance value, which is $\infty$, except for the starting node (0). The tentative distance value is the distance from the start node to a given node. That is, the shortest distance found so far.
- Set the current node to be the initial node.
- Consider the current node. Get all the unvisited neighbours of this node and compute their *tentative* distance value. Formally, let the current node be $c$, along with a tentative value $t_1$ and a neighbour $nb$, with a tentative values $t_2$. The new *tentative* distance value assigned to $nb$ is $t = t_1 + \omega(c, nb)$ if $t < t_2$, where $\omega(c, nb)$ is the weight of the edge $(c, nb)$. For example, if the current node A is marked with distance 6, its neighbour B with distance 9 and the edge connecting the two is 2, then B's tentative distance value has the new value $6 + 2 = 8 < 9$.
- Mark the current node as visited (add it to the visited set and remove it from the unvisited set so that it is never considered again by the algorithm)
- Set the next current node as the unvisited node with the smallest tentative distance and go back to the third step.
- Stop whenever the destination node is set as the current node and then marked as visited, or if the smallest tentative distance among the nodes in the unvisited set is $\infty$ (This can happen if the initial node and the remaining unvisited nodes have no connection between them).

When we look at the algorithm itself, we can wonder why does it actually work? Intuitively, we can see that at each step, we consider the shortest distance towards the neighbours of the current node. Therefore, we could imagine that after finding the destination node, we have a set of chosen edges that together form the shortest path of the graph. Nevertheless, we can prove the algorithm's correctness by induction on the number of visited nodes.

*Proof.* The invariant hypothesis would be that for each node $v$, $dist[v]$ is the shortest distance from the source node to $v$ by taking only visited nodes, or $\infty$ if there is no path. (it does not necessarily mean that $dist[v]$ is the actual shortest path for unvisited nodes)

- The base case is **one only** visited node, the initial node. That obviously satisfies the invariant hypothesis.
- Now, we can assume the hypothesis for **n-1** visited nodes. Consider an edge $(u, v)$ where $v$ is the node among all unvisited nodes that has the least $dist[v]$ and the edge $(u, v)$ is such that $dist[v] = dist[u] + \omega(u, v)$. We can derive a contradiction. $dist[v]$ is considered to be the shortest distance from the source node to $v$. Indeed, if there exists a shorter path, we would have chosen $w$, one of the unvisited nodes. Then, $dist[w] > dist[v]$ if we follow the original hypothesis, which leads to a contradiction. Similarly, if in the graph there were a path that is shorter to $v$ without exploring the unvisited nodes, and if the last but one node on that path were $w$, then it would mean that $dist[v] = dist[w] + \omega(w, v)$, which is also a contradiction according to the hypothesis. When the node $v$ has been processed, the property will hold for the following unvisited node $w$. $dist[w]$ will be the shortest distance from the source node to $w$ and all nodes

that belong to this path are only visited nodes. Indeed, if there were a shorter path that does not go by $v$, we would have found it previously, and if there were a shorter path including the node $v$, this path would have been updated according to $v$ like stated previously.

When the algorithm finally finds the destination node, it means that we have a set of visited nodes and each path from the source node to each visited node is the shortest possible distance, which means that the distance from the source node to the destination node is consequently the shortest path in the entire graph.

□

**Algorithm 1** Dijkstra

```
1:  function DIJKSTRA(G, s, t)
2:      unvisited ← priority queue
3:      visited ← ∅                                          ▷ Empty set
4:      dist[s] ← 0
5:      for each vertex v in G do                            ▷ And v ≠ s
6:          dist[v] ← ∞                                      ▷ tentative distance from start to v
7:          prev[v] ← ⊥                                      ▷ Predecessor of v is None
8:          unvisited.add_with_priority(v, dist[v])
9:          add v to unvisited
10:     end for
11:     c ← s                                                ▷ current node = start node initially
12:     while not (t ∈ visited or dist[c] == ∞) do
13:         for all w ∈ V and (c, w) ∈ E  do                 ▷ Only unvisited neighbours
14:             tentative_dist ← dist[c] + ω(c, w)           ▷ ω(c, w) = edge weight
15:             if tentative_dist < dist[w] then
16:                 dist[w] ← tentative_dist
17:                 prev[w] ← c
18:                 unvisited.decrease_priority(w, tentative_dist)
19:             end if
20:         end for
21:         unvisited.remove(c)
22:         visited.add(c)
23:         c ← unvisited.extract_minimum()                  ▷ vertex with min tentative dist
24:     end while
25:     return dist, prev
26: end function
```

If we want to solve the single-source shortest path problem, we have to change the termination condition.

- **Dijkstra single-source** : stop when the priority queue is empty.
- **Dijkstra single-pair** : stop when the destination node is visited or the priority queue is empty.

When the algorithm has found a solution, we must look at the "`prev`" array and start from

the goal node, going back successively to the starting node to get the full shortest path using algorithm 2.

**Algorithm 2** Reconstruct Shortest Path

```
1: function RECONSTRUCT_PATH(predecessor, current)
2:     full_path ← {current}
3:     while current in predecessor.keys do
4:         current ← predecessor[current]
5:         full_path.add(current)
6:     end while
7:     return full_path.reverse
8: end function
```

We can develop a small example by running the algorithm on this graph below. The start node is $C$ and the destination node is $E$. We expect the algorithm to find the shortest path which is the set of edges defined by these vertices :

$$C - A - B - E$$

with a total distance (path length) of $1 + 3 + 1 = 5$ in our case. The algorithm will terminate when the node $E$ is visited.

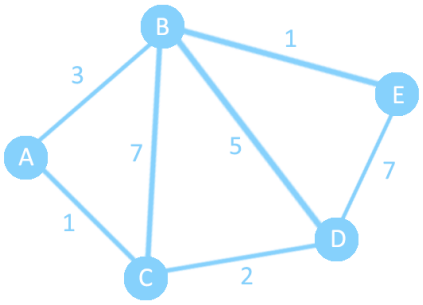
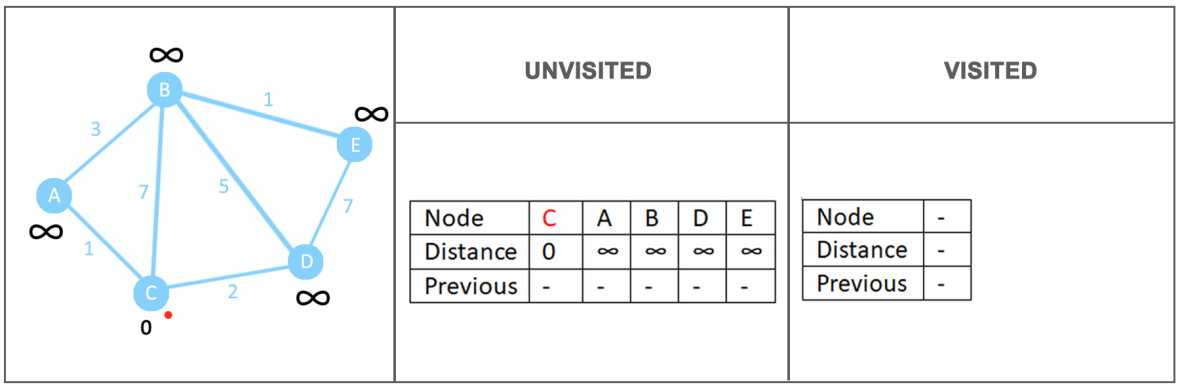
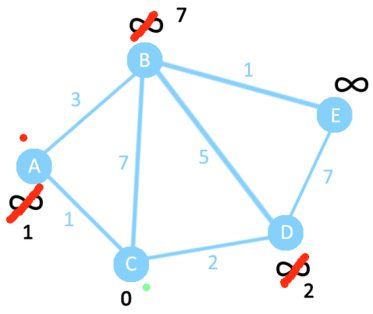



Figure 3.1: Exemple of Dijkstra's execution on a undirected graph



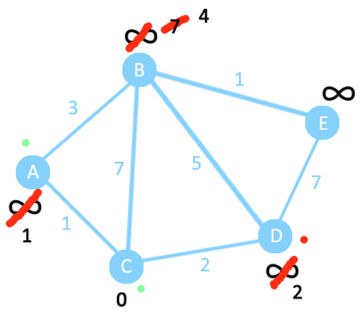



**UNVISITED**

| Node | A | D | B | E |
|---|---|---|---|---|
| Distance | 1 | 2 | 7 | ∞ |
| Previous | C | C | C | - |

**VISITED**

| Node | C |
|---|---|
| Distance | 0 |
| Previous | - |

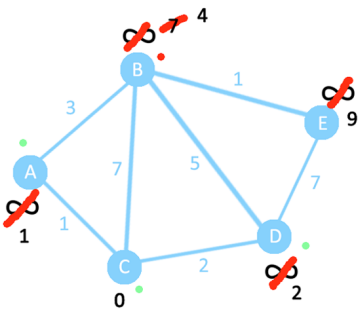



**UNVISITED**

| Node | D | B | E |
|---|---|---|---|
| Distance | 2 | 4 | ∞ |
| Previous | C | A | - |

**VISITED**

| Node | C | A |
|---|---|---|
| Distance | 0 | 1 |
| Previous | - | C |

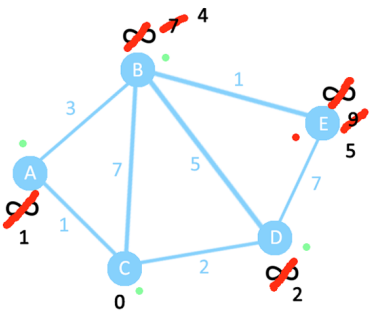



**UNVISITED**

| Node | B | E |
|---|---|---|
| Distance | 4 | 9 |
| Previous | A | D |

**VISITED**

| Node | C | A | D |
|---|---|---|---|
| Distance | 0 | 1 | 2 |
| Previous | - | C | C |

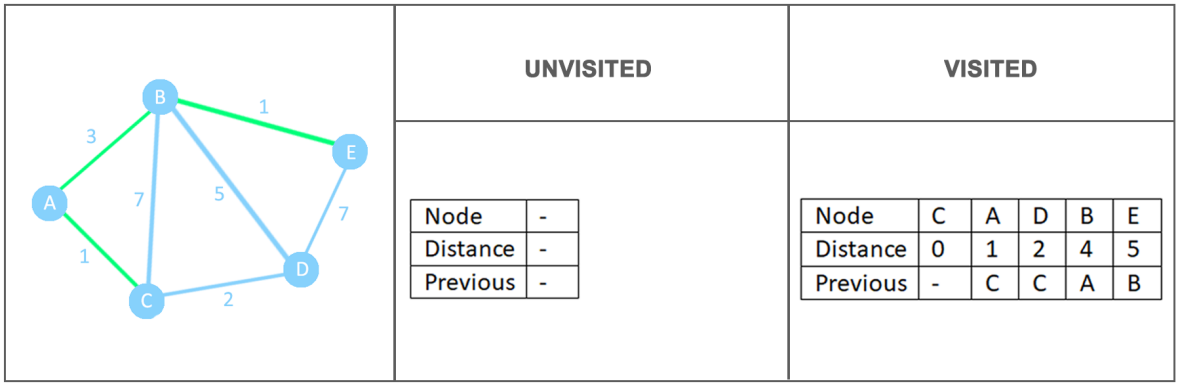



**UNVISITED**

| Node | E |
|---|---|
| Distance | 5 |
| Previous | B |

**VISITED**

| Node | C | A | D | B |
|---|---|---|---|---|
| Distance | 0 | 1 | 2 | 4 |
| Previous | - | C | C | A |



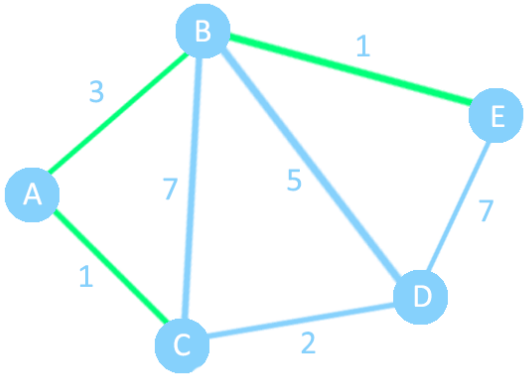




Figure 3.2: Final path found

Since the node $E$ has been visited by the algorithm, we can reconstitute the shortest path by looking at the succession of previous nodes in the visited set. $E$ is preceded by $B$. $B$ is preceded by $A$, and $A$ is preceded by $C$. It gives us the shortest path which is the set of edges defined by the vertices :

$$C - A - B - E$$

with a total distance (path length) of 5, as expected. We could also have computed all shortest paths from node $C$.

### Performance

In terms of performances, Dijkstra's algorithm's time complexity depends on the number of edges $|E|$ and vertices $|V|$ and the data structure used to implement the "unvisited" set $Q$. If the set $Q$ is represented as a simple array or a simple linked list and the extraction time of the minimum value (extract-minimum operation) in it is linear, then the algorithm runs in

$$\mathcal{O}(|E| + |V|^2)$$

This can be explained by the fact that at each iteration of the algorithm, we consider a set of nodes (neighbours of the current node) and we do that until the current node is the destination node. In the worst-case scenario, all the nodes are explored twice. Generically, for any data

structure of the set $Q$, the time complexity is

$$\mathcal{O}(|E|.T_d + |V|.T_{em})$$

where $T_d$ is the time needed to compute the `decrease` function (decrease the priority of keys in the set $Q$), and $T_{em}$ is the time needed to compute the `extract_minimum` operation (extract the minimum value from the set $Q$). To improve the performance of the algorithm, we can use Q as a binary or Fibonacci heap. A `binary heap` [14] [15] is of the form of a complete binary tree (each layer of the tree is fully filled with nodes) and can be used as a priority queue. Each node has either a value greater or equal, or less or equal to its children's values. The time (worst case) needed to insert an element and to remove the smallest or largest element in the binary heap is logarithmic. Thus, the time complexity of Dijkstra's algorithm using a binary heap is

$$\mathcal{O}((|E| + |V|)log|V|)$$

as described in [16]. To improve the time complexity even more, we can use a `fibonacci min-priority heap` [14] instead of a binary heap. The latter is a collection of trees so that the key of a child node in a tree is always greater than or equal to the key of the father node. What is interesting is that the operation that finds the minimum value in the heap is constant $\mathcal{O}(1)$. Thus, the time complexity is improved to

$$\mathcal{O}(|E| + |V|log|V|)$$

as stated in [16].

Now that we have the algorithm, we can be interested in modifying it to find several shortest paths in a directed graph since it is possible that several different paths from the source to destination node share the same length. To do so, we have to store all nodes that satisfy the relaxation condition in the "`prev`" data structure (set of previous nodes), instead of just storing one node. This way, all nodes that connect to the destination node with the same shortest path cost will be added to `prev`. At the end of the algorithm, the `prev` data structure will be a graph which is a subset graph of the original graph but with probably fewer edges. Any path from the starting node in the original graph to any node in the new graph will be the shortest path in the original graph. Therefore, we can backtrack from the target node and provide the shortest paths of that original graph.

**Table 3.1** Dijkstra time complexities

| | Time Complexity |
|---|---|
| Data structure | Dijkstra |
| Simple array | $\mathcal{O}(\|E\| + \|V\|^2)$ |
| Binary heap | $\mathcal{O}((\|E\| + \|V\|)log\|V\|)$ |
| Fibonacci heap | $\mathcal{O}(\|E\| + \|V\|log\|V\|)$ |
| Generic | $\mathcal{O}(\|E\|.T_d + \|V\|.T_{em})$ |

### 3.1.2 Bellman-Ford

Bellman-Ford algorithm **(Richard Bellman and Lester Randolph Ford junior, 1956 and 1958, [17])** [14] [18] finds the shortest path from a source node to all other nodes in a directed graph (single-source shortest path problem). Unlike Dijkstra's algorithm, this algorithm can deal with graphs containing negative weights. If there are *negative cycles* in the graph that are reachable from the source node, then Bellman-Ford cannot find any shortest path since it would be stuck in negative cycles. Bellman-Ford algorithm is able to detect negative cycles during its execution.

The algorithm itself is quite similar to Dijkstra's algorithm in the sense that it relaxes edge weights. The difference is that Dijkstra uses a priority queue in order to extract the minimum distance value at each step whereas Bellman-Ford proceeds to relax all nodes for $|V|$ iterations. We denote "relaxation" by the update of the distance of a vertex by a smaller tentative distance value. When the algorithm has finished relaxing edges, it looks at all edges again in order to detect negative weight cycles. If some edges can still be relaxed, it means that there is a negative cycle.

**Algorithm 3** Bellman-Ford

```
1:  function BellmanFord(G, s)
2:      dist ← map for every vertex in G
3:      prev ← map for every vertex in G
4:      dist[s] ← 0
5:      for all v ∈ G do                                  ▷ And v ≠ s
6:          dist[v] ← ∞                                   ▷ distance from start to v
7:          prev[v] ← ⊥                                   ▷ Predecessor of v is None
8:      end for
9:      for i from 1 to |V| - 1 do                        ▷ |V| - 1 repetitions
10:         for all (u, v) ∈ E do
11:             if dist[u] + ω(u, v) < dist[v] then
12:                 dist[v] ← dist[v] + ω(u, v)           ▷ Relax edge
13:                 prev[v] ← u
14:             end if
15:         end for
16:     end for
17:     for all (u, v) ∈ G.E  do                          ▷ Checks for negative cycles
18:         if dist[u] + ω(u, v) < dist[v] then
19:             return error                              ▷ Contains negative weight cycle
20:         end if
21:     end for
22:     return dist, prev
23: end function
```

#### Performance

The algorithm makes $|E|$ relaxations for every iteration and there are $|V|-1$ iterations. Thus, the worse-case execution time is

$$\mathcal{O}(|V|.|E|)$$

Consequently, the performance depends on the number of vertices and edges in the graph.

### 3.1.3 Floyd-Warshall

The Floyd-Warshall algorithm **(Robert Floyd in 1962, Bernard Roy in 1959 and Stephen Warshall in 1962, [19] [20])** [14] [21] [22] computes the shortest path from all pairs of nodes in a weighted graph with positive or negative edge weights, but not containing any negative cycles (all-pairs shortest path problem). The principle behind this algorithm is based on a recursive property.

Let $G$ be a graph with $|V|$ vertices. We define `ShortestPath`(i,j,k), a function that returns the shortest path from node $i$ to node $j$ so that the intermediate points on that path belong to the set $\{1, 2, \ldots, k\} \in V$. The goal of the algorithm is to find the shortest path from each vertex $i$ to each vertex $j$ using any vertices in $V = \{1, 2, \ldots, |V|\}$. To do that, it computes an estimate of the shortest path between every pairs of vertices and tries to improve this estimate until finding the optimal one. The recursive algorithm works as follow (we will use "`SP`" as an abbreviation of "`shortestPath`") :

**Algorithm 4** Floyd Warshall

```
1:  function FloydWarshall(G)
2:      dist ← ∅                                   ▷ empty arary of size |V| x |V|
3:      for all (u, v) ∈ E do
4:          dist[u][v] ← ω(u, v)
5:      end for
6:      for all v ∈ V do
7:          dist[v][v] ← 0
8:      end for
9:      for k from 1 to |V| do
10:         for i from 1 to |V| do
11:             for j from 1 to |V| do
12:                 if dist[i][j] > dist[i][k] + dist[k][j] then
13:                     dist[i][j] ← dist[i][k] + dist[k][j]
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return dist
19: end function
```

- The base case is
$$\texttt{SP}(i, j, 0) = \omega(i, j)$$
where $\omega(i, j)$ is the weight of the edge $(i, j)$. $k = 0$, it means that the shortest path from vertex $i$ to vertex $j$ is simply the weight of the incident edge between them.

- The recursive case is defined by
$$\texttt{SP}(i, j, k) = min(\texttt{SP}(i, j, k-1), \texttt{SP}(i, k, k-1) + \texttt{SP}(k, j, k-1))$$
. Indeed, for each pairs of vertices, $\texttt{SP}(i, j, k)$ is either

– a path that contains only vertices from the set $\{1, \ldots, k-1\}$, not containing $k$.

$$\texttt{SP}(i, j, k-1)$$

– a path that contains $k$, from $i$ to $k$ and then from $k$ to $j$, both using vertices from the set $\{1, \ldots, k-1\}$. If the shortest path happens to be from $i$ to $k$ and then from $k$ to $j$, then it means that the length of this path from $i$ to $j$, with $k$ along, is the concatenation of the shortest path from $i$ to $k$ (only using intermediate nodes from $\{1, \ldots, k-1\}$), and the shortest path from $k$ to $j$ (only using intermediate nodes in $\{1, \ldots, k-1\}$) :

$$\texttt{SP}(i, k, k-1) + \texttt{SP}(k, j, k-1))$$

.

Thus, we take the minimum value of these two SP.

The algorithm hereafter uses this recursive property and tries values of $k$ from 1 to $|V|$, the number of vertices.

We can develop a small example with this graph below. It is a directed graph with only positive edge weights.

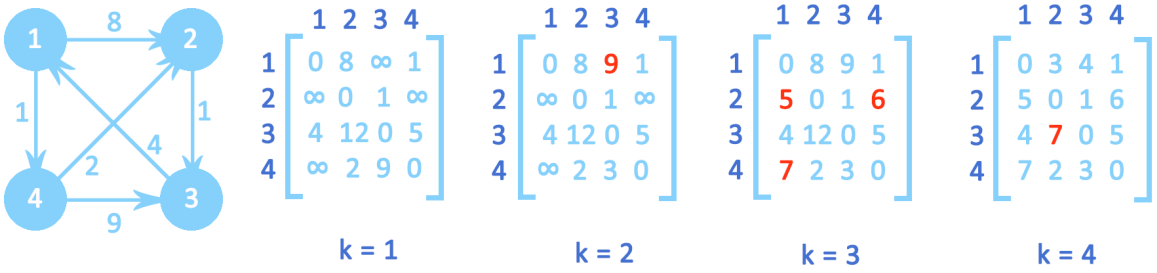



Figure 3.3: Floyd Warshall example of execution

At iteration 4, Floyd-Warshall has found the shortest path between all pairs of vertices.

**Performance**

This algorithm runs in $\mathcal{O}(|V|^3)$ and is used to compute shortest path between all pairs of vertices in dense graphs. Indeed, to compute all $\texttt{SP}(i, j, k)$ and all $\texttt{SP}(i, j, k-1)$, it requires $2|V|^2$ operations. Also, the algorithms computes this function for each $k$ from 1 to $|V|$. Therefore, the number of operations in total is

$$|V|.2|V|^2 = 2|V|^3 = \mathcal{O}(|V|^3)$$

In sparse graphs with non-negative edge weights, Dijkstra is prefered as its running time is better ( $\mathcal{O}(|E||V| + |V|^2 \log |V|)$ using fibonacci heaps).

If we have sparse graphs with negative edges but no negative cycles, we can use Johnson's algorithm, which we describe hereafter.

### 3.1.4 Johnson's algorithm

The Johnson's algorithm **(Donald B. Johnson, 1977, [23])** [14] [24], like *Floyd-Warshall* (3.1.3), finds the shortest path between all pairs of vertices in a weighted directed graph with possibly negative weights, but no negative-weight cycles. We could try to run Dijkstra (3.1.1) for every vertex to obtain a set of single-source shortest paths. The problem is that it does not deal with negative weights. Johnson's algorithm actually uses two existing algorithms that we saw, to compute the shortest paths : **Bellman-ford** (3.1.2) and **Dijkstra's algorithm** (3.1.1). The idea is to add a new node to the set of nodes in the original graph and use Bellman-ford algorithm on the initial graph to find the shortest path from this new node to all other nodes, and assign a weight to every vertex in the graph. Then, the weights of the graph are modified according to these vertex weights in order to get rid of negative weights. Finally, Dijkstra's algorithm is applied to the transformed graph.

**Algorithm 5** Johnson's algorithm

```
1: function JOHNSON(G)
2:     G'.V ← G.V + q
3:     G'.E ← G.E + ((q,v) for v in G.V)
4:     dist ← Bellman-Ford(G', q)
5:     for all (u, v) ∈ G'.E do
6:         ω(u, v) ← ω(u, v) + h(u) − h(v)
7:     end for
8:     Shortest_paths ← ∅                          ▷ To store results
9:     for all v ∈ G.V do
10:        Shortest_paths.append(Dijkstra(G, v))
11:    end for
12:    return Shortest_paths
13: end function
```

Formally, the algorithm can be defined as follow :

- Let $G$ be the considered graph. Add a new artificial node $q$ to $G$, connected to every other nodes in the graph. Let $G'$ be the new graph.
- Apply *Bellman-Ford* algorithm on $G'$ to get all shortest paths from the source node $q$ to all other nodes. It will find for each vertex $v$ the minimum weight $h(v)$ of a path from $q$ to $v$. Note that the algorithm can detect negative cycles if there exists some. In this case, the entire algorithm terminates.
- Update every weights of the graph. Given an edge from $u$ and $v$, with length $\omega(u, v)$, its new value is
$$\omega(u, v) = \omega(u, v) + h(u) - h(v)$$
- Remove q and apply Dijkstra's algorithm to find the shortest paths from each starting node $s$ to every other vertex in the new graph.

Below, we show a small example using Johnson's algorithm. The graph contains positive and negative weights, but no negative-weight cycles.

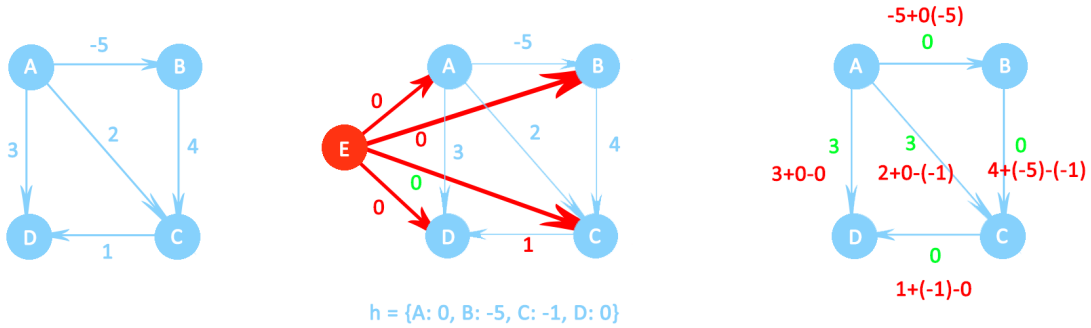



Figure 3.4: Exemple of Johnson's algorithm execution

### Performance

The algorithm takes $\mathcal{O}(|V||E|)$ time to run Bellman-Ford algorithm and $\mathcal{O}(|V|\log|V|+|E|)$ for each of the $|V|$ instanciations of Dijkstra's algorithm (using a Fibonacci heap, as mentionned in 3.1.1). Thus, the time complexity of the Johnson's algorithm is

$$\mathcal{O}(|V|^2\log|V|+|V||E|)$$

Compared to Floyd-Warshall's algorithm, this algorithm is faster for sparse graphs as its time complexity depends on the number of edges $|E|$, while Floyd-Warshall does not. If the number of edges in the graph is small (sparse graph), then

$$\mathcal{O}(|V|^2\log|V|+|V||E|)$$

is faster than $\mathcal{O}(|V|^3)$ (Floyd-Warshall).

## 3.2 Search algorithms

Another approach to the shortest path problem is as a search problem using agents. Indeed, an *agent* can establish goals that result in solving a problem by considering different sequences of actions that might achieve those goals. A goal is a desired state of the world. This goal can be thought of as a set of world states in which it is satisfied or there could be multiple goals in the same world. The agent has the possibility to think about which action he can take in order to get him to the goal state. In our case, the *world* can be represented as a *graph* and the *goal state* is simply a *path* in this graph which is the shortest regarding the weights that constitute it. As said before, the goal could also be the shortest path from one node to all other nodes (single-source shortest path) or the shortest path from all nodes to one specific node (single-destination shortest path) or even the shortest path between all pairs of nodes in the graph (all-pairs shortest path).

An *action* could be to *pick* up a node when searching for the shortest path in a graph. An agent must assess several possible sequences of actions leading to the goal state(s) and choose the best one. To do so, we can use search algorithms that take a problem as input, formulated as a set of goals to be achieved and a set of possible actions to be applied and return a sequence of action which is a solution. We need an *initial state*, which is the starting node(s), operators, which describe actions in terms of states reached by applying them, and a *goal state*, which is the destination node(s). The initial state and operators together define the *state space* of the problem which is the set of all states reachable from the initial state and any sequence of actions by applying the operators. We can describe a generic search algorithm framework [25] as follows:

**Algorithm 6** Agenda Search

```
1: function AgendaSearch(start_state, actions, goal_test)        ▷ Returns an action sequence or failure
2:     seq ← ∅                                                   ▷ an action sequence, initially empty
3:     agenda ← ∅                                                ▷ a state sequence, initially contains start_state
4:     while True do
5:         if Empty(agenda) then
6:             return failure
7:         end if
8:         if goal_test(First(agenda)) then
9:             return seq
10:        end if
11:        agenda ← Queuing_Fn(agenda, Expand(First(agenda), actions))
12:        seq ← Append(seq, Action(First(agenda)))
13:    end while
14: end function
```

- `Queuing_Fn` is a function that determines what kind of search we are doing,
- `Expand` is a function that generates a set of states achieved by applying all possible actions to the given state
- `First` is a function that gives the first element of a set of elements (action of state).
- The `goal_test` function is a boolean function over the states that gives True if the

input is the goal state and False otherwise.

There exist several basic search algorithms that implement different `Queing_Fn` functions. Those algorithms belong to 2 different classes : **uninformed strategies** (the way they expand their nodes is by only using the information in the problem formulation), and **informed** or **heuristic search strategies** that uses a quality measure not strictly in the problem formulation to generate solutions more effectively. In particular, we will only focus on *A** algorithm as it will be crucial in order to introduce a speedup technique later on. The rest of the algorithms are presented in Appendix C.

### 3.2.1 Uninformed search

The order of node expansion defines a strategy. Uninformed strategies only use information in the problem formulation (initial state, operator, goal state, path cost). it can be seen as a blind search or a *brute force* search. Examples of algorithms are *Depth-first search* (DFS, see C.1.1), *Breadth-first search* (BFS, see C.1.2), *Iterative-deepening search* and *Uniform-cost search* (UCS, see C.1.4). Those are described in details in Appendix C

### 3.2.2 Informed search

Informed search strategies rely on *heuristics*, a quality measure that is not in the problem formulation and that allows guiding the order in which nodes are expanded. The informed search methods give particular information about the state space in order to generate the solutions in a more efficient way. The heuristic describes the desirability of expanding nodes. It is an approximation since we cannot know in advance which node to expand in order to find the best solution. Hence, unlike uninformed search algorithms which only consider the information contained in the initial problem formulation, the heuristics involved in the informed search algorithm will help the algorithm to choose between several nodes. The most desirable nodes will be processed first, leading to the optimal solution.

#### A

The *A* algorithm is a combination of *Uniform Cost search* and *Hill-Climbing search*, described in appendix C (C.1.4 and C.2.1 respectively). But now, it takes the cost so far (denoted $g(n)$) and the estimated cost remaining heuristic (denoted $h(n)$) into account. Together, it forms the overall utility function $f(n) = g(n) + h(n)$.

#### A*

Like the A algorithm above, A* **(Peter Hart, Nils Nilsson and Bertram Raphael, 1968 [26])** [27] is an informed search algorithm and aims to find the optimal path from a specific starting node in a graph to a given goal state (single-pair). At each iteration of its main loop, it determines which of its node to extend by evaluating the cost of the path and a specific *heuristic* required to extend the path all the way to the goal. A*, as A, selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where $n$ is the next node on the path, $g(n)$ is the cost of the path from the start node to $n$, and $h(n)$ is a heuristic function that estimates the cost of the shortest path from $n$ to the

goal state. A* ends when the path it chooses to extend is a path that contains the start and the goal state or if there are no eligible paths to be extended anymore.

The difference with A algorithm is that now, A* uses a heuristic which is **admissible**, meaning that it never overestimates the costs to get to the goal. For example, given a node $B$, if the estimated cost is $X$ while the real cost to reach the goal state from $B$ is $Y < X$, then the heuristic is inadmissible because the heuristic $X$ over-estimates the real cost $Y$.

**Lemma 1.** *A* guarantees to return the optimal solution if the heuristic function is admissible.*

*Proof.* We assume that there is an optimal solution $G*$ with an optimal cost $f(G*) = C*$. Suppose A* returns a non-optimal solution. It means that a non-optimal solution $G2$ must be in the agenda at some point of the algorithm. If $G2$ is non-optimal, it means that its cost function $f(G2) > f(G*) = C*$. Further, suppose that there is a node $v$ in the agenda which is on the path towards the optimal solution. If the heuristic $h(v)$ is admissible, then $h(v)$ does not overestimate the actual cost to the goal state, by definition. Therefore, $f(v) < C*$. Now, if we combine the two formulas above, we obtain the following inequality :

$$f(v) < C* < f(G2)$$

Consequently, we can see that every node on the path towards the optimal solution has a cost that is less than the cost of the non-optimal solution $G2$. Hence, we conclude that $G2$ will never be chosen (expanded) from the priority queue during the algorithm execution. $G2$ will never be a solution returned by A*. Therefore, A* will always return an optimal solution if its heuristic $h$ is admissible. □

### Algorithm

To implement that algorithm (see pseudocode 7), we use a *priority queue*, which is called `open set`, in order to store the repeated selection of minimum cost nodes to expand. This `open set` can be implemented with a min-heap (*binary heap*), a *priority* queue or a *hash* set. At each iteration, we remove the node with the lowest $f(n)$ value from the queue and the $f$ and $g$ values of its neighbors are updated. these neighbors are added to the queue. The algorithm terminates whenever a goal node has a lower $f$ value than any node in the queue, or the queue is empty.

The solution found is the shortest path and its cost is the $f$ value of the goal state we end up with since the heuristic $h(t) = 0$ by definition of admissible heuristic. In order to get the sequence of actions leading to the shortest path, the algorithm should also keep track of all predecessors of each node. At the end of the execution, we could go back from the last node to the first node by following the predecessors using Algorithm 2.

An alternative approach would be to use a *monotone*, *consistent* heuristic. If the heuristic $h$ satisfies the additional condition

$$h(x) \leq d(x, y) + h(y)$$

for every arcs $(x, y)$ of the graph, where $d$ is the distance between nodes $x$ and $y$, then $h$ is "*consistent*". Note that this heuristic is equivalent to the definition of admissible heuristic. Indeed, given nodes $x$ and $y$, if the heuristic is consistent, it means that $x$'s heuristic must be lower than or equal to the distance from $x$ to $y$ added to $y$'s heuristic. Since $(x, y)$ is a

directed edge from $x$ to $y$, $x$ has to reach node $y$ before reaching the goal state. Thus, the heuristic $h(x)$ do not overestimate the real cost from $x$ to the goal state assuming $y$'s heuristic is admissible.

The advantage of this heuristic is that A* guarantees to find an optimal solution without preprocessing any node more than once. This version of A* is equivalent to running Dijkstra's algorithm with the reduced cost

$$d'(x, y) = d(x, y) + h(y) - h(x)$$

In another point of view, Dijkstra's algorithm (3.1.1), as another example of uniform-cost search algorithm can be viewed as a special case of A* algorithm with a heuristic function $h(x) = 0$.

**Algorithm 7** A*

```
1: function ASTAR(start, goal, h)                          ▷ h is the heuristic function
2:     openSet ← {start}
3:     pred                                                ▷ pred[n] = node preceding the node n
4:     g ← ∅                                               ▷ g[n] = cost so far, default values set to Infinity
5:     g[start] ← 0                                        ▷ f[n] = g[n] + h(n) = utility function
6:     f ← ∅                                               ▷ default values set to Infinity
7:     f[start] ← h(start)
8:     while openSet ≠ ∅ do
9:         c ← node in openSet with lowest f value         ▷ c = current
10:        if c == goal then
11:            return reconstruct_path(pred, current)
12:        end if
13:        openSet.remove(c)
14:        for each neighbor n of current do
15:            n_g ← g[c] + ω(c, n)                        ▷ distance from start to neighbor through current
16:            if n_g < g[n] then
17:                pred[n] ← c
18:                g[n] ← n_g
19:                f[n] ← g[n] + h(n)
20:                if n ∉ openSet then
21:                    openSet.add(n)
22:                end if
23:            end if
24:        end for
25:    end while
26:    return failure                                      ▷ open set empty, goal not reached
27: end function
```

### 3.2.3 Analysis

Now that we have an overview of the different algorithms, we can compare them in terms of performance. To do so, we can analyse their **time** and **space** complexity (we can find this analysis in [25]). However, there are several ways of defining the complexity of an algorithm. We focus here on the general complexity (time and space complexity in terms of the number of nodes and edges) as well as the complexity induced by the specific graph topology. Each algorithm has been designed to tackle one or several aspect(s) of the shortest path problem (single-source, all-pairs, ...) and each one of them is dealing with the search space differently.

In order to compare those algorithms described in section 3.2, we can assess these following criteria:

- *Completness* : If a solution exists for a given problem, the algorithm is always able to find a solution
- *Time complexity* : Can be measured :
  - in terms of execution time : how much time did the algorithm actually take
  - in terms of search space : number of explored nodes
- *Space complexity* : How much memory did the algorithm use (maximum number of nodes in memory at once)
- *Optimality* : The algorithm is always able to find the least-cost solution.

We can also assess the complexity of the algorithms with the following information about the search tree topology:

- the maximum *branching factor* of the search tree, denoted $b$ (if the tree is dense or sparse depending on the average number of successors per node)
- the *depth* of the optimal solution, denoted $d$
- the *maximum depth* of the state space, denoted $m$

In some cases, the complexity of an algorithm can not be easily characterized in terms of $b$ and $d$. Instead, the performance will be computed using *path costs* and *action costs* (costs of the operations in the algorithm). We can find the analysis of different algorithms in appendix C. We focus here on the analysis of A* :

#### Completness

A* is complete and guarantees to terminate on finite graphs with non-negative edge weights, it will always find a solution if one exists. Even though we are not working with infinite graphs, it is worth mentioning that if the graph is infinite with a finite branching factor $b$ and edges costs satisfying

$$d(x, y) > \epsilon\ 0$$

for a fixed $\epsilon$, then A* guarantees to terminate if there is a solution.

#### Time complexity

The time complexity of A* depends on the chosen heuristic. If the heuristic function is well chosen, it can have a great impact on the performances of the algorithm since it prevents A*

from expanding many of the $b^d$ nodes that an uninformed search would have expanded. To measure the efficiency induced by a given heuristic (in addition to the search space), we can measure the *effective* branching factor $b^*$, which can be calculated empirically by measuring the number of expanded nodes, $N$, and the depth of the solution $d$.

$$N = 1 + b* + (b^*)^2 + \ldots + (b^*)^d$$

For example, if A* finds a goal at depth 5 using 52 nodes, the effective branching factor is 1.91. A good heuristic is characterised by a low *effective* branching factor. The optimal branching factor is 1 ($b^* = 1$). In the worst-case, A* runs in $\mathcal{O}(b^d)$ time, which is exponential in the depth of the solution when the search space is unbounded. If the goal state is not reachable from the starting node and the state space is infinite, then A* will never terminate. But it can terminate with a "failure" output if the goal state is unreachable and the graph is finite. If we do not look at the worst case scenario of A* algorithm, its time complexity would be polynomial if the search space is a tree and the goal state is reachable. One additional condition for the complexity to be polynomial is that the heuristic function satisfies :

$$|h(x) - h(x^*)| = \mathcal{O}(\log h^*(x))$$

where $h^*$ is the optimal heuristic, meaning the exact cost to reach the goal state from a node x. That formula means that the error of the heuristic $h$ will not grow faster than the logarithm of the optimal heuristic $h^*$. It gives us a bound on the time complexity of the algorithm.

**Space Complexity**

Concerning the space complexity, it is similar to the other search algorithms since it stores all generated nodes in memory. Thus, the worst-case space complexity is $\mathcal{O}(b^d)$

**Optimality**

A* is optimal as long as the heuristic is admissible, which is supposed in the algorithm description (it never overestimates the costs to get to the goal state).

## 3.3 Performance summary

Now that we have lots of algorithms at our disposal, we can draw up a summary of their performances and analyse the most interesting ones. We use the same performance measures as in 3.2.3. Namely,

- *General Time and Space complexity* : number of nodes $|V|$ and edges $|E|$
- *Topological Time and Space complexity*, using branching factor $b$, search depth $d$ and path costs $C$
- *Completeness* : The algorithms guarantees to find a solution when there is one
- *Optimality* : The algorithm is always able to find the optimal solution

### 3.3.1 Time and Space Complexity

**Table 3.2** Time and Space complexities

| Complexity | | | | |
|---|---|---|---|---|
| | Time | | Space | |
| Algorithms | General | Topological | General | Topological |
| Depth-First Search | $\mathcal{O}(\|V\| + \|E\|)$ | $\mathcal{O}(b^m)$ | $\mathcal{O}(\|V\|)$ | $\mathcal{O}(bm)$ |
| Breadth-First Search | $\mathcal{O}(\|V\| + \|E\|)$ | $\mathcal{O}(b^d)$ | $\mathcal{O}(\|V\|)$ | $\mathcal{O}(b^d)$ |
| Uniform-Cost Search | / | $\mathcal{O}(b^{1+(C^*/\epsilon)})$ | / | $\mathcal{O}(b^{1+(C^*/\epsilon)})$ |
| Iterative-Deepening Search | $\mathcal{O}(\|V\| + \|E\|)$ | $\mathcal{O}(b^d)$ | / | $\mathcal{O}(d)$ |
| A* | $\mathcal{O}(\|V\| + \|E\|)$ | $\mathcal{O}(b^d)$ | $\mathcal{O}(\|V\|)$ | $\mathcal{O}(b^d)$ |
| Dijkstra | $\mathcal{O}(\|E\| + \|V\|log\|V\|)$ | | $\mathcal{O}(\|V\|)$ | |
| Bellman-Ford | $\mathcal{O}(\|V\|\|E\|)$ | | $\mathcal{O}(\|V\|)$ | |
| Floyd-Warshall | $\mathcal{O}(\|V\|^3)$ | | $\mathcal{O}(\|E\|^2)$ | |
| Johnson | $\mathcal{O}(\|V\|^2 log\|V\| + \|V\|\|E\|)$ | | $\mathcal{O}(\|V\|)$ | |

Here, we only consider the **worst case** time and space complexities. Note that we have two separate ways of defining the search spaces for time and space complexities. On one hand, we define the search spaces *implicitly*, that is, as a set of states and transitions between them. On the other hand, we define the search spaces *explicitly*, that is, as concrete sets of vertices and edges. In the former, states can be vertices and transitions can be edges. However, the set of states may not have finite bounds and the number of edges and vertices cannot be defined by finite cardinalities. Therefore, in this case, it makes sense to define performance in terms of branching factor $b$ in a specific kind of graph, which is a tree, as opposed to vertex and edge cardinalities. On the latter, we have a concrete definition of the search space in a worst-case scenario. It is thus worth examining the two but not comparing them, since they are different. If we want to translate the first notation to the other, we have to change the meaning of V and E. Indeed, consider a measure of the running time of an algorithm in terms of branching

factor $b$ of the search space (tree) and the depth of the destination node (goal node $d$). Once the goal node is found, we stop. In the tree, if we examine every vertex at depth $\leq d$ before finding the goal node, we will end up visiting $\mathcal{O}(b^d)$ vertices before stopping. We can see this as visiting a subset of the graph with $|V| = \mathcal{O}(b^d)$, where $V$ includes only the visited vertices and $|E| = \mathcal{O}(b^d)$, where $E$ includes only the edges we look at. This way, we can assume that the $\mathcal{O}(b^d)$-time algorithm's running time in terms of $V$ and $E$ is $\mathcal{O}(|V| + |E|)$. But $\mathcal{O}(b^d)$ is more informative than $\mathcal{O}(|V| + |E|)$.

When we look at all algorithms in the table 3.2, we can see that a good part of them is close to Dijkstra's algorithm. In table 3.2, Dijkstra's time and space complexities are computed assuming it uses a Fibonacci heap (we saw in 3.1.1 that this was the data structure ensuring the best performances).

- Breadth-First search is identical to Dijkstra's algorithm if edge weights are the same (= 1 for instance).
- Uniform-Cost search is similar to Dijkstra's algorithm as they both store nodes in a priority queue and their cost function is defined by the cumulative distance towards to destination node. The difference lies in their node storage management. Dijkstra stores all nodes while UCS inserts nodes during the execution of the algorithm, progressively.
- If we compare Bellman-Ford to Dijkstra's for solving the single-pair shortest path problem, Bellman-Ford is similar to Dijkstra's as they both relax edge weights in order to determine the distance from a node to the goal node, except that instead of using a priority queue to visit the nodes in order, it relaxes all nodes for |V| iterations.
- A*, as mentionned in 3.2.2, is equivalent to running Dijkstra's algorithm with the reduced cost $d'(x, y) = d(x, y) + h(y) - h(x)$, or with a heuristic function $h(x) = 0$.
- Johnson's algorithm is not similar to Dijkstra. However, it uses it in its procedure (as well as Bellman-Ford algorithm). Thus, it is worth mentioning that Dijkstra's algorithm has an impact on the complexities of Johnson's algorithm directly.

Nevertheless, even though Dijkstra's algorithm is very similar to the ones described hereabove, their differences are reflected through their performances, as we can see in the table 3.2.

As an example, we can notice that it may seem that A* is always more effective than Dijkstra. However, in A* and more generally in UCS, we also need to use the function "queue.extract_minimum" in the algorithm, which requires the same time as in Dijkstra's algorithm. Dijkstra is a special case for A* (when the heuristics are 0). However, A* uses a heuristic. It could be `manhattan distance` towards the goal for example (distance from one point to another in a grid layout [28]), but we have to ensure that it is an admissible heuristic. The performance of A* depends on this heuristic choice. With no assumption about the heuristic of A*, the complexity is the same as Dijkstra. In practice, when the computation of heuristic is time-consuming, Dijkstra can still be better. In terms of space, the search space of A* is smaller or equal to the search space of Dijkstra, which can lead to better execution time. Moreover, Dijkstra's algorithm is mainly used to tackle the single-source shortest path problem (3.3), which obviously requires more time.

Another remark is that Depth-first search and Breadth-first search share the same time complexity. In the worst-case scenario, both have to visit all vertices. DFS may be faster than BFS depending on the "bushyness" of the graph. In addition, DFS only has to store on a

single path, while BFS needs to keep every node in memory.

### 3.3.2 Objective

**Table 3.3** Algorithms goals (to solve the shortest path problem)

| | Role | | | |
|---|---|---|---|---|
| Algorithm names *SS = Single Source, SP = Single Pair, SD = Single Destination, AP = All pairs | SS | SP | SD | AP |
| Depth-First Search | | X | | |
| Breadth-First Search | | X | | |
| Uniform-Cost Search | | X | | |
| Iterative Deepening Search | | X | | |
| A* | | X | | |
| Dijkstra's algorithm | X | X | X | |
| Bellman-Ford | X | | X | |
| Floyd-Warshall | | | | X |
| Johnson's algorithm | | | | X |

We show the main goal(s) of all the algorithms in this table. However, some of these algorithms can be modified in order to take into account other shortest path problems. For instance, Dijkstra's algorithm is mainly used to solve the single-source shortest path problem (as described in 3.1.1). But notice that the single-destination shortest path problem can be solved by Dijkstra's algorithm because it is simply the reverse problem of the single-source problem.

Indeed, the single-destination (as mentioned in 1) consists in finding the shortest path from all vertices in a directed graph to a single destination vertex. Thus, it can be reduced to the single-source problem by reversing the arcs in the directed graph. Bellman-Ford, which also solves the single-source problem can therefore solve the single-destination problem as well.

As far as Floyd-Warshall and Johnson's algorithm are concerned, they both provide an algorithm to solve the all-pairs shortest path problem. Since the algorithm has to take all pairs of nodes into account, it is slightly more difficult to modify the other algorithms in order to tackle this problem. Precisely, if we take Dijkstra, it would require us to run this latter on each vertex of the graph. That is why $FW$ and $JA$ are more suitable for this purpose.

### 3.3.3 Completeness and Optimality

As a reminder :

- *Completeness* : The algorithms guarantees to find a solution when there is one

- *Optimality* : The algorithm is always able to find the optimal solution

**Table 3.4** Algorithms Completeness and Optimality conditions

| | Completeness Condition | Optimality Condition |
|---|---|---|
| DFS | Not complete due to potential loops | Not optimal due to potential loops |
| BFS | Complete if the branching factor is finite | Optimal if the step cost equals 1 |
| UCS | Complete if the branching factor is finite | Optimal if the step cost is positive |
| IDS | Complete if the branching factor is finite | Optimal if the step cost equals 1 |
| A* | Complete if the finite graph has no negative weights | Optimal as long as the heuristic is admissible |
| Dijkstra's algorithm | Complete if no negative weights | Optimal if no negative weights |
| Bellman Ford | Complete | Not optimal because it can detect negative cycles |
| Floyd Warshall | Complete | Optimal |
| Johnson's algorithm | Complete | Not optimal if it detects negative cycles |

All the algorithms are complete as we can see in table 3.4. But it is not always the case regarding specific conditions. Indeed, some of these algorithms are complete only if a certain condition is satisfied. For example, Depth-first search is complete only if the search space does not contain loops that would lead to an infinite depth search space. But it is explained in C.1.1 that it could be easily avoided. Another example is Breadth-first search, which is complete if the branching factor $b$ is finite in the tree. We assumed that the trees and graphs would be finite.

Regarding optimality, we can see that three algorithms are not optimal. They do not guarantee to always find the optimal solution. In the case of Depth-first search, it returns the first path to the goal state it encounters, which is not guaranteed to be the shortest one. Bellman-ford and Johnson's algorithm are not optimal neither. BF has the ability to detect negative cycles in a graph. Thus, the algorithm stops whenever it found one, meaning that it does not return the shortest path in the graph. Since Johnson's algorithm depends on Bellman-ford as it uses it at some point of its execution, it suffers from the same specificity. However, if the graph the algorithm is working with does not contain any negative cycles, then the optimal solution is guaranteed to be found by both. We observe that the common point of these non optimality and completeness is the presence of cycles in the search space or the infinite amount of possible options. That is why some algorithms only work with finite graphs or graphs that do not contain any negative cycles, such as Dijkstra's algorithm.

# Chapter 4

# Speed-up techniques

The main issue that we observe in table 3.2 of section 3.3 is the number of vertices and edges in the graphs or trees in which the algorithms are searching, is a limiting factor. Indeed, as introduced in section 1, there is a recent drastic increase in the amount of data. Classical algorithms are not efficient enough to deal with huge graphs, containing a tremendous amount of nodes and connections. This is the reason why several speed-up techniques appeared. We will first see how Dijkstra can be improved simply with a change of data structure, and then we will look at an overview of several existing speedup techniques. The list of those techniques is not exhaustive.

In the literature, there are two types of speed-up techniques (as stated in [29]). Namely,

- **Directed search/goal-directed** : It is based on the idea of directing the search towards the destination by increasingly expanding the search space
- **Hierarchical search** : It exploits the notion of *hierarchy* in a network. Indeed, some ways of a network can have different a level of importance regarding how likely they are chosen during the search process. For instance, highways are more favorable than regular roads because we can travel faster.

There are a lot of speedup techniques and we will only present the most prominent ones. Among those, we start by presenting *Bidirectional search*, *A* (goal directed)*, *A* with landmarks* - ALT (goal directed) and *Arc-Flags* (goal directed. Then, we overview hierarchical speedup techniques : *Reach*, *Highway Hierarchies*, *Highway Node Routing*, *Contraction hierarchies* (which is derived from Highway Node Routing), and *Transit Node routing*. In particular, we will focus on Bidirectional search, A* and A* with landmarks during the experiments later on.

## 4.1 Simple Dijkstra accelerations

We have seen in section 3.1.1 that Dijkstra's algorithm is the basic algorithm used to solve the shortest path problem. We also depicted in 3.1.1 the performance of Dijkstra's algorithm using Binary heap and Fibonacci heap. We will elaborate on these further in this section. In Dijkstra's algorithm, to choose the next vertice to expand, Dijkstra has to choose the vertex with the shortest distance and it uses a priority queue that is systematically updated each time. This queue is the data structure that can slow down the whole process. Of course, we

can improve this by using a `Heap`, which can considerably reduce the computation time of the algorithm. The main operations that are used in the algorithm are

- *Insert* : insert a new value with a certain priority
- *Extract minimum* : extract the minimum value of the set
- *Decrease* : decrease the priority of keys in the set

### 4.1.1 Array

With a simple array or a linked-list, we have to order the element in order to assign a priority to each one of them.

- *Insert* : take into consideration all nodes in the list until finding the position where the new element belongs to.
- *Extract minimum* : Simply take the value at the extremity
- *Decrease* : Find the element and replace its value.

### 4.1.2 Binary heap

Binary heap [14] is a data structure with the form of a binary tree and that respects two constraints :

1. It is a complete binary tree. That is, every level is fully filled with nodes, with the exception of the last floor which may contain fewer nodes.
2. The values associated with nodes follow a certain property. The value store in a node is either greater than or equal to ("max-heap") or less than or equal to (min-heap) the value of its children.

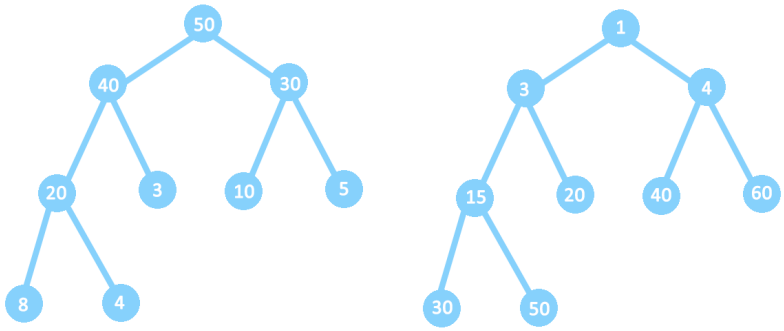



Figure 4.1: Binary Max-heap and Min-heap

- *Insert* : Add the element at the bottom of the heap, then compare the added element with its parent's value and swap if the order is not respected. Repeat that until the order is satisfied.
- *Extract minimum* : Take the value of the root node and then balance the heap accordingly.

- *Decrease* : Find the node with the given value, change the value, and then down-heapifying or up-heapifying in order to restore the heap property (multiple swaps if the order is not respected).

### 4.1.3 Fibonacci heap

Fibonacci heap [15] is a data structure that is constituted of a collection of heap-ordered trees. Each tree satisfies the property of a `Min-heap` or `Max-heap`. That is, the key of a child is always greater than or equal or lower than or equal to the key of its parent. This means that the top value (minimum or maximum) of the structure is always at the root of one of the trees. The advantage of this kind of structure is the flexibility offered by the number of trees. Indeed, the operations such as he *decrease-key* operation are faster. Besides, the degrees of nodes are kept low so that every node has degree at most $\mathcal{O}(log(n))$ and the size of each subtree whose node it of degree k is at least $F_{k+2}$, which is the $k$th Fibonacci number. The Fibonacci numbers [30] are the terms of a sequence of integers in which each term is the sum of the two previous terms.

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

The first Fibonacci numbers are :

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144$$

.

This degree property is possible thanks to the rule which states that at most one child can be cut from each non-root node. If one node has its two children removed, it must become the root of a new tree.

Additionally, the roots are linked using a circular doubly linked list and we keep a pointer to the root node that contains the minimum/maximum value. This allows for $\mathcal{O}(1)$ time for basic operations such as cutting two nodes, merging two trees, or finding the minimum or maximum value.

The children are also linked using a doubly-linked list. Each child has a pointer for both left and right siblings. Moreover, for each node, we maintain the number of its children and a mark. A node is marked whenever it loses a child. A node is unmarked if it loses no children.

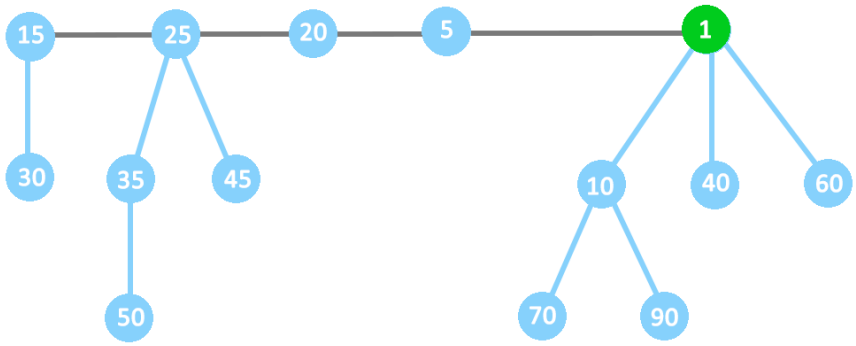



Figure 4.2: Fibonacci heap composed of several trees

- *Insert* : Create a new heap with one element and concatenate the lists of tree roots of the two heaps. This is done in constant time since concatenating two lists is simply an update of pointers in a bidirectional linked list.
- *Extract minimum* :
  - First, remove the root node containing the minimum value. The previous root's children become roots of new trees
  - Combine trees that have roots of the same degree. Repeat until every node has different degrees.
  - Check for all root nodes and find the one with the minimum value, update the pointer so that it points to this new node.

  Most of the speedups provided by Fibonacci heaps come from the fact that they postpone the consolidations of heaps as much as possible. Indeed, all other operations create new trees, and only "extract minimum" possibly needs to arrange the nodes in order to balance the trees and the nodes's degrees.
- *Decrease* :
  - Find the node with the given element (usually, we know the location of this element). Then, decrease its value.
  - if the heap is not violated after this change, nothing has to be executed
  - if the heap is violated, remove the node from its parent and transform it into a new tree.
    * If this parent is not a root node, mark it to specify that it has lost a child.
    * If the parent is already marked, it is removed and its parents are marked. This process is repeated until we reach either a root or an unmarked node. If the decreased node has the minimum value of the Fibonacci tree, the minimum pointer is updated.

**Performance**

**Table 4.1** Small optimizations comparison

| | Time Complexity | | |
|---|---|---|---|
| Data structure | Insert | Extract min | Decrease key |
| Ordered array | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| Unordered array | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Binary heap | $\mathcal{O}(log(n))$ | $\mathcal{O}(log(n))$ | $\mathcal{O}(log(n))$ |
| Fibonacci heap | $\mathcal{O}(1)$ | $\mathcal{O}(log(n))$ | $\mathcal{O}(1)$ |
| Ideal but impossible | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |

Now that we have the complexity of those operations using different data structures, we can compare the complexity of Dijkstra's algorithm using these data structures.

We have seen in 3.1.1 that generically, for any data structure of the set $Q$ used in the algorithm, the time complexity is

$$\mathcal{O}(|E|.T_d + |V|.T_{em})$$

where $T_d$ is the time needed to compute the *decrease* function (decrease the priority of keys in the set $Q$), and $T_{em}$ is the time needed to compute the *extract_minimum* operation (extract the minimum value from the set $Q$).

**Table 4.2** Dijkstra complexity with different data structures

| | Time Complexity |
|---|---|
| Data structure | Dijkstra |
| Ordered array | $\mathcal{O}(\|E\| + \|V\|^2)$ |
| Unordered array | $\mathcal{O}(\|V\|^2)$ |
| Binary heap | $\mathcal{O}((\|E\| + \|V\|)log\|V\|)$ |
| Fibonacci heap | $\mathcal{O}(\|E\| + \|V\|log\|V\|)$ |
| Ideal but impossible | $\mathcal{O}(\|E\| + \|V\|)$ |

In conclusion, we can slightly improve the complexity is Dijkstra using a `Fibonacci heap` instead of an ordinary array. Nevertheless, according to one of the inventors of Fibonacci heaps, *Michael Fredman* [31], stated that Fibonacci heaps have two main drawbacks :

1. They are hard to implement
2. In practice, they are not as efficient as other forms of heaps that are less efficient. The reason for that is the number of pointers needed to represent the data structure. Indeed, Fibonacci heaps require storage and manipulation of 4 different pointers, whereas other structures, namely Binary heap, Binomial heap, Pairing Heap, Brodal Heap and Rank Pairing heap, only need 2 or 3 pointers.

Despite the ameliorations that those data structures bring to Dijkstra's algorithm in terms of performance, it is not sufficiently efficient when we speak about large graphs containing real life-data leading to a tremendous amount of vertices and edges. Therefore, we need some more advanced techniques to speed it up. One simple example hereafter is the bidirectional search.

## 4.2 Bidirectional search

### 4.2.1 Dijkstra's search space

Dijkstra's algorithm runs in $\mathcal{O}(|E| + |V|log|V|)$, which is already a decent worst-case complexity since the running time grows a little bit more than linearly. Mikkel Thorup [32] showed in 1999 that Dijkstra's algorithm can be solved in linear time in a `undirected graph with positive integer weights` (linear in terms of the number of edges and vertices of the

graph). However, no algorithm can solve the single-source shortest path in a directed graph in linear time.

Now, let us consider how Dijkstra's algorithm explores the graph during its execution in order to have an idea about the search space it takes. We consider the `single source` shortest path problem to have a general view. We remember that the algorithm processes vertices in order of increasing distance. After processing every path once (relaxing the out-going edges), it goes to vertices at the next distance. Actually, Dijkstra's algorithm explores the graph in a circle shape like observed in [33]. That property can be explained by this lemma :

**Lemma 2.** *When a vertex u is selected via the "Extract minimum" operation, then*

$$dist[u] = d(s, u)$$

*where dist[u] is the tentative distance from starting node s to node u, namely the shortest path found so far, and d(s,u) is the actual shortest path from s to u.*

Consequently, when a vertex is extracted from the priority queue, all vertices at smaller distances have already been processed (out-going edges are relaxed). We proved it by induction in 3.1.1 when we had to show why Dijkstra actually works.

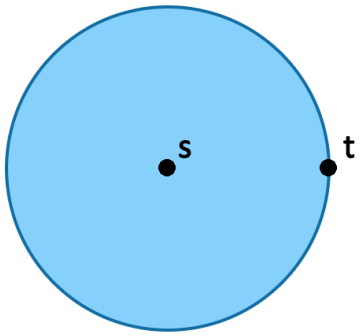



Figure 4.3: Search Space of Dijkstra

When we desire to find the shortest path from a source vertex $s$ to a goal vertex $t$, Dijkstra will generate a "circle" around $s$ until touching $t$. In the single-source shortest path problem, the circle will grow until finding the all shortest paths. In the case of the single-source shortest path problem, the circle will stop whenever it encounters the node $t$. We illustrate this here above.

Having that property in mind, we can introduce a Bidirectional search.

### 4.2.2 Bidirectional approach

Bidirectional search [29] is a search algorithm whose goal is to find the shortest path from an initial vertex to a destination vertex in a directed graph. It was first designed and implemented by **Ira Pohl, 1971** [34]. We have already discussed search algorithms in section 3.2. We reviewed several algorithms such as Breadth-first search (see C.1.2) that is used to search for a goal vertex starting from a source vertex. These algorithms are usually uni-directional, that is, they start from a given node and travel the graph in one direction. Contrariwise, bidirectional search, like its name indicates, runs two simultaneous searches in the graph :

- **Forward**: starting from the initial vertex
- **Backward**: starting from the destination vertex

The search ends when the two simultaneous searches meet at a certain vertex. As soon as the two searches meet, we can obtain the shortest path by combining half of the path from node $s$ to the middle point and the half from the meeting point to node $t$.

We can now adapt Dijkstra's algorithm to integrate this technique. In the algorithm, the two searches are not actually simultaneous. Instead, the algorithm alternates between the forward search and the backward search. In order to satisfy this, we have to know the out-going edges from each node, but also the in-going edges of each node. In the algorithm, we make one turn from $s$, followed by one turn from $t$, and so forth. When both sides encounter a vertex in the middle, the algorithm stops.

In practice, we can use a reverse graph for our bidirectional search. The reverse graph $\overleftarrow{G}$ of a graph $G$ is simply a graph with the same vertices as $G$ but with reversed edges of $G$. Formally,

**Reversed Graph** Reversed graph $\overleftarrow{G}$ for a graph $G$ is the graph with the same set of vertices $V$ and the set of reversed edges $\overleftarrow{E}$, such that for any edge $(u, v) \in E$ there is an edge $(v, u) \in \overleftarrow{E}$ and vice-versa.

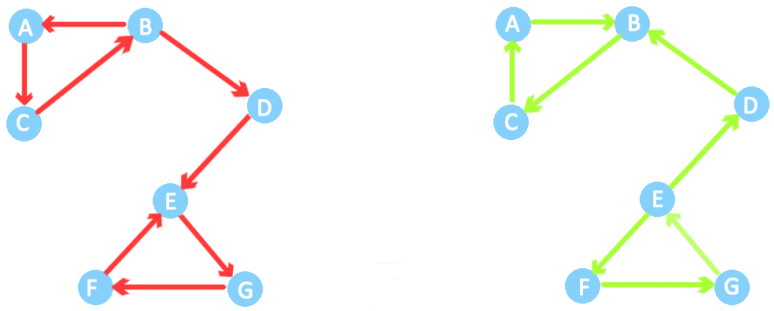



Figure 4.4: Directed graph and its reversed graph

**Algorithm**

The algorithm can thus be written as follows :

1. Construct the reversed graph $\overleftarrow{G}$ based on $G$
2. Run Dijkstra from vertex $s$ in $G$ and from vertex $t$ in $\overleftarrow{G}$
3. Alternate between Dijkstra steps in $G$ and in $\overleftarrow{G}$
4. Stop whenever some vertex $m$ is extracted by both searches in $G$ and $\overleftarrow{G}$
5. Combine shortest paths of both searches as the shortest path from $s$ to $t$

Given point **1**, we can wonder if constructing a new graph is optimal. Indeed, it may not be ideal if the graph contains a lot of vertices and edges, such as the whole map of a big city for instance. Using a reversed graph is one solution but actually, we can use the original graph only. However, we must add a property to it.

Given any node $n$, we must be able to find the set of parent nodes of $n$ such that there exists some operation that allows reaching $n$ from these parents. As far as the edges are concerned, if some edges have inverse arcs (arcs going in both directions), we simply have to consider the cost of the arc in the forward direction. Formally, if $n$ is a node with parent $p$, then $\omega(p, n) = \omega(n, p)$ where $\omega$ is the cost function. Additionally, given any node $n$, and assuming it has some parent $p$, we must be able to obtain the edge cost $\omega(p, n)$.

Now, let us look at point **4**. Bidirectional Dijkstra stops when the two distinct searches meet at a certain vertex $m$. However, $m$ is the first vertex that is processed in both forward and back searches. We can wonder if the path which is the combination of the shortest paths from $s$ to $m$ and from $t$ to $m$ is actually the correct shortest path of the graph. Suppose we have the following shortest path found from $s$ to $t$ :

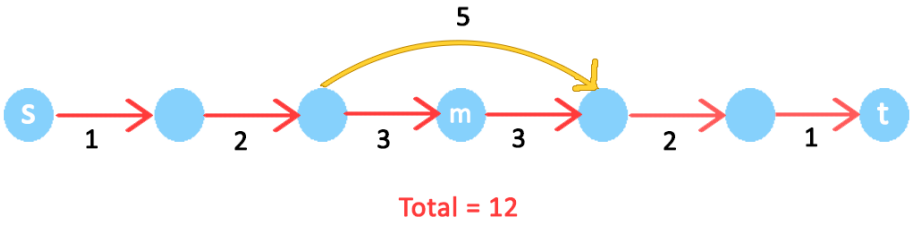



Figure 4.5: Path found by Bidirectional Dijkstra with total cost = 12

During the algorithm's execution, at some point, the forward search will be at node $v$ and the backward search will be at node $w$. If we strictly follow Dijkstra's algorithm, the next node to extract is node $m$ since it is the closest. If we look at the final path's cost assuming $m$ has been chosen, we have a total cost of 12. However, we clearly see that the path that includes the edge (v,w) instead of node $m$ is shorter. Its cost is 11.

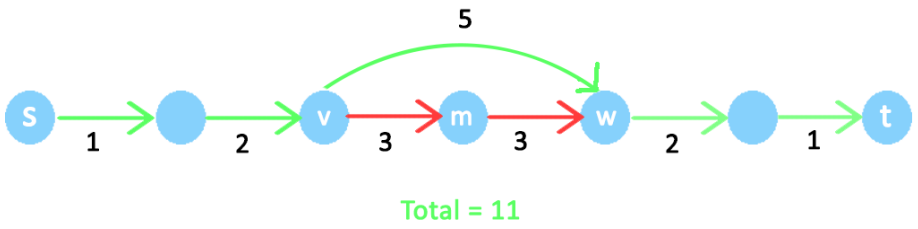



Figure 4.6: Actual Shortest path of Bidirectional Dijkstra with total cost = 11

Thus, we need to define a specific termination condition for the bidirectional version of Dijkstra's algorithm. But first, we need to show what is the property that allows us to find this termination condition.

### Properties

From the observation we made with figure 4.6, let us define the following lemma :

**Lemma 3.** *Let $dist^F$ be the tentative distance in the forward search of Bidirectional Dijkstra from $s$ in the graph $G$ and $dist^B[u]$ be the tentative distance in the backward search from $t$ in the reversed graph $\overleftarrow{G}$. After some node $m$ is processed both by the forward and the backward search in $G$ and $\overleftarrow{G}$, there exists some shortest path from $s$ to $t$ that passes through some other node $u$ which is processed either in $G$, in $\overleftarrow{G}$ or both, and*

$$d(s,t) = dist^F + dist^B[u]$$

.

This lemma means that despite the fact that there is a vertex $m$ in the middle where the two searches meet, there exist another node $u$ that lies either in the search space of the forward search of the search space of the backward search and the shortest path of the whole graph is passing through it.

*Proof.* From the lemma 2, we know that when a node has been processed (its out-going edges are relaxed), it means that the tentative distance from the source node $s$ to this node is actually the real shortest distance. Thus, if the node $u$ is processed in the forward search for instance, then we know by the lemma 2 that the tentative distance is already the actual shortest distance. But if node $u$ is not processed in the backward search, then the total path from $s$ to $t$ is not guaranteed to be the shortest. The lemma 3 is thus the generalization of lemma 3 but for the bidirectional version. We can show that for some node $u$, the shortest path passes through this node and the estimated distances are correct by the moment our forward search meets our backward search.

- First, let us consider the case where there is a node $u$ that is not processed by forward search and not processed by the backward search. We have to show that the path passing through this node $u$ is not the shortest and there is a node $m$ which belongs to the real shortest path from $s$ to $t$. In the illustration hereafter, we represent the search spaces of both forward and backward searches. We know that $m$ is already processed in

the forward search since it is the first node both searches encounters, it is the meeting point in the middle. It means that the distance from $s$ to $m$ is at most the distance from $s$ to $u$. The path's distance from $s$ to $m$ is less or equal than the path's distance from $s$ to $u$. Now, if we consider the backward search, we know that $m$ is also processed, but $u$ is still not processed yet. Therefore, the path's distance from $m$ to $t$ is less or equal to the path's distance from $u$ to $t$. It means that the path going from $s$ to $t$ passing through $m$ has a distance that is less or equal to the distance of the path from $s$ to $t$ passing through $u$. Consequently, it means that $u$ cannot be out of the search spaces.

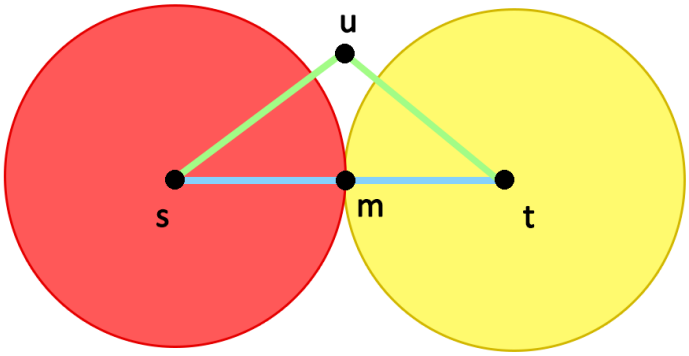



Figure 4.7: node $u$ not processed

- Then, let us consider the case where there is no node that is not processed by both forward and backward searches. Like the previous case, the node $m$ is already processed by both.

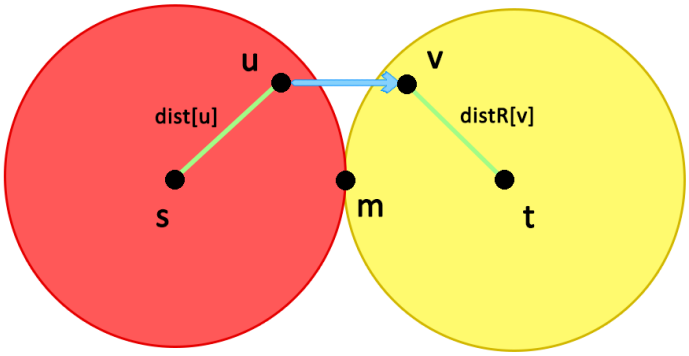



Figure 4.8: node $u$ processed

Let us consider any shortest path from $s$ to $t$ and the last vertex $u$ on such shortest path that is processed by the forward search. By lemma 2, we know that the tentative distance $dist^F$ from $s$ to $u$ is already correct. Let us consider the next vertex on the

path, which is $v$. $v$ has to be processed by backward search since it cannot be processed by the forward search as it already processed the last vertex $u$. $v$ cannot be unprocessed by both searches. $v$ lies in the search space of backward search. It means that the tentative distance $dist^B[u]$ from $t$ to $v$ is also correct. Now, we have that the total distance of the path from $s$ to $t$ passing through $u$ and $v$ is

$$d(s,t) = dist^F + \omega(u,v) + dist^B[v]$$

where $\omega(u,v)$ is the weight function and that represents the length of edge (u,v) here. We have to show that

$$d(s,t) = dist^F + \omega(u,v) + dist^B[v] = dist^F + dist^B[u]$$

in order to prove that this path is shorter than the path containing node $m$.

- Trivially, $dist^F = dist^F$ and the remaining equation is $\omega(u,v) + dist^B[v] = dist^B[u]$
- On one hand, we already know by the definition of tentative distance (estimated distance) that $dist^B[u] \geq d(u,t)$. The distance estimate in any moment is always greater of equal to the real distance. On the other hand, we know that node $v$ has already been processed in the reverse search. Therefore, it means that edge (u,w) has been relaxed. Then,

  $$dist^B[u] \leq \omega(u,v) + dist^B[v]$$

  Thus, the estimated distance of $u$ is both greater than or equal to the true shortest path from $u$ to $t$ and also less than or equal to the shortest path from $u$ to $t$.

  $$d(u,t) \leq dist^B[u] \leq \omega(u,v) + dist^B[v]$$

  Consequently,

  $$\omega(u,v) + dist^B[v] = dist^B[u]$$

□

### 4.2.3 Termination Condition

Given that lemma 3, we can define a proper termination condition for bidirectional Dijkstra's algorithm. Andrew Goldberg [35] defines these termination conditions as follows:

The algorithm alternates turns of forward and backward search until meeting node $m$ in the middle at some point. During this process, it has to save vertices that are processed. In the end, we take those vertices that are processed in either one of the searches and we search for the vertex that minimizes the sum of the distance estimate of the forward search plus the distance estimate of the backward search. We know that the minimum sum is the actual distance of the shortest path.

$$\min d(s,t) = dist^F + dist^B[u]$$

More generally, we can keep track of the forward and reversed distances. We denote those distances by $L^F(u)$ and $L^B(u)$. Let us define $\mu$ as the length of the minimal path founded so far. When an edge $(u, v)$ is scanned,

- $u$ is processed in the forward search and $u \in S^F$
- $v$ is processed in the backward search and $v \in S^B$

We update $\mu$ if

$$L^F(u) + \omega(u, v) + L^B(v) < \mu$$

We stop when

$$top^F + top^B \geq \mu$$

where $top^F$ and $topB$ are the minimal value of the priority queue of respectively the forward and the backward search.

**Correctness**

We can convince ourselves that this condition is correct in order to find the shortest path. Suppose the algorithm stops when the previously stated condition is satisfied. Suppose there is a path $P$ of length $\mu' < \mu$. It means that there exist an edge (w,x) in $P$ so that :

$$d(s, w) < top^F$$

and

$$d(x, t) < top^B$$

since

$$\mu' = d(s, w) + \omega(w, x) + d(x, t) < top^F + top^B$$

But that would mean that $w$ is already scanned in the forward search and $x$ is already scanned in the backward search. But it is not possible because it would mean that at soon as vertex $x$ is scanned, $P$ should have been found. It was not. So, it is a contradiction and $P$ does not exist. Consequently, the termination condition is correct.

We can describe a pseudocode of the bidirectional version of Dijkstra's algorithm using a graph and its reversed version hereafter :

**Algorithm 8** Relax

```
1: function RELAX(u, v, dist, prev)
2:     if dist[v] > dist[u] + ω(u, v) then
3:         dist[v] ← dist[u] + ω(u, v)
4:         prev[v] ← u
5:     end if
6: end function
```

**Algorithm 9** Bidirectional Dijkstra

```
1: function BidirectionalDijkstra(G, s, t)
2:     G← ← ReverseGraph(G)
3:     unvisited^F ← priority queue forward
4:     unvisited^B ← priority queue backward
5:     visited^F ← ∅
6:     visited^B ← ∅
7:     dist^F[s] ← 0
8:     dist^B[t] ← 0
9:     for all v ∈ G and w ∈ G← do                    ▷ And v ≠ w ≠ start node
10:        dist^F[v] ← ∞                              ▷ tentative distance from start to v
11:        dist^B[w] ← ∞                              ▷ tentative distance from dest to w
12:        prev^F[v] ← ⊥                              ▷ Predecessor of v is None
13:        prev^B[w] ← ⊥                              ▷ Predecessor of w is None
14:        unvisited^F.add_with_priority(v, dist^F[v])
15:        unvisited^B.add_with_priority(v, dist^B[w])
16:        unvisited^F.add(v)
17:        unvisited^B.add(w)
18:    end for
19:    c^F ← s                                        ▷ current node = start node initially
20:    do
21:        c^F ← unvisited^F.extract_minimum()        ▷ vertex with min tentative dist
22:        Process(c^F, G, dist^F, prev^F, visited^F)
23:        if c^F ∈ visited^B then
24:            return ShortestPath(s dist^F, prev^F, visited^F, t, ...)
25:        end if
26:        c^B ← unvisited^B.extract_minimum()
27:        repeat symmetrically for c^B as for c^F
28:    while True
29: end function
```

**Algorithm 10** Process

```
1: function Process(u, G, dist, prev, visited)
2:     for (u,v) ∈ E do
3:         Relax(u,v,dist,prev)
4:     end for
5:     visited.append(u)
6: end function
```

**Algorithm 11** Bidirectional Dijkstra find Shortest Path

```
1:  function ShortestPath(s, dist^F, prev^F, visited^F, t, dist^B, prev^B, visited^B)
2:      distance ← ∞
3:      best_u ← ⊥                                                    ▷ None
4:      for u in visited^F + visited^B do
5:          if dist^F[u] + dist^B[u] < distance then
6:              best_u ← u
7:              distance ← dist^F[u] + dist^B[u]
8:          end if
9:      end for
10:     path ← ∅
11:     last ← best_u
12:     while last ≠ s do
13:         path.append(last)
14:         last ← prev^F[last]
15:     end while
16:     path ← Reverse(path)
17:     last ← best_u
18:     while last ≠ t do
19:         last ← prev^B[last]
20:         path.append(last)
21:     end while
22:     return (distance, path)
23: end function
```

### 4.2.4 Search spaces

Now, we can look at the search space covered by both Dijkstra's algorithm and Bidirectional Dijkstra. As we can see in the illustration below, Dijkstra's algorithm covers and explore all nodes in the big circle with center $s$ and radius $t$.

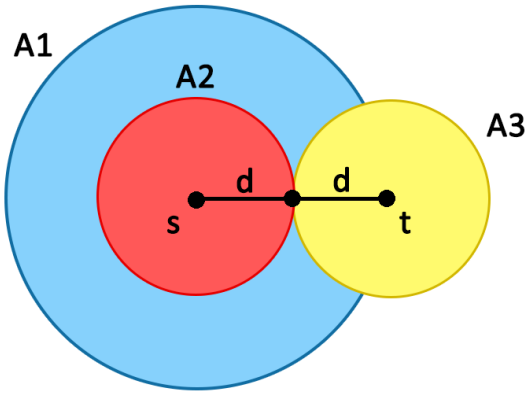



Figure 4.9: Search Space of Dijkstra and bidirectional Dijkstra

As far as bidirectional Dijkstra's algorithm is concerned, it creates two distinct circles of roughly the same radius and the algorithm stops whenever those two circles touch each other. The area covered by the two circles seems to be proportional to the number of scanned vertices of the bidirectional search. We can try to add some measures to this illustration. Let us denote

- the big circle by A1,
- the two other circles by A2 and A3,
- the distance from $s$ to the middle point $m$ (which is roughly equal to the distance from the middle point $m$ to $t$) by $d$

We can compute the areas of each circle :

- area(A1) = $4\pi d^2$ since the radius of A1 is $2d$
- area(A2) = $\pi d^2$
- area(A3) = $\pi d^2$

Thus, the area covered by the bidirectional search is equal to the sum of the two circles A2 and A3 :

$$2.\text{area}(A2) = 2\pi d^2$$

Consequently, we can see that the area covered by Dijkstra's algorithm is approximately twice bigger than its bidirectional version.

### 4.2.5 Speedup on social networks

Previously, we analyzed the speed-up factor offered by bidirectional search on road networks. Let us now consider social networks just to see if the speed-up is different. In 1929, the mathematician Frigyes Karinthy made a Small World conjecture [36] called "Six handshakes" or "Six degrees of separation". He says that when we have to send a message from any person on earth to any other person on earth, using only personal connections in between, we only need at most 6 "handshakes" to achieve that goal, which means that there must be 5 different human beings in the intermediary process. Based on this idea, we can consider a social network as a model. If a person has on average 100 friends, and that person's friends also, then we already consider 10 000 persons. If we consider friends of friends of friends, we will have 1 million, and so forth. The number increases exponentially. We can be interested to find the minimum number of handshakes we need to process in order to send a message from one person to another. It is similar to finding the shortest path in a graph. Therefore, we can use bidirectional search to speed this up. Let us consider person A and person B.

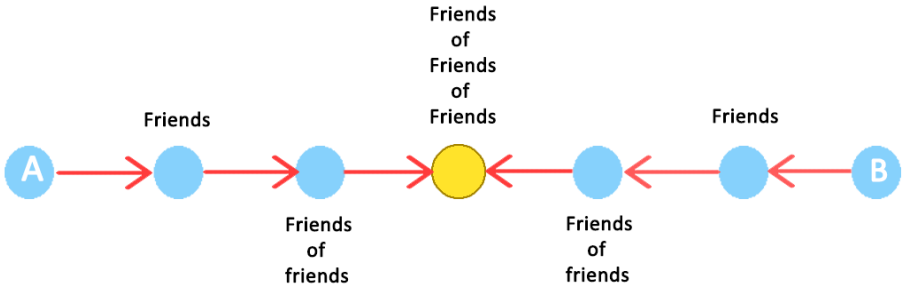



Figure 4.10: 6 handshakes graph

Given the conjecture, we know that we only need at most 6 handshakes between two persons. It means that at some point, A and B will have some friends in common. We can consider the worst-case scenario where A and B have to consider friends of friends of friends each. It would mean that each one of them will explore at most approximately 1 million friends assuming one person has on average 100 friends. In total, 2 million people will be solicited in the search. Consequently, it will be approximately **1000 times** faster than simply using Dijkstra's algorithm, at least in the number of friends scanned.

### 4.2.6 Performance

#### Time complexity

Bidirectional search significantly reduces the search space of Dijkstra's algorithm [37]. In terms of performance regarding search problems, we can see in C.1.2 and C.1.1 that Breadth-first search and Depth-first search complexity is $\mathcal{O}(b^d)$, where $b$ is the branching factor of the the tree produced by the search and $d$ is the distance from source $s$ towards goal $t$. When we use two searches, the complexity becomes

$$\mathcal{O}(b^{\frac{d}{2}})$$

for each search and the total complexity is

$$\mathcal{O}(b^{\frac{d}{2}} + b^{\frac{d}{2}})$$

which means that that the search space is decreased by $\frac{1}{2}b^{\frac{d}{2}}$.

In practice, the speed-up of the Bidirectional version of Dijkstra's algorithm depends on the graph. For route networks, it is approximately a 2x speed-up. For social networks, it can be thousands of times faster. Additionally, we have to take care of the representation of the graph. If we use a reversed graph, the memory consumption is also multiplied by 2 as we have to store some auxiliary graph.

#### Completeness

It depends on the search used in both searches. If we use Breadth-first search, bidirectional search is complete as it is guaranteed to always find a solution when there is one. If we use Dijkstra's algorithm, we have seen in table 3.4 of section 3.3 that the algorithm is optimal with the condition that the graph does not contain any negative edge weights.

**Optimality**

It also depends on the search used in both searches. If we use Breadth-first search, bidirectional search is optimal as it is guaranteed to always find the optimal solution. If we use Dijkstra's algorithm, the algorithm is optimal with the same condition as completeness.

## 4.3 Goal-Directed Speedup techniques

Another optimization of Dijkstra's algorithm is the A* algorithm that we overviewed in section 3.2.2. We saw that this algorithm is a special case of Dijkstra's algorithm and its performance can be better depending on the choice of the heuristic function. However, A* can be optimized by some speed-up techniques. But first, we will recapitulate the properties of the algorithm from a different perspective. Then, we will introduce the bidirectional approach of the algorithm, as we did with Dijkstra's algorithm in section 4.2. Finally, we will present the use of landmarks that can significantly speed up shortest path query times.

### 4.3.1 Potential function

A *Potential function* [38] is a function that maps vertices of the graph to numbers. Denote a potential function by $\pi$.

$$\pi : V \to \mathbb{R}$$

This function defines new edge weights given by the following reduced cost :

$$\omega_\pi(u, v) = \omega(u, v) - \pi(u) + \pi(v)$$

where $\omega(u, v)$ is the edge weight, namely the length of the path from vertex $u$ to vertex $v$. If we replace $\omega(u, v)$ by $\omega_\pi(u, v)$, the problem remains the same. Indeed, for any two vertices $u$ and $v$, the length of the path between those edges only changes by a constant, which is $\pi(v) - \pi(u)$. Thus, the shortest path in the graph using $\omega_\pi$ is the same as the shortest path using $\omega$. We can write the following lemma :

**Lemma 4.** *For any potential function $\pi : V \to \mathbb{R}$, for any pair of vertices $s$ and $t$ in the graph and any path $P$ between them,*

$$\mathcal{W}_\pi(P) = \mathcal{W}(P) - \pi(s) + \pi(t)$$

*Proof.* We have to show that

$$\mathcal{W}_\pi(P) = \sum_{i=1}^{k-1} \omega_\pi(v_i, v_{i+1}) = \mathcal{W}(P) - \pi(s) + \pi(t)$$

Let us consider the path

$$P = (s = v_1, v_2, \ldots, v_k = t)$$

Then,

$$\mathcal{W}_\pi(P) = \sum_{i=1}^{k-1} \omega_\pi(v_i, v_{i+1}) \tag{4.1}$$
$$= \omega(v_1, v_2) - \pi(v_1) + \pi(v_2) \tag{4.2}$$
$$+ \omega(v_2, v_3) - \pi(v_2) + \pi(v_3) \tag{4.3}$$
$$+ \dots \tag{4.4}$$
$$+ \omega(v_{k-2}, v_{k-1}) - \pi(v_{k-2}) + \pi(v_{k+1}) \tag{4.5}$$
$$+ \omega(v_{k-1}, v_k) - \pi(v_{k-1}) + \pi(v_k) \tag{4.6}$$
$$= \sum_{i=1}^{k-1} \omega(v_i, v_{i+1}) - \pi(v_1) + \pi(v_k) \tag{4.7}$$
$$= \mathcal{W}(P) - \pi(s) + \pi(t) \tag{4.8}$$

We can notice that line 2 is equal to $\omega_\pi(v_1, v_2)$, by definition. We can further notice that $\pi(v_2)$ of line 2 and $-\pi(v_2)$ of line 3 cancel each other. The same happens for all other variables, except for $-\pi(v_1)$ of line 2 and $\pi(v_k)$ of line 8. Those two variables are the starting and destination node in the path $P$. That proves the lemma 4 and we know that the shortest path in the initial graph and in the new graph is the same, although their lengths differ by some constant depending on the potential value of starting and ending vertex.

□

### 4.3.2 A*

Consider some potential function $\pi$. If we run Dijkstra's algorithm using this function and with edge weights $\omega_\pi$, we get A* algorithm **(Hart et al., 1968, [26])**. Like stated in section 4.3.1, the shortest path found by the algorithm is the same as if we ran it without considering those edge weights. However, we need a specific condition for the algorithm to work properly.

**Definition 1.** *A potential function $\pi$ is feasible if $\omega_\pi(u, v)$ is non-negative for any edge $(u, v)$.*

In A* algorithm, the potential function $\pi(v)$ is intuitively the estimated distance from node $v$ to destination node $t$. As introduced in section 3.2.2, such potential function must be a lower bound on the actual distance $d(v, t)$. It must be an admissible function, meaning that it never overestimates the real distance towards the goal node.

In each step of the algorithm, A* chooses the vertex $v$ that minimizes

$$dist[v] - \pi(s) + \pi(v)$$

Since $\pi(s)$ is common to all vertex $v$, we are minimizing

$$dist(v) + \pi(v)$$

Knowing that potential function $\pi(v)$ is an estimation of $d(v, t)$, we have that algorithm is minimizing the estimation of $d(s, v) + d(v, t)$ at each step. It is equivalent as the notation we

gave in section 3.2.2 :

$$\min f(n) = g(n) + h(n)$$

where $g(n)$ is the cost of the path from start node to $n$ and $h(n)$ is a heuristic function that estimates the cost of the shortest path from $n$ to $t$.

In terms of performances, it depends on the choice of potential function.

- *Best case* : if the lower bound on the distance to the target node is equal to the distance from vertex $v$ to destination node, $\pi(v) = d(v, t)$ for all nodes $v$, then edge weights will be equal to 0, $\omega_\pi(u, v) = 0$ if and only if edge $(u, v)$ lies on the actual shortest path. In this case, A* will only choose edges with weight 0.
- *Worst case* : if the potential function $\pi(v) = 0$ for all vertex $v$. In this case, it is the same as running Dijkstra's algorithm.

Ideally, we want the lower bound on the distance to the target node to be as tight as possible in order to reduce the number of scanned vertices in the process. It is thus important to choose a potential function wisely.

### 4.3.3 Lower bound

Now, we can be interested in determining what can be a good feasible potential function we can use in A* algorithm. First, let us define a lemma :

**Lemma 5.** *If $\pi$ is feasible and $\pi(t) \leq 0$, then $\pi(v) \leq d(v, t)$ for any $v$.*

We said previously that the potential function of a vertex must be a lower bound on the distance from this vertex to the destination vertex. Lemma 5 states that whenever a potential function is *feasible* and the potential function of the destination vertex is non-positive, then this potential function is indeed a lower bound.

*Proof.* We have the hypothesis that $\pi(t) \leq 0$ and that $\pi$ is feasible. Also, by definition of feasibility, $\omega_\pi(x, y) \geq 0$ for any $x$ and $y$, since $\pi$ is feasible. Then, we have that for any path $P$, $\omega_\pi \geq 0$ since each edge of this path is non negative. Let us now consider a shortest path from vertex $v$ to $t$,

$$P = (v, v_1, v_2, \ldots, t)$$

Then,

$$\begin{aligned}
&\mathcal{W}_\pi(P) \geq 0 && (4.1)\\
&\Leftrightarrow \mathcal{W}(P) - \pi(v) + \pi(t) \geq 0 && (4.2)\\
&\Leftrightarrow \mathcal{W}(P) - \pi(v) + \pi(t) \leq \mathcal{W}(P) - \pi(v) && (4.3)\\
&\Leftrightarrow \mathcal{W}(P) - \pi(v) \geq 0 && (4.4)\\
&\Leftrightarrow \mathcal{W}(P) \geq \pi(v) && (4.5)\\
&\Leftrightarrow \pi(v) \leq d(v, t) && (4.6)
\end{aligned}$$

In line 3, we know that $\pi(t) \leq 0$ from the hypothesis. At line 5, we have that $\mathcal{W}(P)$ is the shortest path from node $v$ to $t$, thus it is equal to $d(v, t)$. At the end, we obtain our lower

bound $\pi(v) \leq d(v,t)$. □

We can now enumerate some usable potential functions.

### Euclidean distance

**Definition 2.** *[39] The Euclidean distance between two points in the Euclidean space is the length of the line segment between those points.*

It can be computed using the Pythagorean theorem with Cartesian coordinates.

**Lemma 6.** *Consider a road network on a 2-dimensional map with each vertex $v$ having coordinates $(v.x, v.y)$. The potential function using Euclidean distance between $v$ and $t$ $\pi(v) = d^E(v,t) = \sqrt{(v.x - t.x)^2 + (v.y - t.y)^2}$ is feasible, and $\pi(t) = 0$*

*Proof.* By lemma 5, $d^E(v,t)$ must be a lower bound on $\omega(u,v)$. It is the case since we are using the straight line segment between two points. $\omega(u,v) \geq d^E(u,v)$. Besides, $\pi(t) = 0$ is trivial since the the distance from node $t$ to node $t$ is equal to 0, $\pi(t) = d^E(t,t) = 0$.

$$\pi(u) = d^E(u,t) \tag{4.1}$$
$$\Leftrightarrow d^E(u,t) \leq d^E(u,v) + d^E(v,t) \tag{4.2}$$
$$\Leftrightarrow d^E(u,v) + d^E(v,t) \leq \omega(u,v) + \pi(v) \tag{4.3}$$
$$\Leftrightarrow \pi(u) \leq \omega(u,v) + \pi(v) \tag{4.4}$$
$$\Leftrightarrow \omega(u,v) - \pi(u) + \pi(v) \geq 0 \tag{4.5}$$
$$\Leftrightarrow \omega^E_\pi(u,v) \geq 0 \tag{4.6}$$

Line 2 is the triangle inequality property applied on $d^E(u,t)$. [40]

**Definition 3.** *The triangle inequality is a mathematical basic property that states that for any triangle, the sum of lengths of any two sides is greater than or equal to the third side's length. $z \leq x + y$*

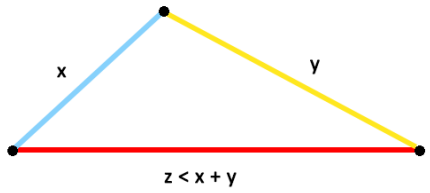



Figure 4.11: Triangle inequality

Here, we are talking about triangle inequality with respect to the shortest path in the graph, not any other metric included in the Euclidean space.

In line 3, we know that $d^E(u,v) \leq \omega(u,v)$ like said before and $d^E(v,t)$ is exactly equal to the potential function of $v$, $\pi(v)$. At line 6, we have concluded that any edge $(u,v)$ has a non-negative edge weight using the Euclidean potential function. It means that this potential function is *feasible*, by definition. □

### Manhattan distance

**Definition 4.** *[28] The Manhattan distance between two points is the distance between those points, traveled by a taxi in city where roads are arranged according to a network or grid. It is defined as the sum of the absolute differences of Cartesian coordinates. That is, given two vectors x and y in an n-dimensional vector space with Cartesian coordinates, the distance $d^M$ is*

$$d^M = \sum_{i=1}^{n} |x_i - y_i|$$

For a 2 dimensional space, we have 2 points $(x_1, y_1)$ and $(x_2, y_2)$ and the distance is

$$d^M = |x_1 - x_2| + |y_1 - y_2|$$

According to [41], the manhattan distance heuristic can be admissible. We can also reason by contradiction.

*Proof.* Suppose we have a grid network where the goal is to find the shortest path between two distinct nodes $s$ and $t$ using Manhattan distance heuristic. Trivially, $h(t) = 0$. Let us assume that $h(s) > C$ where C is the number of actions needed to reach $t$. Thus, we have that $h(s) - C > 0$. Since the goal node can be reached in C actions, we have $h(t) \geq h(s) - C > 0$. This contradicts our initial hypothesis because $h(t)$ should be equal to 0. Consequently, we need to have $h(s) \leq C$ for all starting node $s$. □

Having this admissibility in mind, we can apply the same reasoning as for the Euclidean distance heuristic and we have a feasible potential function. We can also notice that Euclidean distance is the lowest distance possible since it is the straight line segment between two points. Therefore, Manhattan distance is an upper bound of Euclidean distance, even though these two distances are not applicable in the same kind of graph.

### Potential function combination

According to [38] and [42], the average or any convex linear combination of feasible potential functions is also feasible. Taking the maximum or the minimum between feasible potential functions is also feasible. For instance, let us consider the maximum potential function.

**Lemma 7.** *If $\pi_1$ and $\pi_2$ are feasible potential functions, then $\pi^{Max} = max(\pi_1, \pi_2)$ is a feasible potential function*

*Proof.* Consider edge $(u,v) \in E$, the set of edges of a graph. By the feasibility of $\pi_1$ and $\pi_2$, $\omega(u,v) - \pi_1(u) + \pi_1(v) \geq 0$ and $\pi_2$, $\omega(u,v) - \pi_2(u) + \pi_2(v) \geq 0$. Let us suppose $\pi_1(u) \geq \pi_2(u)$. If $\pi_1(v) \geq \pi_2(v)$, then

- $\omega(u,v) - \pi^{Max}(u) + \pi^{Max}(v) = \omega(u,v) - \pi_1(u) + \pi_1(v) \geq 0$

- $\omega(u,v) - \pi^{Max}(u) + \pi^{Max}(v) = \omega(u,v) - \pi_1(u) + \pi_2(v) \geq \omega(u,v) - \pi_1(u) + \pi_1(v) \geq 0$ Otherwise.

The reasonning is symmetric if $\pi_2(u) \geq \pi_1(u)$. □

### 4.3.4 Bidirectional A*

In section 4.2, we have introduced a bidirectional version of Dijkstra's algorithm. We can also review a bidirectional version of A* algorithm [43], which requires a small modification. We have to use potential functions :

- *Forward potential function* : $\pi^F(v)$ that estimates $d(v,t)$
- *Backward potential function* : $\pi^R(v)$ that estimates $d(s,v)$

However, with the introduction of these new potential functions, a problem arises. We have seen in Bidirectional Dijkstra's algorithm that although it contains two different searches, the edge weights remain the same in the original graph and the reversed graph. That is, The length of any edge $(u,v)$ of the original graph is the same as in the reversed graph. To ensure this property, we have to satisfy equality.

$$\omega_\pi^F(u,v) = \omega(u,v) - \pi^F(u) + \pi^F(v) \tag{4.1}$$
$$\omega_\pi^R(u,v) = \omega(u,v) - \pi^R(v) + \pi^R(u) \tag{4.2}$$
$$\omega_\pi^F(u,v) = \omega_\pi^R(u,v) \tag{4.3}$$
$$\Leftrightarrow \pi^F(u) + \pi^R(u) = \pi^F(v) + \pi^R v \tag{4.4}$$

In order to satisfy line 4, we can define a new potential function so that $\pi^F(u) + \pi^R(u)$ is a constant that forces $\pi^F(v) + \pi^R v$ to be equal to the same constant. We can use the average value of potential function for example

- $p^F(u) = \frac{\pi^F(u) - \pi^R(u)}{2}$
- $p^R(u) = -p^F(u)$

With those potential functions, we have $p^F(v) + p^R(v) = 0$ for any node $v$.

We can prove that this new potential function is *feasible*, like we did in section 4.3.3.

**Lemma 8.** *If $\pi^F$ is feasible for forward search and $\pi^R$ is feasible for backward search, then $p^F = \frac{\pi^F - \pi^R}{2}$ is a feasible potential function for forward search.*

*Proof.*

$$\omega(u,v) - \pi^F(u) + \pi^F(v) \geq 0 \tag{4.1}$$
$$\omega(u,v) - \pi^R(v) + \pi^R(u) \geq 0 \tag{4.2}$$
$$\Leftrightarrow 2\omega(u,v) - (\pi^F(u) + \pi^R(u)) + (\pi^F(v) + \pi^R(v)) \geq 0 \tag{4.3}$$
$$\Leftrightarrow \omega(u,v) - \frac{\pi^F(u) - \pi^R(u)}{2} + \frac{\pi^F(v) - \pi^R(v)}{2} \geq 0 \tag{4.4}$$
$$\Leftrightarrow \omega(u,v) - p^F(u) + p^F(v) \geq 0 \tag{4.5}$$

Lines 1 and 2 are correct since both $\pi^F$ and $\pi^R$ are feasible. In line 3, we summed inequalities of lines 1 and 2 and rearranged the terms. Finally, we find that $\pi^F(x)$ is feasible for any node $x$ since edge weights induced by this potential function are non-negative, $\omega_p(u,v) \geq 0$. □

In conclusion, in order to apply bidirectional search to A* algorithm, we need to first choose a feasible potential function for forward and backward search (Euclidean distance for instance) and then compute the new potential function's values for each node. When it is done, we can run the algorithm. The termination condition is the same as for bidirectional Dijkstra's algorithm. That is, we keep the shortest distance found so far $\mu$ and as soon as a new scanned edge is processed and $\mu$ is no longer optimised, the algorithm stops.

### 4.3.5 Landmarks - ALT

We have seen in section 4.3.3 some feasible potential functions. However, we can use landmarks in order to obtain a more efficient potential function.

**Definition 5.** *A landmark is a fixed vertex $\lambda \in V$, where $V$ is the set of vertices of a graph.*

**Lemma 9.** *The potential function $\pi^L(v) = d(\lambda,t) - d(\lambda,v)$ is feasible and $\pi(t) = 0$.*

*Proof.*

$$\omega_\pi^L(u,v) = \omega(u,v) - \pi(u) + \pi(v) \tag{4.1}$$
$$\Leftrightarrow \omega_\pi^L(u,v) = \omega(u,v) - d(\lambda,t) + d(\lambda,u) + d(\lambda,t) - d(\lambda,v) \tag{4.2}$$
$$\Leftrightarrow \omega_\pi^L(u,v) = \omega(u,v) - d(\lambda,u) + d(\lambda,v) \tag{4.3}$$
$$\Leftrightarrow \omega_\pi^L(u,v)\omega(u,v) - d(\lambda,u) + d(\lambda,v) \geq 0 \tag{4.4}$$
$$\Leftrightarrow \omega_\pi^L(u,v) \geq 0 \tag{4.5}$$

In line 2, we can simplify the $d(\lambda,t)$ and we get line 3. Line 4 is possible with the triangle inequality property. Additionally, $\pi^L(t) = d(\lambda,t) - d(\lambda,t) = 0$.

□

Given this new potential function, we can use it to speed up A* algorithm. We have to select an arbitrary number of landmarks among all nodes in $V$ and compute their distances to all other vertices. Indeed, the potential functions $\pi^L(v) = d(\lambda,t) - d(\lambda,v)$, where $d(\lambda,t)$ is the actual distance from landmark $\lambda$ to node $t$ and $d(\lambda,v)$ is the distance from landmark $\lambda$ to node $v$. To determine these distances, we have to do a preprocessing before running A* algorithm. A* algorithm combined with the use of landmarks and triangle inequality is called `ALT` algorithm *(Goldberg et al., 2003)*.

**Algorithm 12** ALT

```
1: function ALT(G, s, t)
2:     L ← generate_landmarks(G, k)                    ▷ select a set of k landmarks
3:     for all v ∈ V do
4:         pred ← ⊥
5:         dist(s,v) ← ∞                                ▷ Except for dist(s,s) = 0
6:     end for
7:     openset ← s
8:     while openset not empty do
9:         c ← v ∈ openset with lowest dist(s, v) + π^L(v)
10:        if c == t then
11:            return reconstruct_path(pred, c)
12:        end if
13:        openset.remove(c)
14:        for all w ∈ V with (c, w) ∈ E do
15:            if dist(s, c) + ω(c, w) + π^L(w) < dist(s, w) + π^L(w) then
16:                pred(w) ← c
17:                dist(s,w) ← dist(s, c) + ω(c, w)
18:                openset.add(w)
19:            end if
20:        end for
21:    end while
22: end function
```

Besides, for any landmark, we know that

- $d(v,t) \geq d(\lambda,t) - d(\lambda,v)$
- $d(v,t) \geq d(v,\lambda) - d(t,\lambda)$

according to the triangle inequality. However, we need to precompute all distances from landmarks to nodes and from nodes to landmarks. Moreover, we know that the tightest the lower bound of $d(v,t)$ is, the better the performance of A* since the best lower bound possible is actually equal to the actual $d(v,t)$. Let us consider two feasible potential functions $\pi_1$ and $\pi_2$ such that $\pi_1$ dominates $\pi_2$, that is $\pi_1(v) \geq \pi_2(v)$ for any $v$. We have the theorem from [38] :

**Theorem 1.** *The set of vertices scanned by A* search using $\pi_1$ is contained in the set of vertices scanned by A* search using $\pi_2$.*

By this theorem, we can notice that any algorithm using a lower bound with non-negative potential function will systematically visit fewer vertices than Dijkstra's algorithm since it is equivalent to running Dijkstra with a potential function equal 0. However, [44] showed that this theorem is not true if there are ties in the algorithm. That is, when selecting the next node according to the potential function, two nodes have the same cost. The way we break this tie can impact the search space of the algorithm and sometimes the version of the algorithm using a dominating potential function can produce a larger search space in terms of the number of scanned vertices.

Thus, in the case of A* with landmarks, we want to maximize

$$d(v,t) \geq \max(d(\lambda,t) - d(\lambda,v), d(v,\lambda) - d(t,\lambda))$$

### Landmarks selection

Given theorem 1 and the previous remark, we can intuitively conclude that a good landmark is situated before node $v$ or after node $t$.

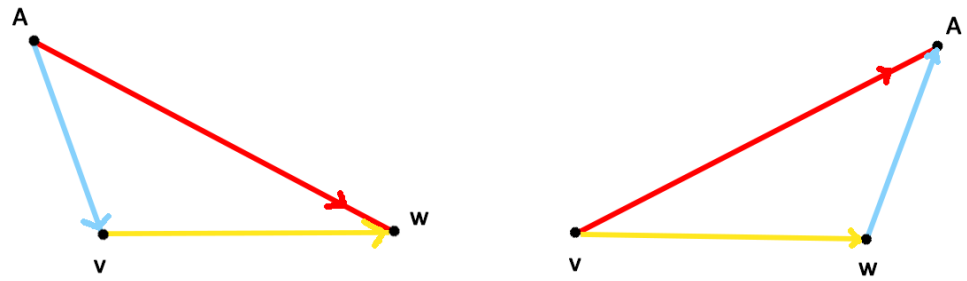



Figure 4.12: Landmark before and after nodes

Indeed, The longer the length of $d(\lambda,t)$, the bigger the lower bound on $d(v,t)$, and thus the better the performance of A* with landmarks. Of course, in order to maximize the efficiency, we have to speed up every $(s,t)$ query. Therefore, we have to choose landmarks that are located before any node or after any node. To do that, the easiest way is to choose landmarks on the border of the graph.

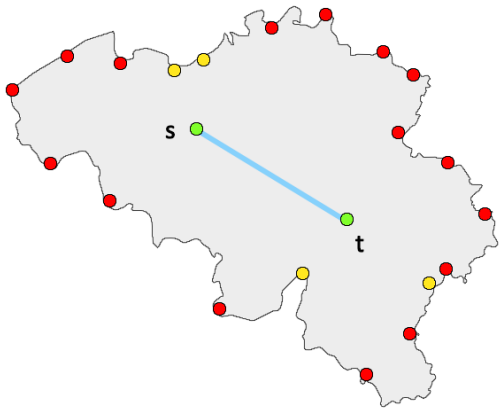



Figure 4.13: Example of landmarks on the border of Belgium map

The difficulty is actually to select these landmarks in the preprocessing phase of the algorithm. According to [45] and [44], the problem of finding $k$ different landmarks so that the search space for random point-to-point queries, is NP-hard. But there already exists some heuristics [44]. Among them, we have (from [38])

- *Random* : Simply select $k$ landmarks randomly

- *Farthest* : Choose a start vertex and succesively take a vertex that is farthest away from it, iteratively until having $k$ landmarks. This method works for any kind of graphs
- *Planar* : Fix a central vertex $c$, then divide the graph into $k$ pieces cented in $c$. For each piece, we select the farthest vertex with respect to the center $c$.

### 4.3.6 Arc-Flags

This speed-up technique **(Ulrich Lauther, 2004, [46] [47])** is based on a *pruning strategy*. The idea is to reduce the search space of Dijkstra's algorithm by analysing the importance of each edge in the graph. By associating additional data to each edge of the graph, we can check whether an edge belongs to the shortest path to a target node or not. Hence, this method requires a preprocessing phase. Concretely, it consists of 2 steps :

- Partition the graph into cells. Formally, given a graph $G = (V, E)$, we split $V$ in $r$ cells so that $V = \mathcal{C}_1 \cup \ldots \cup \mathcal{C}_i \cup \ldots \mathcal{C}_r$. As it is mentioned in [48], the partitioning is hard and can greatly impact the preprocessing phase as well as the query phase.
- Associate a label composed of a set of flags to each edge. For each cell, there is a flag associated with each edge in order to tell whether this edge belongs to a shortest path reaching this cell. Formally, for each $e \in E$ and for each $\mathcal{C}_i$, we associate a label $\mathcal{F}_1 \ldots \mathcal{F}_i \ldots \mathcal{F}_r$ where $\mathcal{F}_i = 0, 1$ is the binary flag telling if there exists at least one shortest path $P$ from any source $s \in V$ to a destination $t \in \mathcal{C}_i$ so that $e \in P$. Similarly to ALT preprocessing, we can compute all shortest paths from all nodes $u \in \mathcal{C}_i$ using a single-source version of Dijkstra's algorithm in $\overleftarrow{G}$ in order to check if $e$ is part of at least one of them. However, in practice, optimizations are needed.

Given these labels, Arc-Flags-Dijkstra queries remain the same as Dijkstra's queries except for the relaxing phase. Given a pair of nodes $s - t$, only edges with $\mathcal{F}_i = 1$ for $t \in \mathcal{C}_i$ are relaxed.

However, on long-range queries, simple unidirectional Arc-Flags-Dijkstra becomes inefficient. Indeed, no speed-up is achieved for queries that occur in the same cell $\mathcal{C}_i$, while the preprocessing computation time is high. Hence, during the search in a unidirectional query, more and more edges become important, leading to a "*coning effect*" (as shown in [46]) when approaching the target cell. Then, when we are in the cell of the destination node, all edges are considered as important and thus relaxed by the algorithm. To overcome this issue, we can use the bidirectional version of Arc-Flags, allowing the backward search to avoid getting trapped in the cone effect.

## 4.4 Hierarchical search speed-up techniques

### 4.4.1 Reach-based routing

Reach **(R. Gutman, 2004, [49])** is a very fast speedup technique. The idea is to take into account the "reach" of a node in a given graph. Considering pairs of points in a graph, we can determine if a node is likely to belong to the shortest path between the two points by looking at the minimum distance between the node and the two points. Concretely, given a $s-t$ path $P$ and a node $v \in P$, we associate an absolute reach value $\bar{\mathcal{R}}(v) := \max_{s,t \in V} \mathcal{R}_{s,t}(v)$ where $\mathcal{R}_{s,t}(v)$ is the reach with respect to the $s - t$ path $P$ and $\mathcal{R}_{s,t}(v) = \min(d(s, v), d(v, t))$. The absolute is the maximum reach on all shortest paths. Intuitively,

- Small $\bar{\mathcal{R}}(v)$ : $v$ can belong to the shortest path only if it is close enough to the source vertex of destination vertex.
- Big $\bar{\mathcal{R}}(v)$ : $v$ can belong to the shortest path only even if it is far away (e.g. motorways).

Reach algorithm will first compute all $\bar{\mathcal{R}}(v)$ in the **preprocessing** phase. However, simply computing all pairs of shortest paths is prohibitive for huge graphs. Thus, we need to compute upper bounds on the reaches. To obtain those upper bounds, Gutman [49] proposes a method thah consists in iteratively performing local Dijkstra queries on small regions, then continuing on a subgraph composed of unvisited nodes until the number of nodes without associated reach values is small enough.

During the query phase, given a source node $s$ and a destination node $t$, a node $v$ is pruned if it satisfies the two following conditions :

$$d(s,v) > \bar{\mathcal{R}}(v) \text{ and } d(v,t) > \bar{\mathcal{R}}(v)$$

Indeed, $v$ can be pruned since it cannot be part of the shortest path from $s$ to $t$. Assume the two properties are satisfied, it would mean that $\mathcal{R}_{s,t}(v) > \mathcal{R}(v)$ since $\mathcal{R}_{s,t}(v) = \min(d(s,v), d(v,t))$ and thus, $\bar{\mathcal{R}}(v) > \mathcal{R}_{s,t}(v)$ by definition, which is a contradiction.

During the query, we need $d(s,v)$, $d(v,t)$ and $r(v)$ when encountering node $v$. However, $d(v,t)$ is unknown when processing $v$. Thus, we need to compute the lower bound on the distance to the destination vertex. We can use a Bidirectional search so that the backward search distance from $t$ is the lower bound on the forward search.

### 4.4.2 Highway Hierarchies

Highway Hierarchies **(P. Sanders and D. Schultes, 2005, [50] [51])** exploits the hierarchy of the graph. In road networks, there are different kinds of roads and its inherent hierarchy implies that some roads have more importance than others. For long-distance travel, it is better to go through highways in order to minimize the travel time. Usually, a trip is constituted of a mix of small ordinary roads and highways. However, the notion of hierarchy in a road network is only perceived by humans. To translate this idea into a graph, we have to create a hierarchy within the nodes. Using some heuristics, we can assess the importance of each node.

Highway Hierarchies (HHs) [50] [51] consists of a preprocessing phase and a query phase.

#### Preprocessing phase

HHs assigns a hierarchy of levels to nodes and edges of the graph by applying a nodes removal procedure. *Highway* edges are identified as important. This identification is done by local Dijkstra's algorithm executions. We remove unimportant nodes from the graph (important if the network is very sparse). In particular, nodes of small degrees (non-highway edges) are removed. Concretely, each node $v$ is labeled with a neighbourhood radius $r(v)$. An edge $(u,v)$ is a highway edge if it belongs to some shortest path $P$ from a source node $s$ to a destination node $t$ such that $(u,v)$ is neither fully contained in the neighborhood of $s$ nor in the neighborhood of $t$. Formally, $d(s,v) > r(s)$ and $d(u,t) > r(t)$.

It results in a *highway hierarchy* graph composed of L levels (a natural hierarchy of the

network). A low level corresponds to unimportant edges while a high level corresponds to very important edges (e.g. motorways).

**Query phase**

The query phase is a bidirectional Dijkstra search with the difference that edges do not need to be expanded when the search is sufficiently far away from the source or target nodes. Each node is associated with a search level $\ell$ and a *gap* value to the next applicable neighborhood border. During the bidirectional search, there are two restrictions to comply with.

- Both forward and backward searches can only go upward. That is, given an edge, only edges of higher level are considered for expansion and the rest are ignored.
- As long as we stay inside a given neighbourhood, no constraint is applied. However, whenever an edge $(u, v)$ crosses its neighbourhood border (which means that the length of $(u, v)$ is greater than the gap), we switch to a higher level $\ell$. When a level $\ell$ is accessed through a node $u$, the *gap* becomes its neighbourhood radius. Each time the edge $(u, v)$ is relaxed, we apply $gap(u) - \mathcal{W}(u, v)$, where $\mathcal{W}$ is the edge weight. We can only switch to a higher level $\ell$ if the gap becomes negative.

This method offers the advantage of being independent of route information and also offers low preprocessing and query times. As stated in [52], it only takes 15 minutes to preprocess the Western European road network and 0.5ms to process a query.

### 4.4.3 Highway Node Routing

Highway Node Routing **(P. Sanders and D. Schultes, 2007, [53])** is based on the same idea of hierarchy as Highway Hierarchies. However, the construction of the hierarchy in the preprocessing is different. It is done in two phases. In the first phase, nodes are classified into different levels. Then, in the second phase, shortcuts are added to the graph. This phase can be quickly performed and offers easy updates if edge weights change.

The query is a bidirectional search with the only constraint that edges must only move to higher levels and avoid suboptimal branches of the search space on lower levels of the hierarchy.

The performance of Highway-Node Routing is similar to that of Highway Hierarchies even though the former has the advantage of efficiently updating the preprocessing (shortcuts) and efficiently using memory (low memory consumption). By using only important nodes for higher levels, the query performance is also similar to Highway Hierarchies performance.

### 4.4.4 Transit Node Routing

Transit Node Routing **(P. Sanders and D. Schultes, 2007, [54] [55])**, is a generic framework that allows to find shortest path costs in road networks (not the actual shortest path). The main idea is to precompute distances between some pairs of nodes in a preprocessing phase, and then use those precomputed distances during the query phase. The native implementation would be computing all shortest path pairs in the network and storing them in a distance table of $V \times V$. In practice, it is not suitable for big road networks in terms of preprocessing time and memory consumption. Formally, the framework consists of 4 steps (as presented in [56] and D. Schultes PhD thesis [57]) :

- A set $\mathcal{T} \subseteq V$ of *transit nodes.*

- A *distance table* $\mathcal{D}_{\mathcal{T}} : \mathcal{T} \times \mathcal{T} \rightarrow \mathbb{R}_0^+$ of shortest path distances between all pairs of transit nodes is created.
- A forward (backward) *access node* mapping $A^{\uparrow} : V \rightarrow 2^{\mathcal{T}}$ ($A^{\downarrow} : V \rightarrow 2^{\mathcal{T}}$ ). For any shortest $s-t$ path $P$ containing transit nodes, $A^{\uparrow}(s)$ ($A^{\downarrow}(t)$) must contain the first (last) transit node on $P$. Thus, for each node $v \in V$, a set of access nodes is determined. An access node is the first transit node in a shortest path from $v$. Then, for each node $v$ and each access nodes $a^{\uparrow}$ in $A^{\uparrow}$ ($a^{\downarrow}$ in $A^{\downarrow}$), the distance $d(v, a^{\uparrow})$ ($d(v, a^{\downarrow})$) is computed.
- A *locality filter* $\mathcal{LF} : V \times V \rightarrow true, false$. $\mathcal{LF}(s,t)$ is set to true when there is no transit node in the path between $s$ and $t$. However, it is possible to have false positives, meaning that $\mathcal{LF}(s,t)$ may contain be true even when a shortest path contains a transit node. Such locality filter can be implemented using geometric disks with the property that the disks of source and target nodes overlap if they are too close (as mentioned in [57]).

During the query phase, we first check the locality filter so determine if there exist a transit node in the considered shortest path. If $\mathcal{LF} = true$, then it means that there is no transit node and some fallback algorithm is applied to handle the local query. The fallback algorithm can be another speedup technique. Otherwise, we apply the following formula :

$$\mu(s,t) = \mu_{min}(s,t) := \min_{a_s \in A^{\uparrow}, a_t \in A^{\downarrow}} \{d_{A^{\uparrow}}(s, a_s) + \mathcal{D}_{\mathcal{T}}(a_s, a_t) + d_{A^{\downarrow}}(a_t, t)\}$$

where $\mu(s,t)$ is the shortest path between $s$ and $t$. The advantage is that this method requires a small number of table lookups.

### 4.4.5 Contraction Hierarchies

#### Hierarchy

In section 4.2, we have seen that the bidirectional version of Dijkstra's algorithm can provide a speedup of 2x for road networks. We now overview a speedup technique that provides a speedup of thousand. The idea behind this technique is based on the notion of road hierarchy as for Highway Hierarchies and Highway Node Routing (as described in subsections 4.4.2 and 4.4.3.

*Contraction hierarchies* (CH, **Geisberger et al., 2008, [58]**) [33] [29] [59] is a well studied speed-up technique that is an extreme case of the hierarchies in highway node routing [53] (its node ordering is based on Highway Hierarchies) because it defines its proper level of hierarchy. Besides, CH is conceptually simpler.

Contraction hierarchies algorithm works as follows: In a **pre-processing phase**, it creates shortcuts in the graph which enable the algorithm to skip unimportant vertices. Instead of considering full paths, the algorithm will only take one edge into consideration, which significantly reduces the queries time. In the **query phase**, in order to process different shortest path queries, another algorithm (bidirectional search) is used, taking advantage of those shortcuts.

### Preprocessing

The main goal of the preprocessing phase [55] is to transform the original graph into an *augmented graph* by successively eliminating nodes base of their order and adding shortcuts between some nodes. When a node is eliminated, it could be that in the original graph, it had in-going and out-going edges. In this case, we have to add a shortcut between its neighbours with an appropriate length. Removing a vertex is called a **contraction**. Let us define the general contraction process. At each step of the preprocessing, a node is contracted, which means that we remove it from the original graph and put it in the augmented graph. Let us consider this small graph :

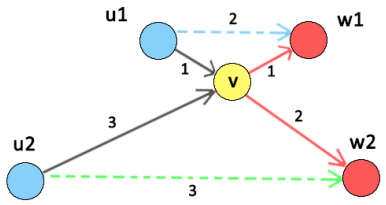



Figure 4.14: Node contraction - 2 cases

After contracting node $v$, we have to add shortcuts [55] in order to preserve the link between the remaining nodes (in figure 4.14, between $u_1$ and $w_1$).

**Definition 6.** *A witness path $P_{uw}^W$ from node $u$ to $w$ is a shortest path from $u$ to $w$ not containing node $v$.*

For every pair of edges $(u, v)$, $(v, w)$ that were originally adjacent to $v$, we have two possible situations :

1. If there exists some witness path $P_{uw}^W$ from $u$ and $w$ that is shorter than $\omega(u, v)+\omega(v, w)$ and that does not pass through $v$, then we do not add any shorcut between $u$ and $w$ (in figure 4.14, witness path from ($u_1$ to $w_1$)).
2. If there is no witness path, then we add a new shorcut edge $(u, w)$ with $\omega(u, w) = \omega(u, v) + \omega(v, w)$ (in figure 4.14, shortcut $(u_1, w_1)$).

To determine if there exists some witness path when contracting node $v$, we have the witness search. [58]

**Definition 7.** *A witness search is a search for a witness path, that is determining if the shortest path between $u$ and $w$ contains $v$.*

The resulting augmented graph has the same number of nodes as the original graph and the same edges as well. Besides, it also has additional edges that are shortcuts. The augmented graph, denoted $G^+$ is defined as follows :

**Definition 8.** *The augmented graph of graph $G$, $G^+ = (V, E^+)$ is the graph containing the set same of vertices $V$ as $G$ and an augmented set of edges $E^+$, that is a combination of all initial edges $E$ of $G$ and the shortcuts added in the preprocessing phase of contraction hierarchies algorithm.*

Let us consider an example on a simple graph that does not contain any witness paths :

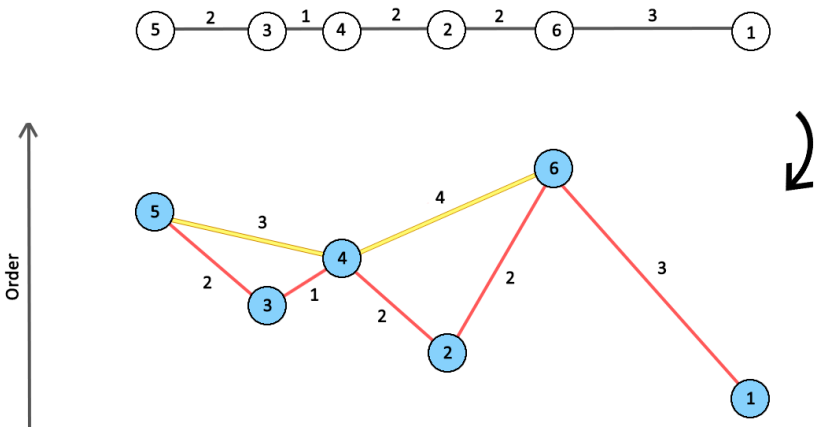



Figure 4.15: Preprocessing phase : node contractions

Nodes were contracted according to their ordering (here their respective numbers). 2 shortcuts are added (between 5 and 4 and between 4 and 6). Node 6 is not contracted since it is the last node to be considered.

### Witness Search

Witness search is important. If we do not have this search, the preprocessing will add shortcuts between every predecessors and successors of contracted nodes, leading to a tremendous amount of added edges in the augmented graph $G+$. Finding the shortest paths in this resulting graph then becomes harder.

In order to find witness paths, we have to run a shortest path algorithm between two nodes.

**Definition 9.** *Given a node $v$, its predecessor is a node $u$ so that there exists $(u, v)$ and its successor is a node $w$ so that there exists $(v, w)$.*

Given node $v$, We can run Dijkstra's algorithm for each of its predecessor $u$, from $u$ without considering $v$. However, it can rapidly become inefficient. Therefore, we can optimise this witness search with some conditions:

**Stopping Criterion** Stop the search when the distance from a predecessor is already bigger than the maximum distance from the predecessor and the successor of the node we consider. If $d(u_i, x) > \max_{u,w}(\omega(u, v) + \omega(v, w))$, then there is no witness path going through node $x$ to some successor of $v$. We limit the search distance to $\max_{u,w}(\omega(u, v) + \omega(v, w))$.

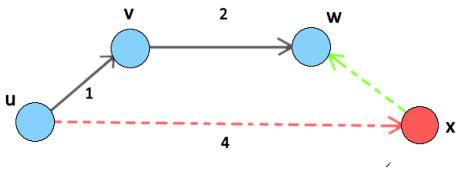



Figure 4.16: Witness search stopping condition - case 1

We can have another condition. Let us assume we have the predecessor $w'$ of the successor $w$ of $v$. If $d(u, w') + \omega(w', w) \leq \omega(u, v) + \omega(v, w)$, it means that we have a witness path from $u$ to $w$. In this case, we want to limit the distance to $\max_{u,w} \max_{w',w}(\omega(u, v) + \omega(v, w) - \omega(w', w))$, which is the difference between the sum of the two longest incoming and outgoing edges and the last edge weight on the path.

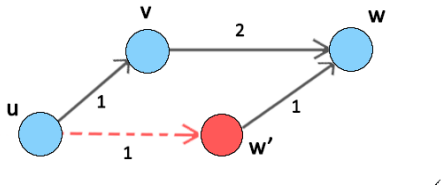



Figure 4.17: Witness Search stopping condition - case 2

**Hops limit** Limit the search to a number of hops, meaning that we only consider shortest paths from the source node with at most $k$ edges. In this case, it is possible that no witness path is found during the process. If there is no witness path, we add a shortcut. This leads to a tradeoff between preprocessing time and the number of edges in the resulting augmented graph $G+$. $k$ can be set to 1 at the beginning and then increase gradually over time.

### Node importance

A good node ordering allows us to minimize the number of shortcuts added in the augmented graph. The problem of finding an optimal node ordering is NP-Complete. [60]

There exists some heuristics we can use in order to determine a node ordering.

- *Bottom-up* : The order is not known in advance and the heuristics work with a greedy principle, meaning that it orders nodes based on the information about previous node contractions and neighbouring nodes. This method is cheaper in terms of preprocessing time
- *Top-down* [61] : The ordering is done for all nodes before contracting any nodes. This method is expensive in terms of preprocessing time.

We now define some existing bottom-up heuristics [59] that when combined, gives an importance value for each nodes. let us define $I(v) = ed(v) + cn(v) + sc(v) + nl(v)$, the importance of node $v$, which is a combination of 4 heuristics described hereafter.

**Number of shortcuts** We want to minimize the number of added shortcuts in the augmented graph $G+$. We can keep track of the number of shortcuts added $s(v)$ and define the edge difference as $ed(v) = s(v) - inc(v) - out(v)$, where $inc(v)$ is the incoming degree of $v$ (number of predecessors) and $out(v)$ is the outcoming degree of $v$. Those values represent the deleted edges after node contraction. When a node $v$ is contracted, the number of edges increases by $ed(v)$. We want to contract nodes with small $ed(v)$ value.

**Number of contracted neighbours** In road networks, important nodes are usually spread uniformly on the map. In order to avoid clusters of contractions in some regions of the graph,

we can simply keep track of already contracted neighbours $cn(v)$ and contract node with small $cn(v)$.

**Shortcut cover** Some nodes are more important than others in the sense that they cannot be avoidable. We want to contract those nodes later. Let us define the *shortcut cover* $sc(v)$, the number of nodes covered by the contraction of a node $v$. That is, the number of neighbours $w$ of $v$ such that the added shortcuts are from or to $w$. The bigger the shortcut cover, the bigger the importance of node $v$. Thus, we want to contract nodes with small $sc(v)$.

**Node level** In the augmented graph, we can define $nl(v)$, the upper bound on the number of edges contained in the shortest path from any pair $s$ to $v$. Initially, $nl(v) = 0$ since there is no path from any vertex to $v$ in the augmented path. Then, after contracting $v$, we must update its neighbours's level. For some neighbour $u$ of $v$,

- Either we can reach $u$ without going through $v$ and its level remains the same
- or we can reach $u$ by going through $v$ and its level becomes $L(v) + 1$ since from $s$ to $v$, it is nl(v) level, and one more for the edge from $v$ to $u$.

Thus, for each neighbour $u$ of $v$, we update $nl(u) = max(nl(u), nl(v) + 1)$. We want to contract node with small $nl(v)$ level.

**Algorithm** Given the combination of heuristics $I(v)$, we can redefine the preprocessing phase of contraction hierarchies algorithm. All nodes will be added to a decreasing priority queue based on their importance. At each iteration, the least important node is extracted. When a node is extracted, we have to recompute its importance because it could have changed due to the other nodes contractions. We want to contract this extracted node, but first, we need to make sure its new importance value is indeed the minimum among all nodes in the queue. To check that, we can simply compare its importance with that of the priority queue's head importance even though the latter is not updated yet. If the extracted is not the minimum, then it is put back in the priority queue and it is not contracted. Since an extracted node is updated anyway, the algorithm will pick a valid node at some point since in the worst case, all nodes in the queue will be updated accordingly. Thus, eventually, the algorithm will terminate.

### Shortest path queries

During the query phase, we want to find the shortest path from two nodes on the augmented graph resulted from the preprocessing. One can run a Bidirectional search (described in section 4.2) that will profit from the shortcuts introduced in the graph as well as the node ordering. Since nodes are ordered according to their importance, we can first start each search with the least important nodes and increasingly considering more important nodes.

**Definition 10.** *The rank $r(v)$ of node $v$ is its position in the node order defined by the preprocessing phase.*

Geisberger et al [58] proved that given some pair of vertices $s$ and $t$, there exists a node $u^*$ which is has the highest rank $r(v)$ such that $d_s(u^*) = dist(s, u^*)$ and $d_t(u^*) = dist(u^*, t)$.

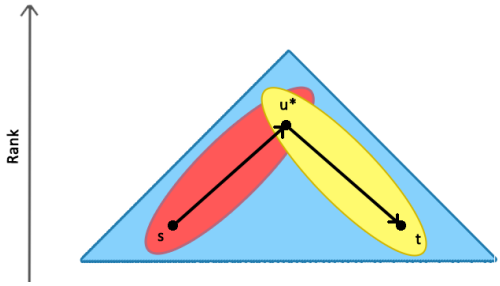



Figure 4.18: Bidirectional search for shortest path query

However, $u^*$ is not systematically the first node that is encountered by forward and backward search in the bidirectional search. Thus, the algorithm cannot stops always whenever the two distinct searches meet. Instead, we must keep track of an estimate of the shortest path in both searches and stop whenever a new processed node gives a distance greater than the estimated distance to the target.

**Definition 11.** *A path $P = (v_1, v_2, \ldots, v_k)$ in the augmented graph $G^+$ is called* ***increasing*** *if ranks $r(v_1) < r(v_2) < \ldots < r(v_k)$. $P$ is called* ***decreasing*** *if ranks $r(v_1) > r(v_2) > \ldots > r(v_k)$.*

Given two nodes $A$ and $B$, the edge adjacent to them is upward if $B$ has been contracted after $A$ in the preprocessing phase ($B$ is more important than $A$) and vice-versa for backward edge. To implement this bidirectional search, we can use a upward graph $G_\uparrow$ containing upward edges and a backward graph $G_\downarrow$ containing downward edges. In the end, forward search finds increasing shortest path from $s$ to $u^*$ and backward search finds increasing shortest path from $t$ to $u^*$. The final shortest path is thus the increasing path from $s$ to $u^*$ plus the decreasing path from $u^*$ to $t$. Algorithm 9 is modified and becomes :

**Algorithm 13** Query phase of Contraction hierarchies algorithm

```
1:  function ContractionHierarchiesQuery(G, s, t)
2:      μ ← +∞                                              ▷ estiamte distance to t
3:      Fill dist^G and dist^R with +∞ for all v ∈ G.V
4:      dist^G[s] ← 0, dist^R[t] ← 0
5:      visited^G ← ∅, visited^R ← ∅
6:      while No nodes need to be processed do
7:          c^G ← Extract_minimum(dist^G)                   ▷ using priority queue
8:          if dist^G[c^G] ≤ μ then
9:              Process(c^G ...)                            ▷ see algorithm 10
10:         end if
11:         if c^G ∈ visited^G and dist^G[c^G] + dist^R[c^G] < μ  then
12:             μ ← dist^G[c^G] + dist^R[c^G]
13:         end if
14:         c^R ← Extract_minimum(dist^R)
15:         repeat symmetrically for c^R as for c^G
16:     end while
17:     return μ
18: end function
```

Nevertheless, note that the query as described hereabove yields an estimated distance and not the actual shortest path from $s$ to $t$. In order to obtain such a path, we have to take into account nodes from the original graph. Indeed, the search might have processed shortcuts in the augmented graph and those shortcuts were originally the combination of two different edges.

**Algorithm correctness**

We show that algorithm 13 is correct in Appendix B.2

## 4.5 Advanced techniques and Performance

All the speedup techniques that we overviewed are efficient on their own, but it is possible to achieve way faster speedups using combinations of different techniques. Bidirectional search can be combined with almost all other techniques as it only requires creating a reverse graph. We have seen that it is a required ingredient of highway hierarchies (4.4.2), transit node routing (4.4.4), highway-node routing (4.4.3) and contraction hierarchies (4.4.5). It can also considerably improve reach-based routing and arc-flags.

ALT has great synergy with reach-based routing as it will be described in subsection 4.5.2 whilst it only gives a small speedup when combined with Highway Hierarchies [62]. The combination of goal-oriented search with a hierarchical approach is also very successful (e.g. Arc-Flags (also called *"Edge flags"*) with shortcuts as it will be described hereunder in 4.5.1) Combining Arc-Flags with contraction hierarchies yields large query times improvement. Currently, the fastest combination (in road networks) is composed of Arc-Flags and Transit-node routing. We will look at some performance comparisons in subsection 4.5.3.

### 4.5.1 SHARC

SHARC **(R. Bauer and D. Delling, 2010, [63])** is an improved version of Arc-Flags (**Sh**orcuts + **Ar**c-Flags) which can be seen as a combination of a goal-directed search and a hierarchical methodology. It benefits from the notion of "*contraction*". The idea is to iteratively remove nodes that are not important for the search and add adequate shortcuts in order to keep the correctness of the distances between the remaining nodes. This notion has already been described in detail in section 4.4.5. It is done during the preprocessing phase and significantly reduces its computation time. This method yields fast unidirectional queries and offers the great advantage of being goal-directed and unidirectional, which means that it can be applied to time-dependent graphs (bidirectional search is not suitable).

### 4.5.2 REAL

REAL **(Goldberg et al, 2006, [64])** is the combination of **Re**ach and **AL**T. The idea is to run ALT algorithm and prune nodes based on reach conditions. It works because ALT transforms edge lengths but the shortest paths remain invariant. REAL has two independent preprocessing algorithms: one to compute shortcuts and reaches, and the other to choose landmarks and compute distances from all vertices to them. The query takes as input the graph with shortcuts, reach values and ALT landmarks distances. REAL uses the heuristic costs of ALT in order to have tighter distances lower bounds, which leads to more node pruning.

### 4.5.3 Performance

All speed-up techniques offer significant improvement over plain Dijkstra's algorithm. However, those speedups depend on the preprocessing computation time (if any), query time, and memory consumption. We take the table from [65] to have an idea of those speedups :

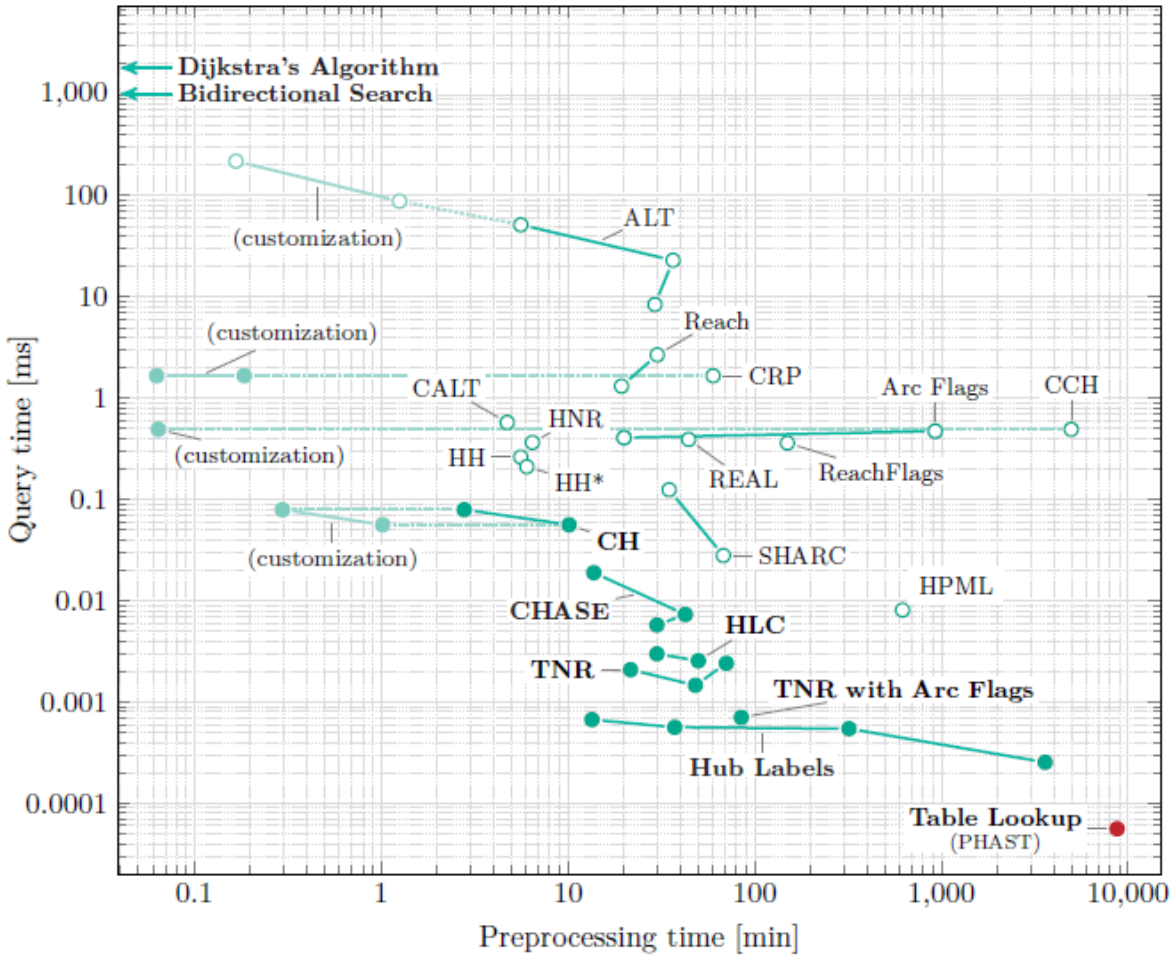



Figure 4.19: Preprocessing and average query time performance for speedup techniques on the road network of Western Europe, using travel times as edge weights. Illustration from [65] who took inspiration from [66]

# Chapter 5

# Intermodality and preferences

Intermodal routing is a very interesting and well-studied subject that attracts many researchers's interests (e.g we have [67], [68] and [69]). It consists in mixing several modes of transportation (= *modalities*) using a single data structure (single graph). The goal in our case is to apply the shortest path problem on multimodal road networks graphs. To meet this goal, we must establish some conditions for the modelling of such multimodal graphs :

## 5.1 Types of modalities

First, a multimodal (intermodal) network can be viewed from several aspects. From a *functional* point of view, it can be classified into 2 modes : **private** (foot, bike, car) and **public** (bus, train, tram, metro). According to [70], this approach is suitable for a multimodal network since private mode applies at any time and anywhere, whereas public mode requires time tables and physical nodes (stations). Here are a non-exhaustive list of some modes of transport that can be interesting to deal with in an intermodal graph. They are grouped based on their types :

### 5.1.1 Free-floating service

Free floating is a vehicle sharing service without a station. The principle is simple: it is a question of making means of transport available to the public everywhere in the city, without a tether terminal. Examples of such vehicules are bikes, scooters (Lime, Dott, ...), cars, etc.

### 5.1.2 Stations

**Bikes** The city contains bike stations and any user can take a bike in a station then leave it in another station. In Belgium, a good example is "Villo" stations.

**Cars** An user can take a car at a station, but he must return it to the same station afterwards. (examples : Cambio, ZenCar, ...)

### 5.1.3 Personal

A user can use his own vehicule and can thus leave it anywhere at anytime in order to use another mode of transportation.

### 5.1.4 Public transportation

Public transportation is a system of transport available for the public unlike private transport, managed on a schedule and operated on predefined routes separated by stops where users can embark or disembark.

## 5.2 Scheduling

Secondly, when applying the shortest path problem to multimodal graphs, we must consider the relevance of the scheduling of modalities. That is, with certain combinations of modalities, ordering problems can arise. For example, consider a bimodal graph with a free-floating car layer and a station-based bike layer. If a user chooses to take the car during a certain amount of time, then switch to a bike, he will not be able to use his car again later, which means that he will be forced to use a bike for the rest of the trip.

To avoid that problem, we can simply constraint the ordering of modalities by setting forward transition edges strategically. For instance, there can be only 1 single forward transition edge from the base layer $\mathcal{L}_0$ to a personal layer so that when a user ends up in this layer, he is forced to continue his trip with the same modality until the end.

## 5.3 Data structure

Thirdly, in order to implement an intermodal road network graph, we can organize it as a set of layers, which represent the different modes of transportation and that we denote as $\mathcal{L}$. This approach is adopted in [70] and [67] as well as in many more publications. To construct such graph, we start by setting up an initial base graph $G = (V, E)$ embedded in $\mathbb{R}$ (containing all the nodes and edges of a considered region). let us denote the base layer as $\mathcal{L}_0$. The modality of the base graph must allow a user to go through any node and any edge (e.g.: foot, free-floating car). The foot layer is the best candidate since it do not require any vehicule and the user can go anywhere without any constraint. Then, we duplicate the entire graph for each modality and add it to a separate layer. Each node belongs solely to one layer. In some cases, depending on the modalities (we will look at some examples later), we cannot simply duplicate the base graph. Instead, we must only add edges that are accessible by the considered mode of transport. Finally, we add **transitions** edges where needed. We must have 2 kinds of transition edges :

- **Forward** : An edge that we denote as $e^F = (u, v)$ that starts from the base layer $\mathcal{L}_0$ ($u \in \mathcal{L}_0$) and ends up in another layer $\mathcal{L}_i$ for any $i \in \mathcal{L}$ with $v \in \mathcal{L}_i$
- **Backward** : An edge that we denote as $e^B = (u, v)$ that starts from a layer $\mathcal{L}_i$ for any $i \in \mathcal{L}$ ($u \in \mathcal{L}_i$) and ends up in the base layer $\mathcal{L}_0$ with $v \in \mathcal{L}_0$

Those edges will attach to specific nodes depending on the modalities of the layers. Now, we will illustrate this concept with simple bi-modal graphs. For a more detailed example of multimodal graph with more than 2 modalities, we can look at [5] or [67].

### 5.3.1 Example 1 : Walk + personal car

To construct such graph, we simply create a new layer and duplicate the base layer. Indeed, the topology of both layers can be identical since we can usually use a personal mode of

transport on the same roads as by foot. Then, we must add a single forward transition edge in order to ensure the scheduling of modalities is relevant (as explained in section 5.2). We can associate a cost to this edge. For example, we can set the time needed to switch to the other modality.

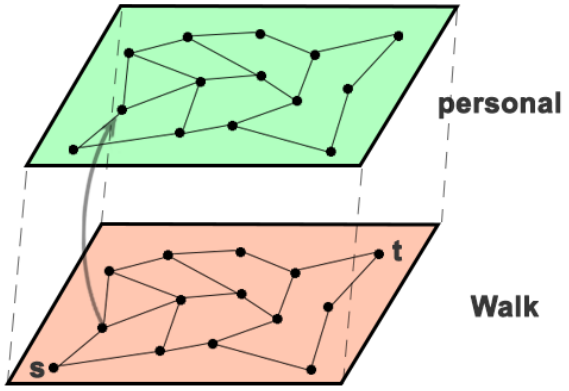



Figure 5.1: Walk layer + personal car layer

### 5.3.2 Example 2 : Walk + Station-based bike

This combination of modalities less trivial than the previous one. We create a layer for the station-base bike mode of transport and we add a forward and a backward transition edge to each station. The edge cost could be the service price.

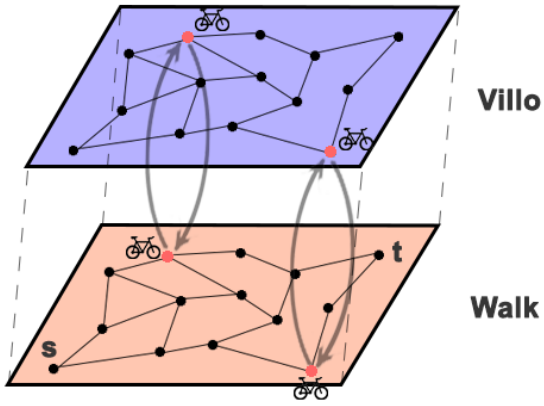



Figure 5.2: Walk layer + bike-stations layer

### 5.3.3 Example 3 : Walk + public transport (bus)

We must create one layer per public transport line. Here, we only consider one bus line. Also, for simplicity, we consider that a line is simply a straight line between two selected nodes (public transport stations), but ideally we should have considered the same roads and interesections as the base layer to match reality. As for the station-based modality, we add a forward and a backward transition edge to each station. The edge cost could be the combination of the time needed to embark and the service price. Ideally, we can adopt a time dependent approach where time table events are modeled as link cost function.

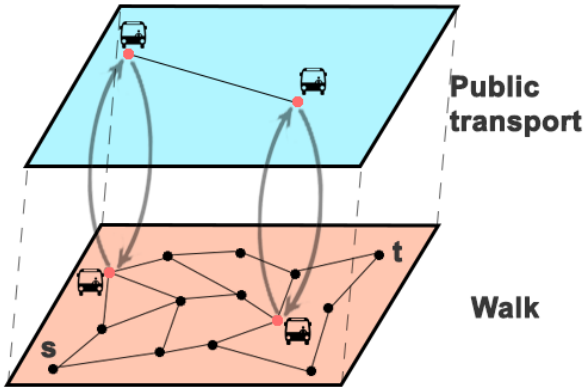



Figure 5.3: Walk layer + public transport layer

## 5.4 User preferences

Fourthly, in order to model the multimodal transport network to make it meet a multi-criteria model, we must ensure that each edge is associated with a generalized cost measure that combines several relevant metrics. Indeed, there is a motivation for turning a multimodal transport network into a user-adapted configuration, and thus a multi-criteria problem. In a uni-modal configuration, there is no preferences involved since the user can only use one mode of transport. Contrariwise, in a multimodal configuration, the user can assign a certain degree of importance to each modality. Indeed, passengers may want to exclude some transportation modes (e.g. foot or bike when the wheater is not adequate) or prefer to pay more over time by taking the bike instead of the car for instance. Furthermore, they may want to limit the number of changes between modalities.

## 5.5 Model

In order to model user preferences in multimodal graphs, we have 2 possible approaches :

### 5.5.1 Scalarized Model

- Weighted Sum :

$$\hat{\omega}(e) = \sum_{i=1}^{\mathcal{M}} c_i \omega_i(e)$$

  where $\mathcal{M}$ is the number of metrics, $c_i$ is the weight coefficient modelling the user's preference and $\omega_i$ is the value/weight of the metric $i$ for edge $e$. We associate to each edge $e \in E$ its weighted sum $\hat{\omega}(e)$

- Weighted Chebycheff [71] : the augmented weighted tchebycheff norm is another form scalarized model for multicriteria optimization: $\max_{i=1,\ldots,\mathcal{M}} (c_i|\omega_i - y_i|)$. We will not enter in details nor use this approach in our experiments.

### 5.5.2 Multi-criteria Model

The shortest path problem becomes a multiobjective combinatorial optimization problem. It has been well-studied in the scientific field (see for example [72] [73] [74]) We have a finite set

$X$ of feasible solutions. The goal is to minimize the multiobjective function :

$$min_{x \in X} f(x) = (f_1(x), \ldots, f_{\mathcal{M}}(x))$$

where $f$ maps $x \in X$ to $\mathbb{R}^{\mathcal{M}}$ and $\mathcal{M}$ is the number of objectives. At the end, we obtain the set of shortest paths that are said to be pareto-optimal regarding all objectives. Given a set of feasible solutions $X$, a solution $x \in X$ is Pareto optimal iff

$$\nexists x' \in X s.t. f(x') \leq f(x)$$

$$\iff f(x') \neq f(x) \text{ and } f_i(x') \leq f_i(x), i = 1, \ldots, \mathcal{M}$$

which means that for all metrics, there exists no other solution with at least one criterion performing strictly better, and all other solutions performing at least as good.

## 5.6 Metrics

If we consider time, there are 3 types of edge costs :

- Time **independent** : static costs
- Time **dependent** : varying through time based on past events (history)
- **Stochastic** time dependent : varyting through time based on past events and real-time events

We will only consider time *independent edge costs* (= **static**) for the experiments we will conduct later. However, public transport networks are always time-dependent due to its nature and possibly schocastic time dependent. In our case, we will not model the time dependency for the transition edges nor for the public transport lines.

Considering static time independent edge weights, we can have the following relevant metrics : *Time*, *price* and *ecological impact*. In case of foot and bike modes of transport, only the time is relevant.

- **Time**: This can easily be measured using the geographical distance between 2 nodes's coordinates (using haversine formula that will be described later) and the mode of transport speed.
- **Price**: It can also be easily measured. For a car, we take the average fuel price and the average gas consumption. Then, given 2 coordinates, we can get the distance between the 2 and thus compute the gas price for that distance. Of course, there are different types of fuels (gasoline, diesel, ..) and different types of cars (different gas consumptions).
- **Ecological impact**: It is more difficult to quantify. One possibility would be to compute the amount of damaging elements rejected by a heat engine (in case of a car using a thermal motor). Here is a non-exhaustive list of them:
  - carbon dioxide (CO2), a greenhouse gas
  - carbon monoxide (CO), a colorless, odorless and poisonous gas. It is produced during incomplete combustions.

- monocyclic aromatic hydrocarbons (HAM)
- nitrogen oxides: nitrogen monoxide (NO) and nitrogen dioxide (NO2). Nitrogen oxides form ozone by reacting under the action of UV.
- particles

Since the gas price is directly positively correlated with the ecological impact through the gas consumption, it can be relevant to only consider the price as a global metric. Indeed, the more expensive the trip is, the more pollution will be produced.

## 5.7 Applicability

Finally, we must select algorithms that are suitable with multimodal graphs in order to compute shortest paths. [67] applied several speedup techniques on multimodal networks successfully.

- With the multi-criteria model, there exists an algorithm called "**Label Correcting algorithm**" [75] [76] that is an adaptation of Dijkstra's algorithm to multi-modal multi-criteria route networks. It consists in assigning labels (path lengths) to each node and at each iteration, select a new node such that this node is not dominated by the others. We will not describe it in details nor apply this approach in our experiments although it could be interesting in order to determine how it performs against a speedup technique such as ALT.
- With the scalarized weighted sum model, general shortest path algorithms (Dijkstra, A*, ..) can be applied since they only require a non-negative weighted graph. The multimodal graph is simply an extension of a simple graph with more nodes and edges. However, we must ensure that the edge weights are non-negative, otherwise the algorithms will not work properly. Since $\omega$ is already defined as being non-negative, we must constraint the coefficients $c_i \geq 0, \forall i \in 1, \ldots, \mathcal{M}$. Logically, those weights should be computed on the fly during the query and the preference coefficients must be pre-defined before the query.

  Although general shortest path algorithms can easily be applied to multi-modal networks with weighted sums edges, it is difficult if we take speedup techniques into account. In particular, we focus on ALT that needs a preprocessing phase (as described in section 4.3. ALT is always applicable to such user-adapted networks as it relies on the triangle inequality property during the preprocessing phase, as long as the edge weights remain non-negative and the preference coefficients $c_i$ are bounded. It would imply to compute all distances to landmarks each time a user picks a different set of preference coefficients, which lead to tremendous running times. Moreover, new edge weights are no longer simple distances, thus the results of the algorithm may not be correct and that is what we will try to see in the experiments.

# Chapter 6

# Experiments and Discussion

Now that we have all the necessary background, we can conduct some experiments. We start by explaining how data were acquired and processed, then we present the different graphs used throughout the experiments. Following that, we establish design choices and assumptions and finally we perform the experiments and analyze them.

## 6.1 Data

### 6.1.1 Data aquisition

We get real world data of road networks of Belgium from the OpenStreetMap (OSM) database [1]. Additionally, we use Matsvei Tsishyn's javascript module "GraphFromOSM" (GFO) [2] in order to transform the data from OpenStreetMap to actual usable graphs. It allows us to perform OSM queries and obtain a Graph-structured data (directed graph) in "geoJSON" format (a format that represents geographic data). This module requires 2 main parameters :

1. **The geographic region** : We specify a "bounding box" as a set of 2 geographic coordinates, the bottom left and top right corners of a box (longitude and latitude in degrees for each point).
2. **Roads types** : OSM works with a system of "tags". A tag is a key-value pair. In our case, we are interested in the "highway" key tag which identifies all roads, streets and paths. Given that key, we can choose the road type : footway, motorway, cycleway, etc.)

After specifying the 2 main parameters, GFO will proceed to call the OSM API, convert the data to a GeoJSON graph-structured data file. However, before converting the data, it processes OSM raw data in order to correct the overall structure. Indeed, the raw data is not structured as a proper graph as illustrated hereunder. Nodes do not always correspond to proper road intersections for instance.

The GeoJSON file is structured as a set of features composed of

- **Point** : It contains geographic coordinates and an unique ID

[1] `https://www.openstreetmap.org/`
[2] `https://github.com/MatveiT/GraphFromOSM`

- **LineString** : It contains a source node ID, a destination node ID and a list of geographic coordinates defining the edge as well as other information (oneway, maxspeed, etc..)

### 6.1.2 Parsing and Pre-processing

Given the GeoJSON file, we can parse it and transfer the data to an actual usable data structure in the code. We choose to model the graph in the form of an adjacency list. In particular, it contains a set of key-value pairs. The keys are node's IDs and the values are lists of Edges adjacent to these nodes. Hence, parsing the GeoJSON file is easy as the structure is the same as an adjacency list. However, edge weights need to be computed based on the "Links" information. In particular, given the list of geographic coordinates constituting the edge between 2 points, we can compute the time needed to travel through the edge or its length in kilometers.

$$length\ km = \sum_{i=0}^{k-1} haversine(coord_i, coord_{i+1})$$

where $k$ is the number of coordinates and *haversine* computes the great-circle distance between 2 points on a sphere given their longitudes and latitudes.

$$haversine = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos\phi_1 \cos\phi_2 \sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right)}\right)$$

where $r = 6371km$ is the Earth radius, $\phi_1$ and $\phi_2$ are the latitude of point 1 and point 2 (in radians) and $\varphi_1$ and $\varphi_2$ are the longitudes (in radians). To get the edge traversal time, we simply divide the length in km by the speed of the considered vehicule.

- Foot : $5km/h$ in average for an adult
- Bike : $25km/h$ in average [3]
- Car : It depends on the road types : $30, 50, 120$..

Once we have the adjacency list, we may want to preprocess the parsed graph before applying the different algorithms. Let us denote the unprocessed graph by $G^u$

- $G^u$ may potentially contain parallel edges. That is, multiple edges sharing the same pair of nodes. The solution is to keep the edge with the lowest weight (distance)
- $G^u$ may also contain loop edges. That is, the source node is identical to the destination node. We simply avoid them and do not add them
- $G^u$ is not strongly connected. Indeed, since we choose an arbitrary bounding box during the query, some edges are not part of the graph. Also, some roads are disconnected from the rest of the network. The advantage of having a strongly connected graph is that we always have a valid pair of nodes to test our queries on. To transform $G^u$ to a strongly connected graph, we apply this procedure (from Victor's thesis [5]: Given $G^u$ and its reverse graph $\overleftarrow{G}^u$ as previously defined in 4.2.2, take any node $v$ randomly. Run Dijkstra's algorithm on both graphs, starting at the same node $v$. Those runs will yield two sets of search spaces

[3] `https://cyclinguphill.com/average-speeds-cycling/`

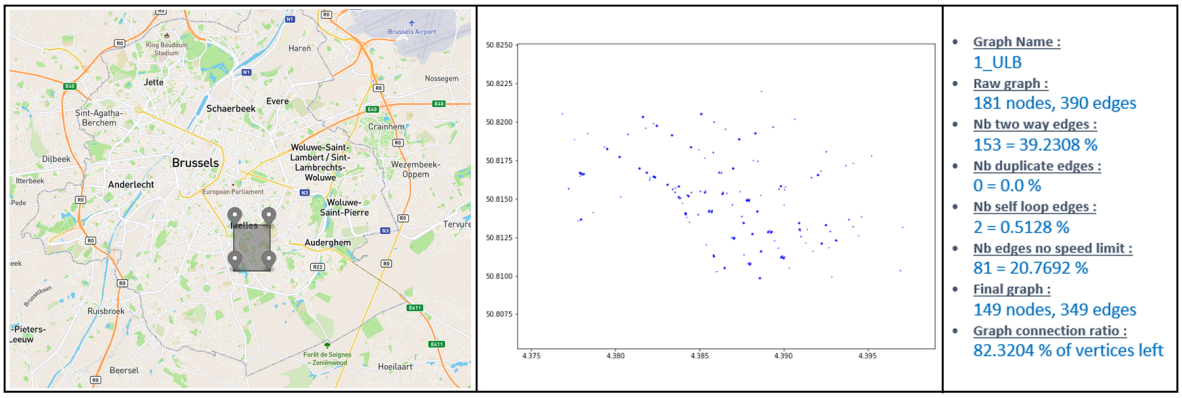
– $V^G$ : nodes that are reachable starting from $v$

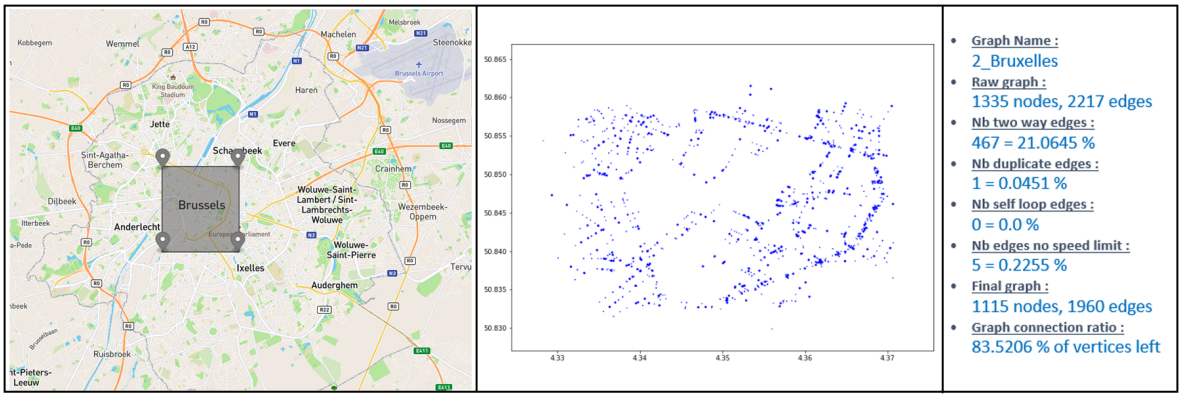
– $V^{\overleftarrow{G}}$ : nodes that are reachable starting from $v$ in the other direction

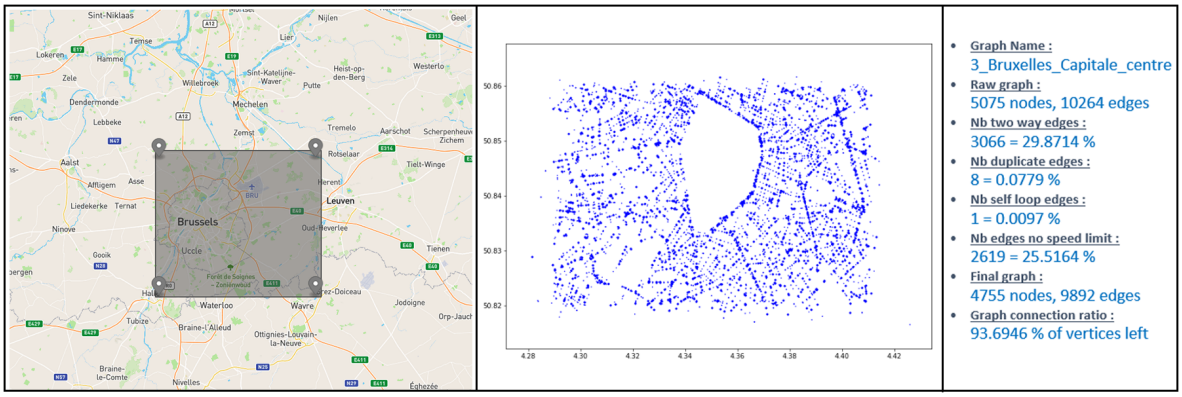
The intersection $V^G \cap V^{\overleftarrow{G}}$ contains only vertices such that each pair of vertices are mutually reachable, which means that this subset of nodes is strongly connected. We repeat the process with other nodes $v$ until we cannot get a bigger strongly connected graph. The final graph is $G = V^G \cap V^{\overleftarrow{G}}$.

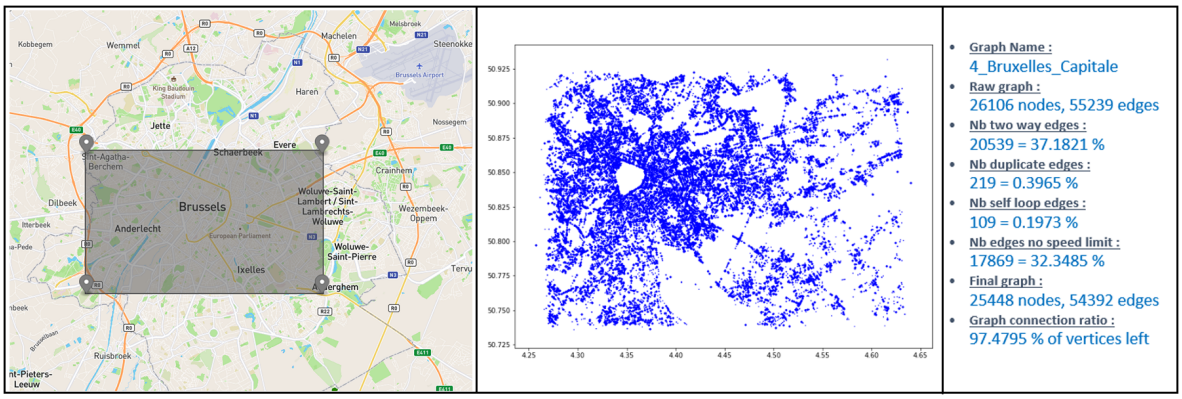
### 6.1.3 Graphs

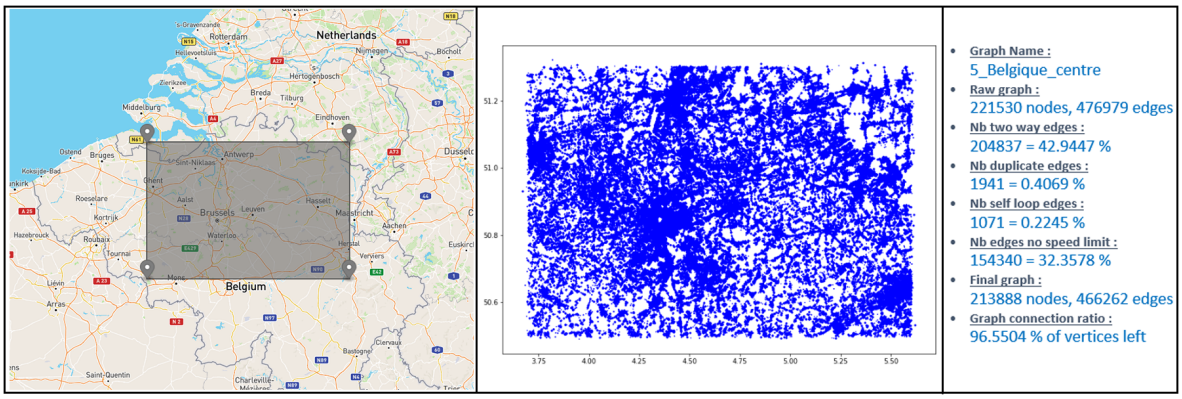
Now we will take a look at the real-world road networks we will use for the experiments. In total, 6 different graphs will be used, by increasing size. Due to some difficulties with the OSM API queries, we use some graphs that Victor [5] provided to us.





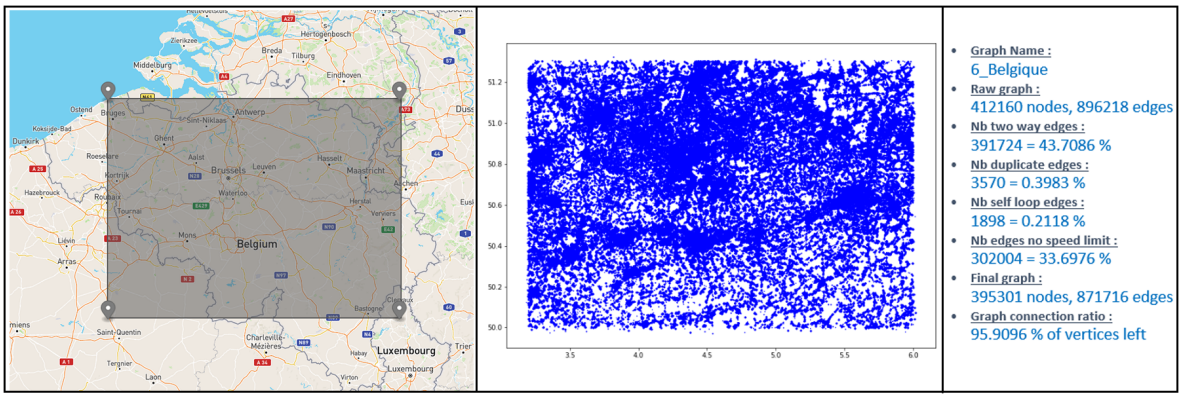




**Table 6.1** Graphs stats

| N° | Graph name | \|V\| | \|E\| | \|E\|/\|V\| | avg deg | size (Mo) |
|---|---|---|---|---|---|---|
| 1 | *ULB* | 149 | 349 | 2.34 | 2.34 | 0.28 |
| 2 | *Bruxelles* | 1115 | 1960 | 1.76 | 1.76 | 1.01 |
| 3 | *Bruxelles Capitale centre* | 4755 | 9892 | 2.08 | 2.08 | 7.62 |
| 4 | *Bruxelles Capitale* | 25448 | 54392 | 2.14 | 2.14 | 35.4 |
| 5 | *Belgique centre* | 213888 | 466262 | 2.18 | 2.18 | 263 |
| 6 | *Belgique* | 395301 | 871716 | 2.21 | 2.21 | 488 |

## 6.2 Design choices

### 6.2.1 Data structures

#### Static uni-modal Graphs

As mentionned in 6.1.2, we choose to model the graphs with adjacency lists. Given a graph $G = (V, E)$, the adjacency list $A$ is a set of key-value pairs where each key is the ID of a node $v$ and each value is the list of edge adjacent to $v$.

Another possibility would be adjacency matrices, which is a squared matrix of $|V| \times |V|$ elements. Each cell $(i, j)$ contains a binary value determining whether a directed edge $(i, v)$ exists or not. However, it is not suitable with road networks considering their sparsity (which is confirmed with our data if we look at the average nodes degrees of table 6.1). Indeed, a lot of cells would be empty, which would lead to high memory utilisation.

As stated in chapter 5, our graphs will not be dynamic nor time dependent. Indeed, we will not consider time-dependency and the edge weights will not change once they are fixed. Furthermore, we will not add, delete or replace nodes or edges during the execution of the algorithms.

#### Static multi-modal Graphs

To construct the multimodal graphs, we also use an adjacency list. We simply duplicate the edges or add some edges depending on the considered modality like described in chapter 5. However, we choose to make 2 assumptions for simplicity sake :

- The base layer $\mathcal{L}_O$ can be a free-floating car layer and we assume there is a station at each node. It will be useful in order to have bi-modal car-bike networks
- We do not allow the user to discard any modalities. Indeed, as mentioned before, a user may want to avoid switching between modes of transport too often. In our case, we keep every chosen modalities.

We will elaborate on these ideas during the experiments on multimodal graphs.

#### Quadtree

For our experiments, we have decided to implement the following algorithms : Dijkstra, Astar, ALT, Bidirectional Dijkstra, Bidirectional Astar and Bidirectional ALT. For shortest path queries, all the algorithms can be applied to the adjacency lists representing the graphs. However, ALT and Bidirectional ALT need to apply a preprocessing phase (like described in section 4.3). Hence, landmarks have to be determined and distances to those landmarks from all nodes must be computed. Although it can be achieved using simple adjacency lists, we used a more sophisticated data structure : **a Quadtree**. It is a two-dimensional tree data structure in which each node has exactly 4 different children. The leaves can be any sort of unit data. Quadtrees follow these interesting properties :

- Space is decomposed into several "cells"
- Each cell (bucket) has a limit of unit data and when this is limit is crossed, the bucket is split into smaller cells

There are multiple variants of quadtree. We focus on the point quadtree. Indeed, it is an adaptation of a binary tree used to represent two-dimensional point data. This data structure exhibits $\mathcal{O}(\log(n))$ search times for given point data, which makes it very useful for searching on a 2D grid. Hence, it can be interesting to use for the preprocessing phase of ALT algorithm, especially for landmarks selection. Each node of the graph is added to a quadtree. In particular, it can be useful in order to search for the closest node of a given pair of geographic coordinates. Speeding the landmark selection process will not bias our benchmarks since it is just reducing the overall preprocessing computation time by the same factor. Given a graph, it will always take the same amount of time to find the positions of the landmarks.

As presented in section 4.3, we use 3 different methods for finding the landmarks : planar selection, farthest selection and random selection. Planar and farthest can benefit from the quadtree structure. Indeed, each of them starts by picking the central node of the graph. The quadtree takes the central node's coordinates and finds the closest node. We illustrate the 3 landmarks selections hereunder (with $k = 16$ on graph 2) :

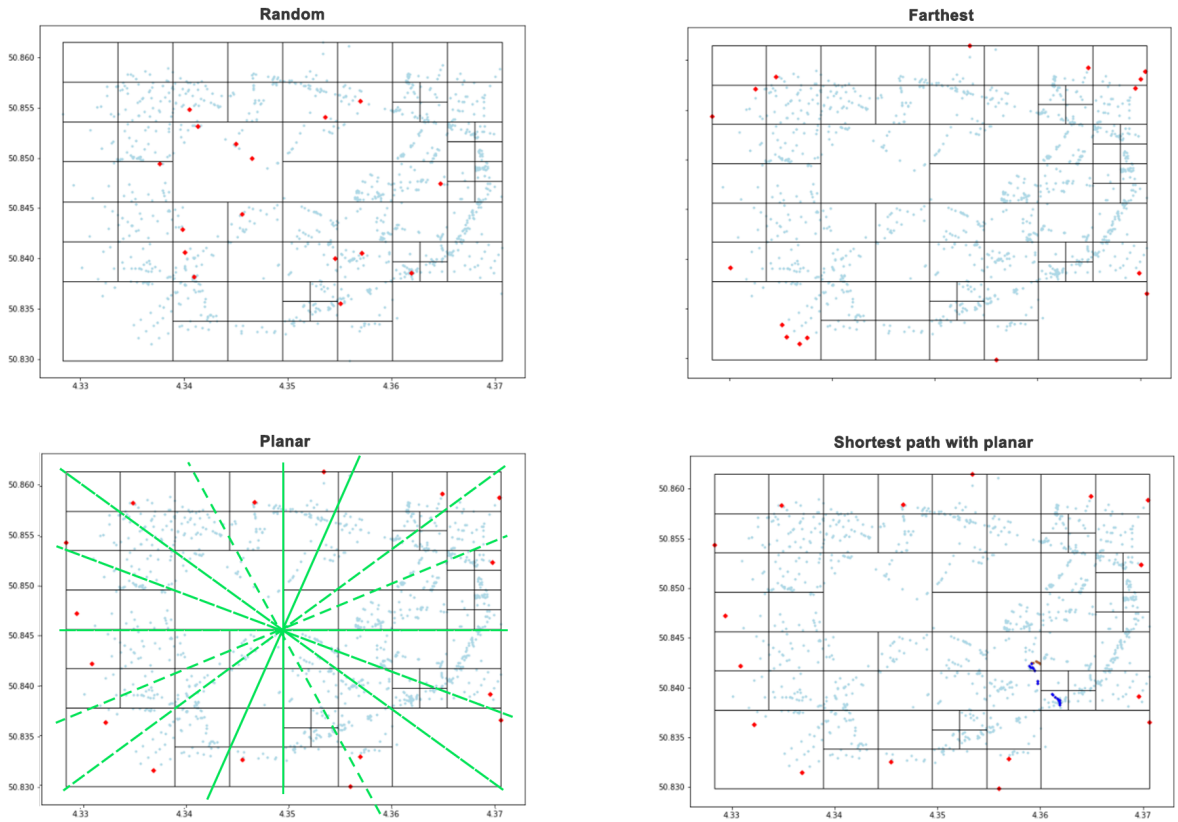
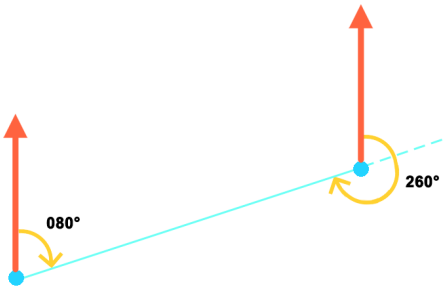



Figure 6.1: Landmark selections on Quadtree

For the planar landmark selection, we first separate the area of the graph into k sub areas using bearing angles. Then, we find a landmark that is the farthest from center in each region. The bearing angle is the horizontal angle between the direction of an object and another object. In particular, the absolute bearing is the clockwise angle between the magnetic north and an object. In our case, for each node of the graph, we compute the absolute bearing angle of this node from the origin node (using geographic coordinates). This way, we can

determine which node belongs to which region. To compute the bearing angle between 2 points $(lat1 = \phi_1, lon1 = \varphi_1)$ and $(lat2 = \phi_2, lon2 = \varphi_2)$ in radians, we have this formula :




$$\beta = \text{atan2}(X, Y)$$

$$Y = \sin(\varphi_2 - \varphi_1) * \cos(\phi_2)$$

$$X = \cos(\phi_1) * \sin(\phi_2) - \sin(\phi_1) * \cos(\phi_2) * \cos(\varphi_2 - \varphi_1)$$

Figure 6.2: Example of Bearing angle

### 6.2.2 Villo

In order to implement station-based multi-modal graphs, we need station nodes. We use the Villo network of Brussels (355 stations) using "Overpass Turbo", a tool that allows us to gather OSM data. In practice, we have to find the closest node in the base layer for each Villo-station. We will use the following graphs in combination with villo stations :

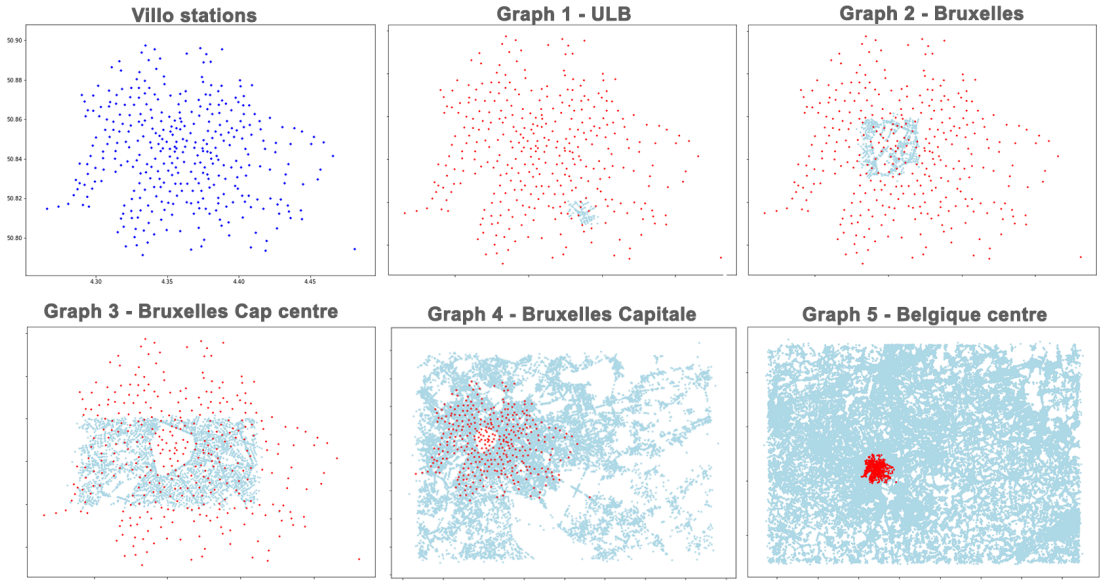



Figure 6.3: Villo stations (red) - Graphs 1 to 5 of 6.1

### 6.2.3 Implementation

In order to program the algorithms and the experiments, we choose Python as a programming language. Considering the fact that we are working with speed-up techniques, the logical and reasonable choice would have been to take C++ or Java. However, Python offers the advantage of faster development and the goal is not to have the best possible timing but to see if algorithms such as ALT can handle multimodal networks.

All the code, as well as all the reproducible experiments results, are available on Github [4]. We choose an object-oriented approach to implement all the algorithms. A class diagram can be found in Appendix D.

### 6.2.4 Plots metrics

We define now the metrics (and their notations) that will be used in the experiments plots.

- **QT** : average query time over 100 runs : expressed in seconds. We used the `perf counter` method from the Python *time* library to measure precise timings.
- **SS** : average Search space : number of nodes that has been explored during the algorithm (e.g: Dijkstra) over 100 runs. For Dijkstra's algorithm, we have the following expected search spaces :
  - Single-source : $|V|$
  - Single-pair : $\frac{|V|}{2}$ nodes $v$ such that $d(s,v) < d(s,t)$ for randomly chosen $s$ and $t$.
- **RS** : average Relaxed space : number of edges that has been relaxed during the algorithm (e.g: Dijkstra) over 100 runs
- **Improvement:**
  - **Speedup** : speedup in terms of computation time : QT algo 1 / QT algo 2
  - **Search space** : improvement of the number of nodes in the search space : SS algo 1 / SS algo 2
  - **Relaxed edges** : improvement of the number of relaxed edges in the relaxed space
- **Average max lb:** The maximal average lower bound of ALT algorithm. This metric measures how close the distance lower bounds are to the actual shortest distance. We saw in 4.3.5 that the higher the lower bound is, the faster ALT becomes.

$$\frac{\sum_{v \in V} \max(d(\lambda,t) - d(\lambda,v), d(v,\lambda) - d(t,\lambda))}{|V|}$$

### 6.2.5 Specifications

CPU time measurements are based on a specific implementation and run-time environment of the given algorithms. Therefore, we specify the run-time environment :

- Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz, 4 cores
- 16 Go RAM, 1333 MHz, DDR3, DIMM
- Operating system : Windows 10, version 1909

During the experiments, each benchmark is composed of **100** runs of random of s-t pairs (unless another number is specified during an experiment) and are then averaged.

The plots were constructed using Python's *`matplotlib`* library.

[4] `https://github.com/AlexandreHnf/MasterThesis`

## 6.3 Experiments

### 6.3.1 Single-modality - Dijkstra and A* parameters

#### Experiment 1

For the first experiment, we compare the 3 priority queue data structures using Dijkstra on the same set of s-t pairs in order to determine which is the fastest : simple list, Fibonacci heap (implemented using Python *fibheap* library) or binary heap (implemented using *heapq* library).

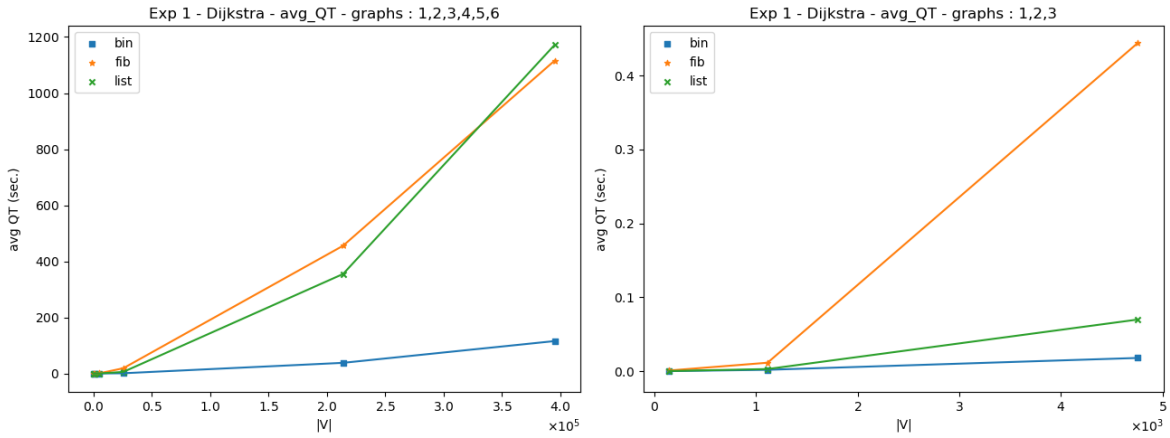



Figure 6.4: Exp 1 - Average query times. **Left**: graphs 1 to 6, **Right**: graphs 1 to 3

On smaller graphs (1 and 2), all methods share the same results. On bigger graphs (3 to 6), the binary heap yields good results, while Fibonacci heap and the simple list seem to produce poorer computation times. We have seen in table 4.2, that Dijkstra coupled with Fibonacci heap exhibits better time complexity ($\mathcal{O}(|E| + |V| \log |V|)$ than binary heap ($\mathcal{O}((|E| + |V|) \log |V|)$) so we might expect it to perform better. However, we also saw in 4.1.3 that in practice, Fibonacci heaps are not as efficient as other forms of heaps that are less efficient due to the use of more pointers internally. For the rest of the experiments, we will keep the binary heap.

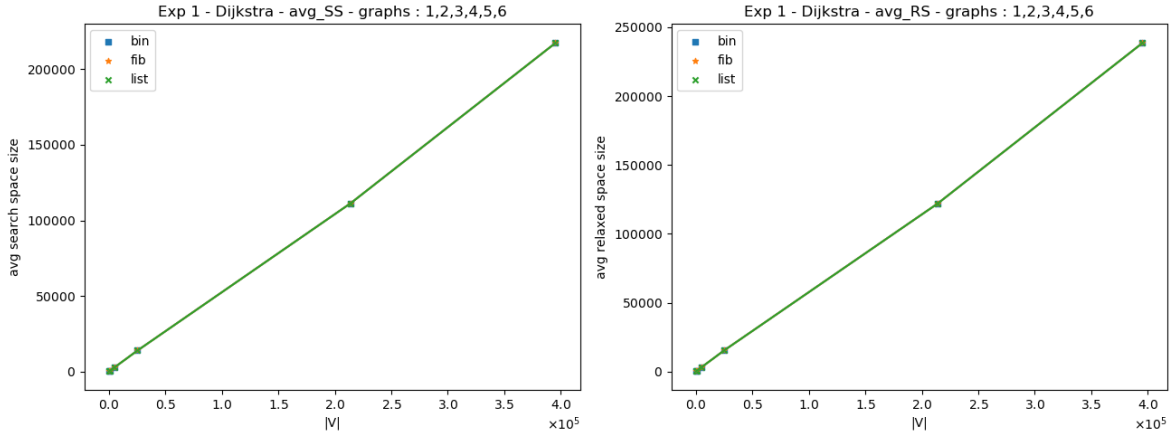



Figure 6.5: Exp 1 - **Left**: Average search space, **Right**: Average relaxed space

If we look at the average search spaces and average relaxed spaces, we can notice that both have the same behaviour, except that there are more relaxed edges than visited nodes. It is expected since there are two times more edges than nodes (as shown in table 6.1) and each time we arrive at a node, Dijkstra potentially relaxes several out-going edges. Hence, for future experiments, we could only concentrate on one of them.

### Experiment 2

For experiment 2, we compare A* heuristics on the same set of s-t pairs (Euclidean distance, manhattan distance, octile distance) to determine which one gives better results in average.

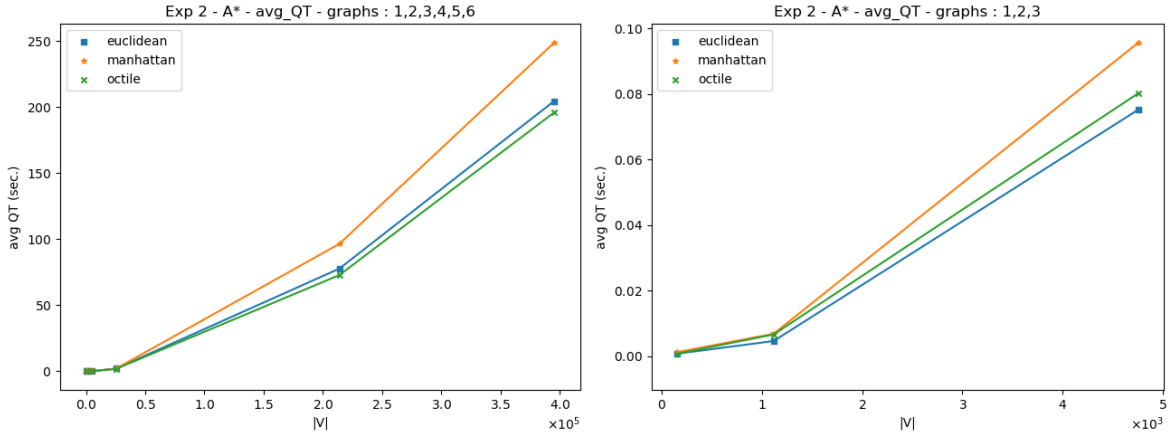



Figure 6.6: Exp 2 - Average query times. **Left**: graphs 1 to 6, **Right**: graphs 1 to 3

We can see that A* with manhattan distance gives slightly worse results, while the two others are similar. The reason is that Euclidean distance is composed of 1 haversine computation whilst manhattan and octile need 2 haversine computations. This small difference leads to an overhead in the long run. We can see that the difference in average query time is negligible for small graphs, but becomes significant in bigger graphs. Since euclidean distance seems to be the fastest, we keep this heuristic for the rest of the experiments.

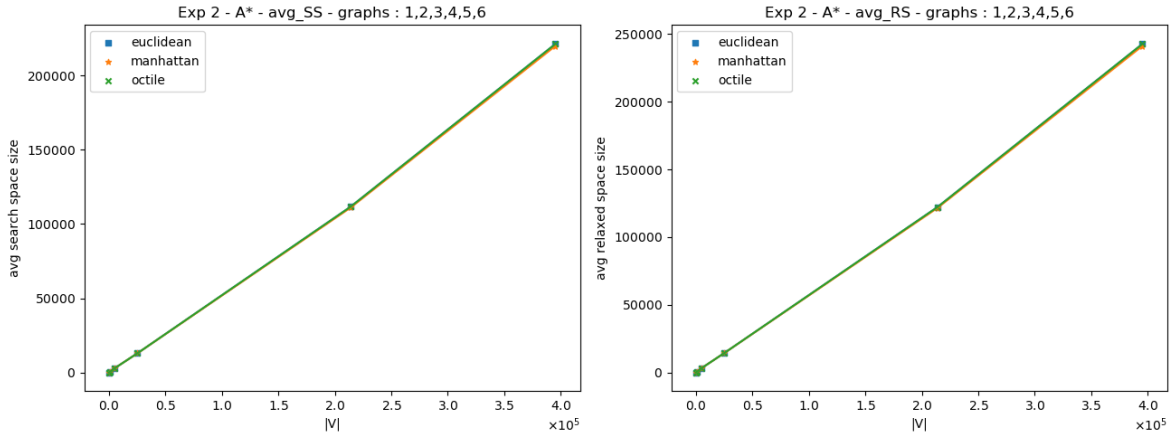



Figure 6.7: Exp 2 - **Left**: Average search space, **Right**: Average relaxed space

As expected, the relaxed edge space and search space are nearly identical for the 3 heuristics. Indeed, the heuristic distances direct the search to the same nodes in the graph.

### 6.3.2 Single-modality - ALT parameters and preprocessing

In this section, we test different parameters of ALT speed-up technique on single-modal road networks.

#### Experiment 3

For the third experiment, we focus on landmark selection for the preprocessing of ALT algorithm. We compare the 3 methods on the same set of s-t pairs: farthest, planar, and random landmark selections. Internally, to implement ALT preprocessing, we have applied the following steps :

- Select k landmarks given a specified methodology (farthest, planar or random),
- Compute all distances from each node to each landmark. Instead of applying Dijkstra algorithm on every pair $v - \lambda$, we first reverse the graph, then for each landmark, we apply the single-source Dijkstra's algorithm to obtain the distances from a landmark to every node in $\overleftarrow{G}$. In the end, we have the distances from each node to every landmark,
- Run the $k$ single-source shortest paths on $k$ threads. It will speed-up the preprocessing computation by preserving the multiplication factor between different graphs query times (as we will see in experiment 5).

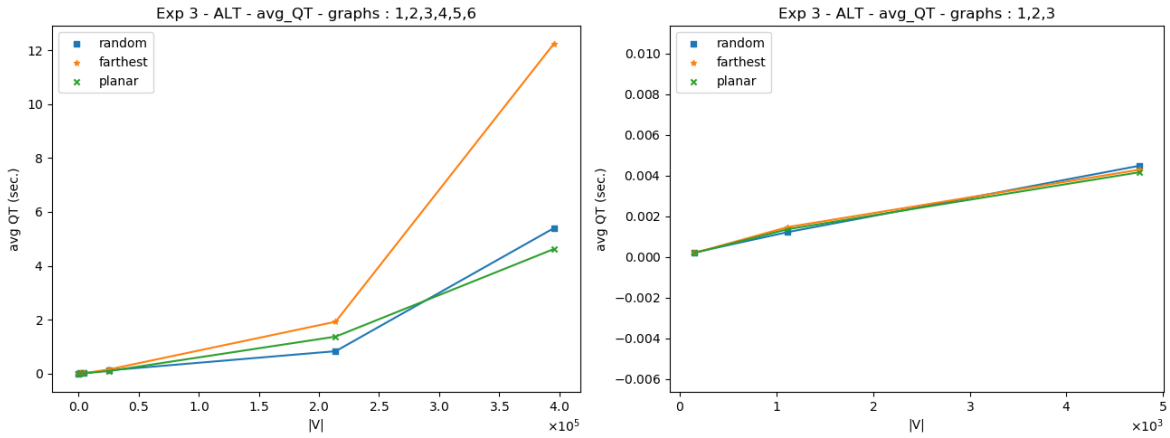



Figure 6.8: Exp 3 - Average query times. **Left**: graphs 1 to 6, **Right**: graphs 1 to 3

On smaller graphs (1,2 and 3), we can see that the difference in terms of an average query time is negligible. But when we transition to bigger graphs, a clear difference appears. Farthest landmark selection yields 2x slowdown compared to planar and random. Random landmark selection produces surprisingly good results.

We will see in experiment 5 which selection leads to the best preprocessing time.

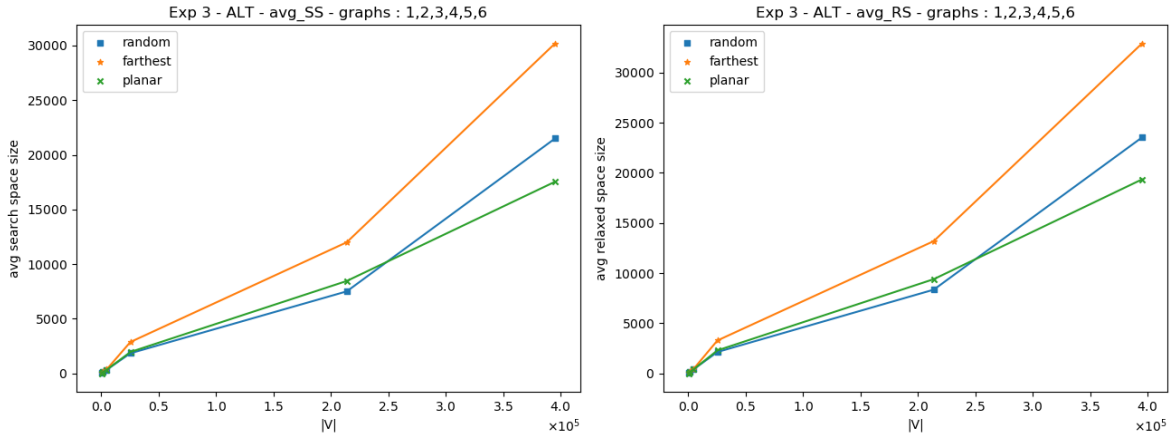



Figure 6.9: Exp 3 - **Left**: Average search space, **Right**: Average relaxed space

As we can see from the average search space and average relaxed space plots, farthest landmark selection produces poorer lower bounds than the others, hence the difference in terms of relaxed edges amount.

### Experiment 4

The experiment 4 focuses on landmarks number. We look at the behaviour of ALT when we change k on the set same of s-t pairs. We test the following values : [1, 2, 4, 8, 16, 32].

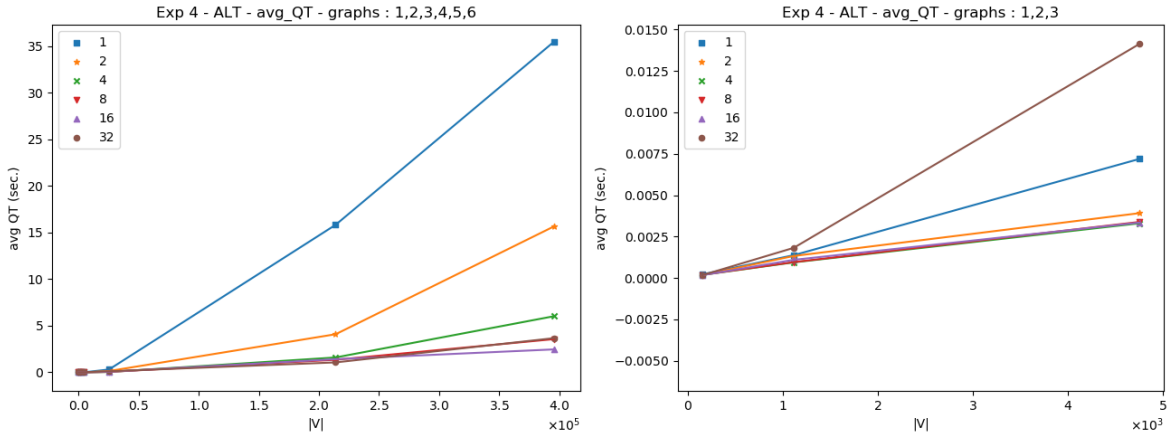



Figure 6.10: Exp 4 - Average query times. **Left**: graphs 1 to 6, **Right**: graphs 1 to 3

In fig 6.10, we observe a clear negative correlation between the number of landmarks and average query time. The more landmarks we have, the quicker ALT queries are performed.

However, we can see that this difference decreases as the number of landmarks increases, meaning that adding further landmarks will not yield drastic improvement in query times.

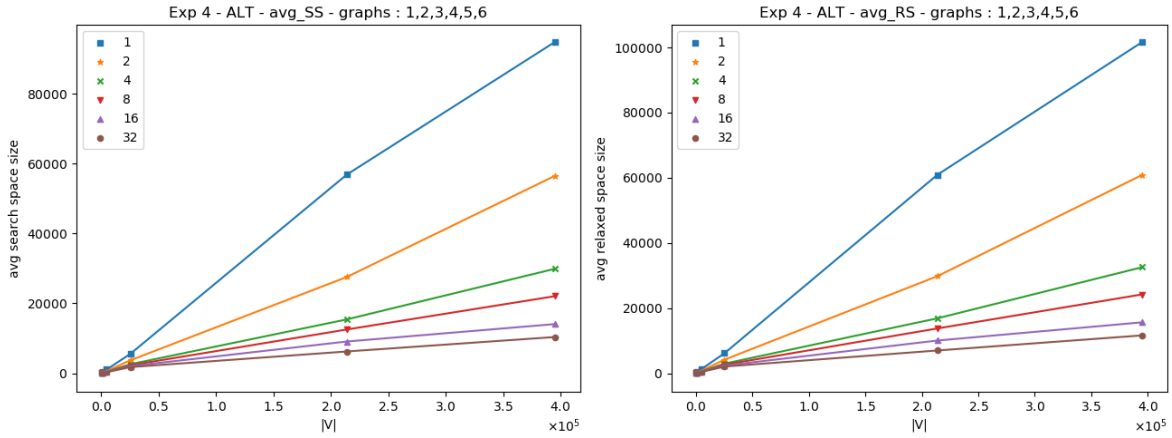



Figure 6.11: Exp 4 - **Left**: Average search space, **Right**: Average relaxed space

Fig 6.11 confirms the previous observations as it shows that a bigger set of landmarks provides better actual distance lower bounds, hence fewer edges to relax and fewer nodes to visit.

For future experiments, we keep $k = 16$ as the results are not significantly different when using $k > 16$ and it saves some preprocessing time as we will see in the next experiment.

### Experiment 5

The experiment 5 deals with ALT preprocessing and its parameters. In particular, we first show the preprocessing times of experiment 3 (comparing random, farthest and planar landmark selections). Then, we show the preprocessing times based on experiment 4 (comparing different amounts of landmarks). Finally, we select the best parameters according to the two former experiments and we show the preprocessing times.

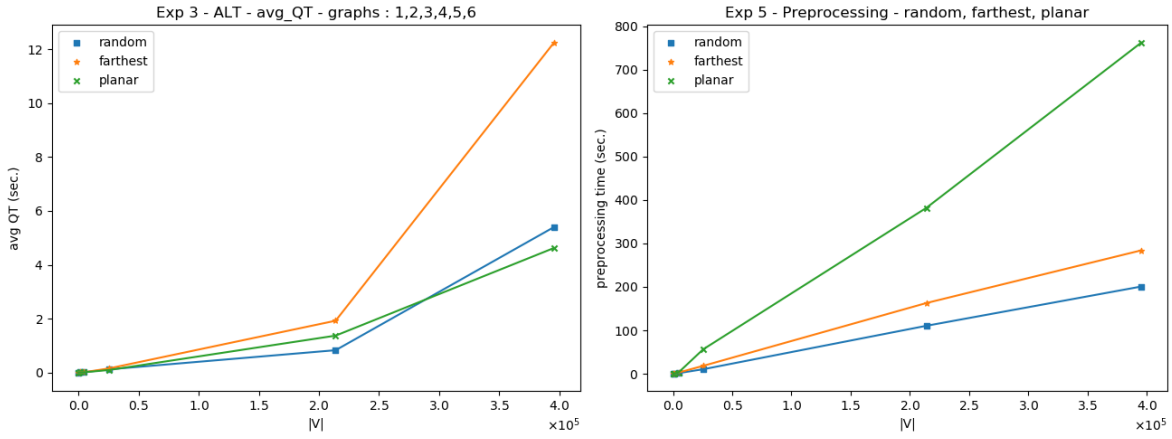



Figure 6.12: Exp 5 - **Left**: Exp3 query times, **Right**: Preprocessing times

As we can see, planar landmark selection is way slower than farthest and random as expected. This selection procedure has to split the entire area into $k$ regions and then find the farthest node on each region. Random landmark selection is obviously the quickest as it only selects $k$ nodes randomly.

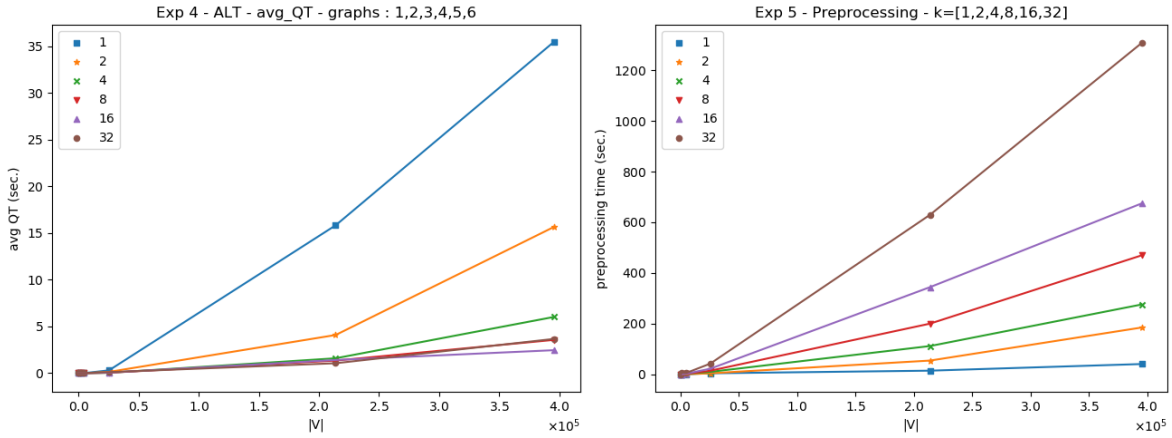



Figure 6.13: Exp 5 - **Left**: Exp4 query times, **Right**: Preprocessing times

Fig 6.13 shows the preprocessing times of experiment 4. As expected, the preprocessing time is inversely proportional to the query times. Furthermore, the preprocessing increases rapidly with the number of landmarks. The preprocessing time with $k = 32$ is almost twice as high as with $k = 16$. The multiplication factor is nearly exponential.

This leads to a tradeoff between preprocessing time and ALT query time. Among random, farthest and planar selections, we keep planar as it provides the best query times but at the expense of higher preprocessing time. Also, for bigger graphs, planar selection is more reliable than random selection.

As far as the number of landmarks is concerned, we keep $k = 16$ as mentioned in experiment 4 analysis. It offers great query times at the expense of relatively reasonable preprocessing times.

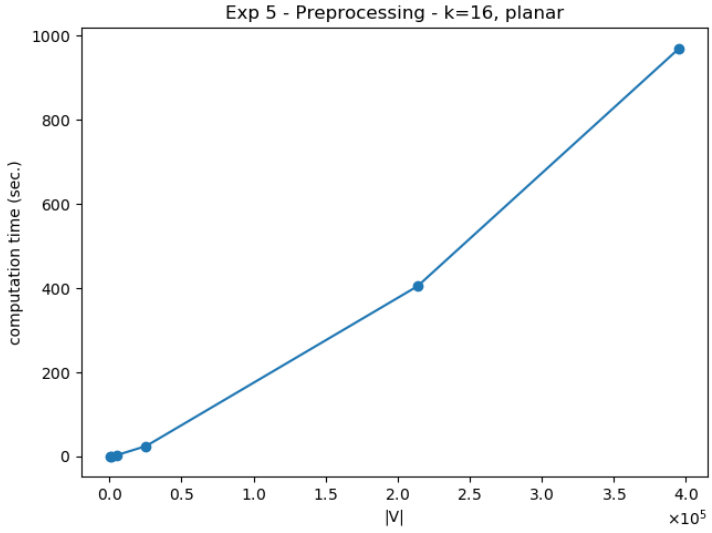



Figure 6.14: Exp 5: Preprocessing times - k = 16, Planar

Now that we selected the best parameters regarding our empirical experiments, we have the preprocessing times in fig 6.14 for graphs 1 to 6. The preprocessing time of ALT increases

quickly with the graph size but stays reasonably low considering the number of nodes in the biggest graph.

Note that we could have applied a combinatorial optimisation method to determine the best possible combination of parameters (such as genetic algorithm). However, it is not the purpose here. Besides, as it is shown in [77], most of the preprocessing-based speed-up techniques for Dijkstra (such as ALT, Arc-Flags, SHARC, Highway Node Routing and Contraction hierarchies) have some degrees of freedom which are NP-hard to determine optimally. That is why we can use heuristics in practice to determine them. In particular, selecting a set of k landmarks that minimizes the search space for random single-source single-destination queries is shown to be NP-hard.

### 6.3.3 Single-modality - Algorithms

#### Experiment 6

In experiment 6, we compare 6 different algorithms in terms of preprocessing time, query time, search space and relaxed space : Dijkstra, A*, ALT, Bidirectional Dijkstra, Bidirectional A* and Bidirectional ALT.

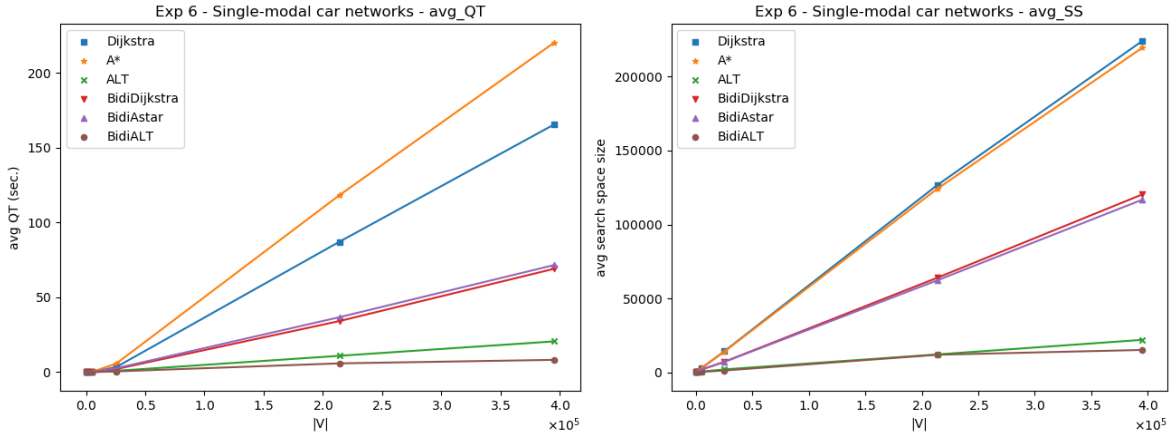



Figure 6.15: Exp 6 - **Left**: Average query times, **Right**: Average search space

As expected, both query time and search space are linear in the network size for each algorithm as we can see in fig 6.15. However, each algorithm has a different slope which characterize their differences. Dijkstra's algorithm shows a slope of $\frac{1}{2}$ which corresponds to its theoretical expected search space $\frac{|V|}{2}$ as mentioned in 6.2.4.

When we look at A*, we can notice that there is a slow down compared to Dijkstra. It can be explained by two reasons : According to [29], the classical way to use A* for route planning in road maps estimates $d(v,t)$ based on Euclidean distance between $v$ and $t$. However, this is a very conservative estimation and the speedup for finding the quickest routes is rather small. Goldberg et al. [38] even report a slow-down of more than a factor of two since the search space is not significantly reduced. The second reason is that the computation of the heuristic estimate at each relaxation creates a considerable overhead in the long run. As we can see from the right plot of fig 6.15, A* search space is indeed very close to that of Dijkstra. It is confirmed in the right plot of fig 6.16 hereafter.

Contrariwise, Bidirectional searches and ALT yield significant improvement in terms of query time and search space. As expected, Bidirectional Dijkstra reduces the search space size by half (as described in 4.2.4). It is confirmed in fig 6.16 hereafter. A* search space improvement is negligible compared to that of ALT and Bidirectional searches.

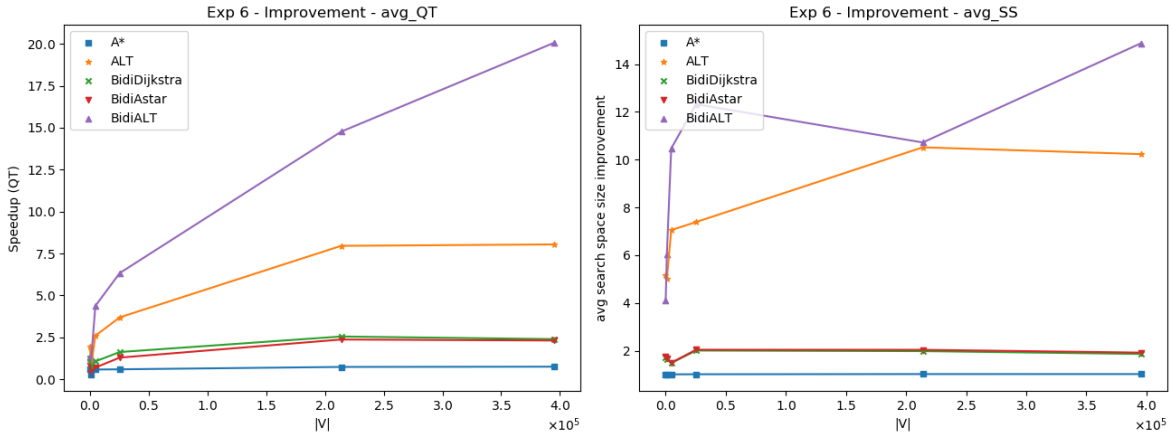



Figure 6.16: Exp 6 - **Left**: Average query times improvement, **Right**: Average search space improvement

**Table 6.2** Query times and Speed-ups

| | Pre | Query time (sec.) and Speed-ups (su) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| G | (sec.) | D | A* | su | ALT | su | BA | su | BA* | su | BALT | su |
| 1 | 0.061 | 0.0003 | 0.0004 | 0.75 | 0.0001 | 2.3 | 0.0003 | 1 | 0.0003 | 1 | 0.0002 | 1.5 |
| 2 | 0.496 | 0.002 | 0.0069 | 0.28 | 0.002 | 1 | 0.0028 | 0.7 | 0.0049 | 0.4 | 0.0017 | 1.2 |
| 3 | 2.642 | 0.091 | 0.156 | 0.58 | 0.035 | 2.6 | 0.085 | 1.1 | 0.130 | 0.7 | 0.02 | 4.6 |
| 4 | 28.24 | 3.385 | 5.69 | 0.59 | 0.917 | 3.7 | 2.083 | 1.6 | 2.62 | 1.3 | 0.534 | 6.3 |
| 5 | 347.9 | 87.2 | 118.31 | 0.73 | 10.95 | 7.9 | 34.17 | 2.6 | 36.76 | 2.4 | 5.905 | 15 |
| 6 | 680.9 | 165.7 | 220.43 | 0.75 | 20.599 | 8 | 69.169 | 2.4 | 71.652 | 2.3 | 8.254 | 20 |

Hereabove, we gathered all query times of the 6 algorithms over the 6 graphs (G) as well as their speed-ups (su) with regard to Dijkstra (D).

We can illustrate the search space of Dijkstra, A*, ALT and Bidirectional Dijkstra using the following quadtrees. We ran all these algorithms on the same random s-t shortest path in graph 4. We can see that A* search space is nearly identical to that of Dijkstra. Some nodes are not appearing in brown in the right plot of fig 6.17 but it is not visible. Fig 6.18 shows that the search space is almost only composed of nodes present in the shortest path (in blue). Finally, we can see that the search space of Bidirectional Dijkstra is indeed halved.

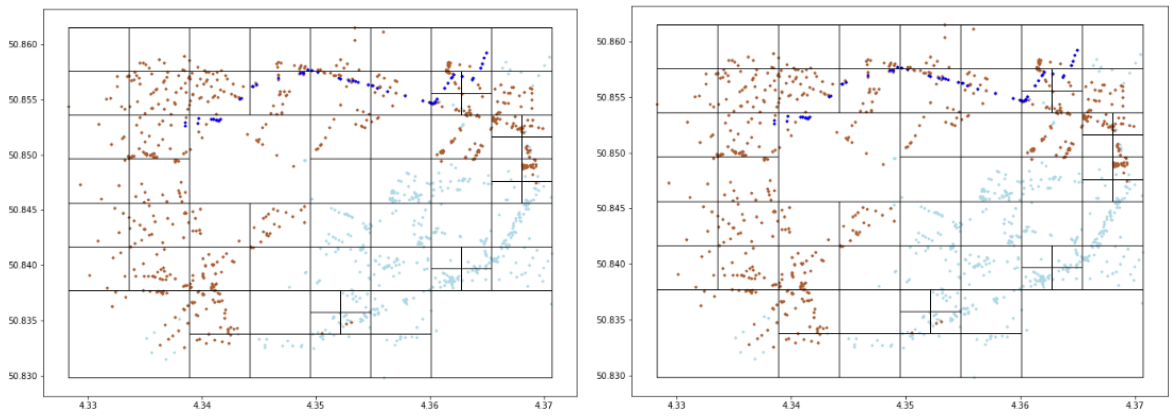


Figure 6.17: Exp 6 - **Left**: Dijkstra's search space, **Right**: A* search space

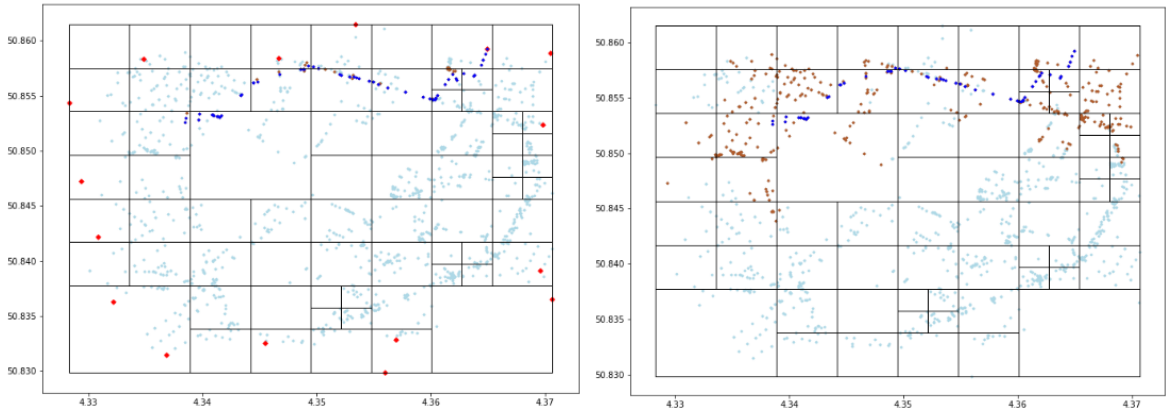


Figure 6.18: Exp 6 - **Left**: ALT's search space, **Right**: Bidirectional Dijkstra search space

It confirms the theoretical search space analysis done in 4.2.4.

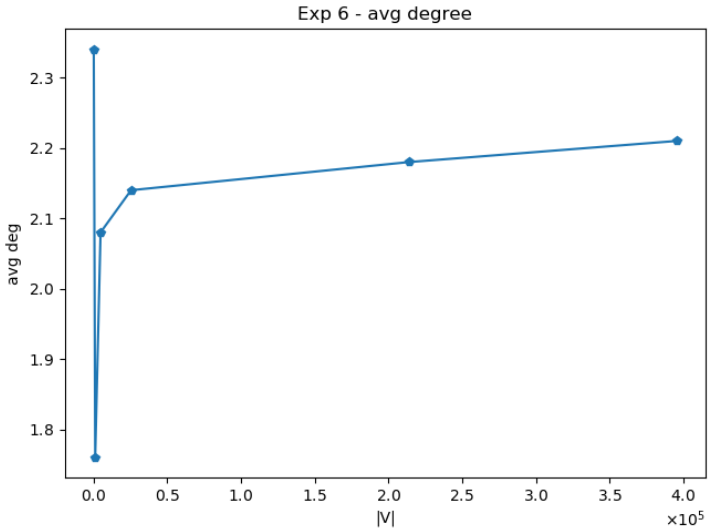



Figure 6.19: Exp 6 : average node degree

If we look at the average node degree of each graph, we can see that graph 1 has more outgoing edges on average than the others. Thus, it explains why there is a bigger improvement in terms of average relaxed space (in fig 6.16).

### 6.3.4 Multi-modality - Public transport

**Experiment 7**

For the 7th experiment, we will try to reproduce the same experiment as the experiment 7.7 of [5]. We take graph 4 (Bruxelles Capitale) composed of 25448 nodes and 54392 edges. Then, we transform it into a multi-modal network by adding layers simulating public transport lines. Concretely, the multimodal graph is composed of :

- A base layer $\mathcal{L}_0$ : edges weights are time lf traversal by foot (In average, the walking speed of an adult is 5km/h [5]).
- A set of public transport line layers $\mathcal{L}_i$ composed of single edges for a fixed number $|added\_edges|$. Each line is a single added edge that is adjacent to 2 randomly chosen nodes in the base layer network $\mathcal{L}_0$. As stated in [5], ideally, we should choose $|added\_edges|$ so that $|E|$ increases by maximum 2%, which means that overall it does not change that much (for graph 4 composed of 54392 edges, $|added\_edges| = 1000$).

It is basically the structure we described in subsection 5.3.3 except that we do not add forward and backward transition edges :

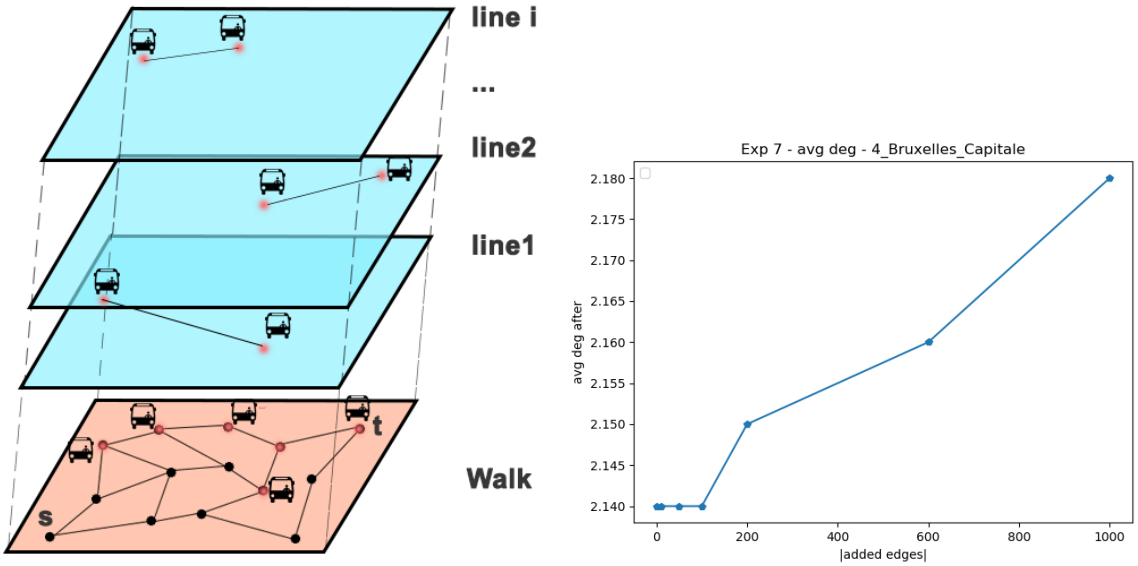



Figure 6.20: Exp 7 - **Left**: multi-modal public transport network, **Right**: Average node degree

We test the following speeds for the public transport lines : 0.1 (extreme low case), 15 (bus/tram) [6], 30 (metro) [7], 90 (train) and 1e10 (representing $\infty$, extreme high case) km/h

[5] https://www.irontimepieces.fr/blog/course-et-marche/marche-normale-vitesse-moyenne.html

[6] https://www.lalibre.be/regions/bruxelles/2016/05/26/des-trams-et-des-bus-encore-plus-lents-a-bruxelles-4ITQYXEGSZGYRFXOYYOWH2OMRE/

[7] https://fr.wikipedia.org/wiki/M%C3%A9tro_de_Bruxelles

and the following number of added edges : 0, 10, 50, 100 and 200 to see if it brings a difference in terms of search space size. [5] tested Contraction hierarchies on such network, we will test ALT algorithm. For each nb of added edge step, we use the same set of s-t paths for all speeds (**500** s-t pairs for each parameters combinations in this experiment).

The right side of fig 6.20 (above) reports the average node degree with the increasing number of added edges. We can see that except for the first steps, the degree increases a little with the number of added edges.

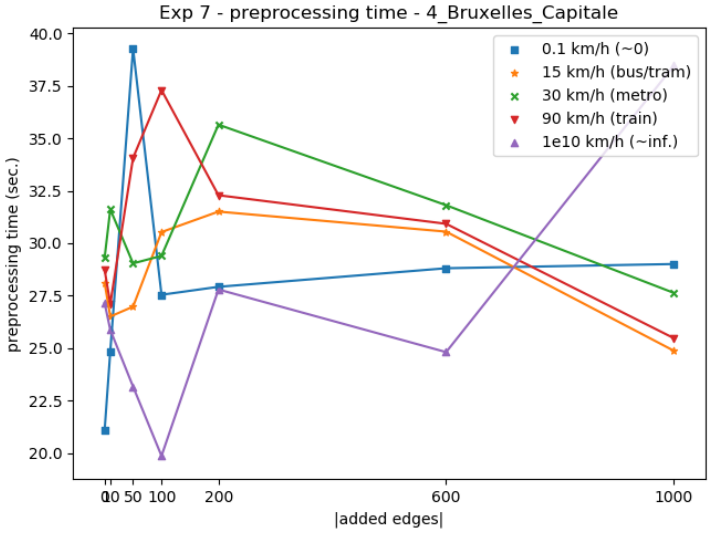



Figure 6.21: Exp 7 : Preprocessing times

In terms of preprocessing times (see fig 6.21), there is no drastic increase in the number of added edges. Indeed, the times vary from 22 seconds min to 40 seconds max and 30 seconds on average, which is a small overhead with respect to the single-modal timing of 28 seconds (see table 6.2).

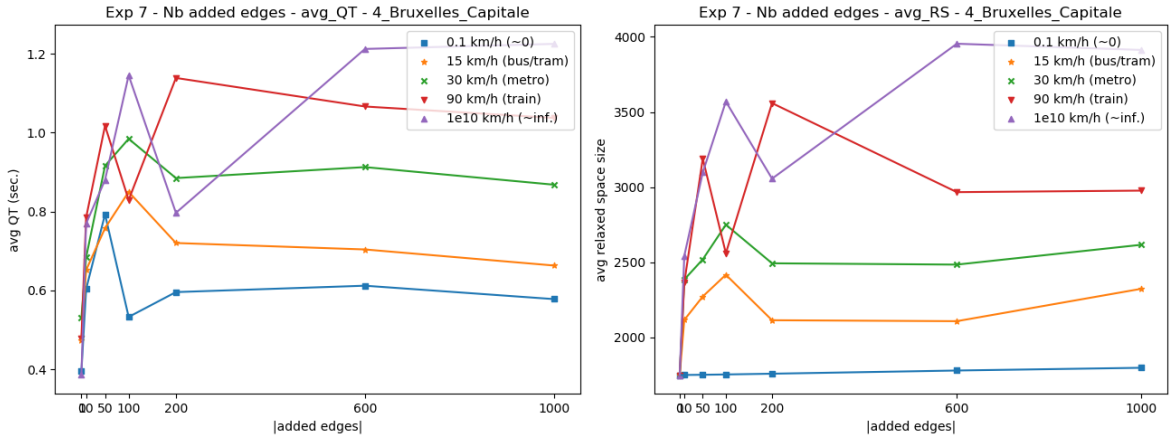



Figure 6.22: Exp 7 - **Left**: Average query times, **Right**: Average relaxed space

Fig 6.22 exhibits 2 interesting patterns. Firstly, the average query time and average relaxed space increase with the number of added edges, which is not really surprising. We assume that

the fewer public transport lines there are, the less likely they are selected by the algorithm. When the number of added edges is high, the search can benefits from those lines, especially when their speed is high, hence the increase in the number of relaxed edges (we can see that from the relaxed search space) of fig 6.22. However, we can notice that when going from 600 added edges to 1000, the query time decreases a little. Maybe it will decrease further if we keep adding lines.

Secondly, when the network gets bigger, we might expect the query time to be inversely proportional to the edge speed. However, it is not the case. 15km/h and 30 km/h yields better query times than 90km/h and 120 km/h within 200 added edges. It is due to a high number of relaxed edges when the speed gets higher since those edges are interesting to take.

If we run Dijkstra's algorithm on the same multi-modal network, and report the improvement in terms of average query time and relaxed space, we get fig 6.23 hereunder. It matches the previous results as the speedup decreases with the number of added edges, except with 0.1km/h as edge speed.

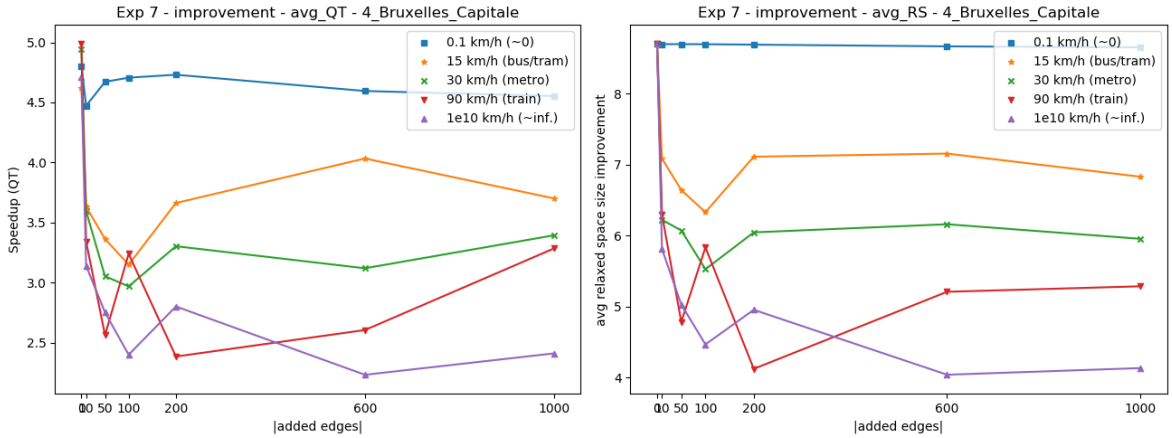



Figure 6.23: Exp 7 - **Left**: Average query times improvement over Dijkstra, **Right**: Average relaxed space improvement

In fig 6.24 hereunder, we have, to the left, the maximal average distance lower bounds of ALT. As we stated in 6.2.4, the higher the lower bound is, the faster ALT becomes. The results of the previous plots are indeed reflected here with the decreasing lower bounds, except for 0.1km/h that remains constant and independent of the number of added edges. It could mean that edges with such speed are never chosen.

To the right of fig 6.24, we have the average modalities repartition. That is, for each speed, the average percentage of each modality in the shortest path over all added edges steps. As we can see, the shortest path is mostly composed of foot edges. However, we can see that the percentage of public transport lines used increases a little with the speed, which is expected.

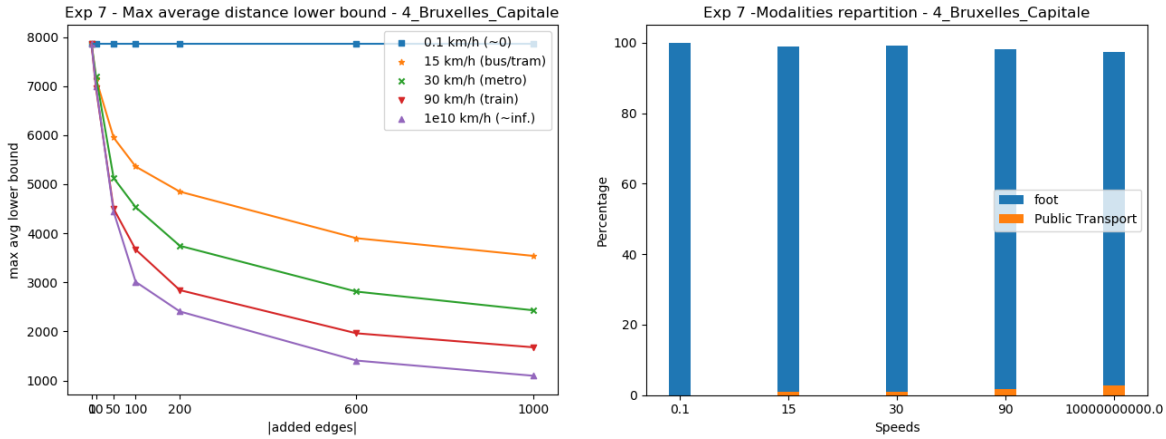



Figure 6.24: Exp 7 - **Left**: Maximum average lower bound, **Right**: Average modalities repartition

### 6.3.5 Multi-modality - Stations

Now, we will conduct some experiments on multi-modal station-based networks, namely a bi-modal network composed of:

- A base layer $\mathcal{L}_0$. This layer is composed of two possible edge weights :
  - Foot : with a walking speed of 5km/h as for the previous experiment.
  - Station-based shared car : in order to have a car base layer, we must assume there is a station at each node of the base layer.
- A second layer composed of bike-stations and in particular, we take the Villo-station multimodal network modeled in 5.3.2. Namely, 355 Villo stations in the area of Bruxelles-Capitale. For each station, we determine the closest node in the base layer, then we duplicate the base layer. The edge weights of the second layer are computed according to the average bike speed of 20km/h [8]. Finally, we add forward and backward transition edges in order to link the bike layer to the base layer.

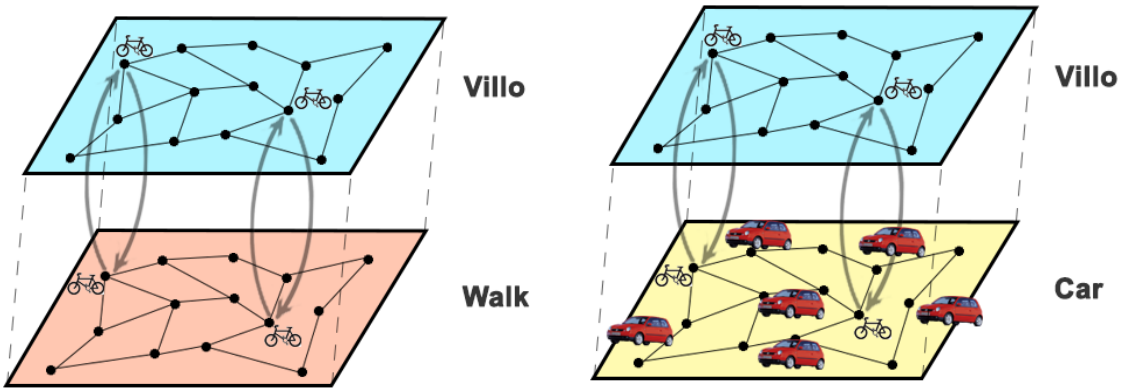



Figure 6.25: **Left**: multi-modal foot-villo network, **Right**: multi-modal shared car-villo network

[8] https://cyclinguphill.com/average-speeds-cycling/

**Experiment 8**

For the 8th experiment, we test Dijkstra's algorithm and ALT algorithm on the bi-modal bike-villo network like described hereabove. We make 2 assumptions regarding these graphs :

- The forward and backward transition edges are all set to 60, which represents the number of seconds it takes to switch between modalities. This number is arbitrary and can impact the performance of the algorithm.
- A shortest path can start in the base layer and end in another layer, meaning that it is not mandatory to go back to the base layer to close the shortest path as s-t pairs can belong to 2 different layers.

We first compare the preprocessing times on the multi-modal graph with the preprocessing times on the single-modal graph of experiment 5. Then, we evaluate the query times of Dijkstra and ALT as well as the improvements.

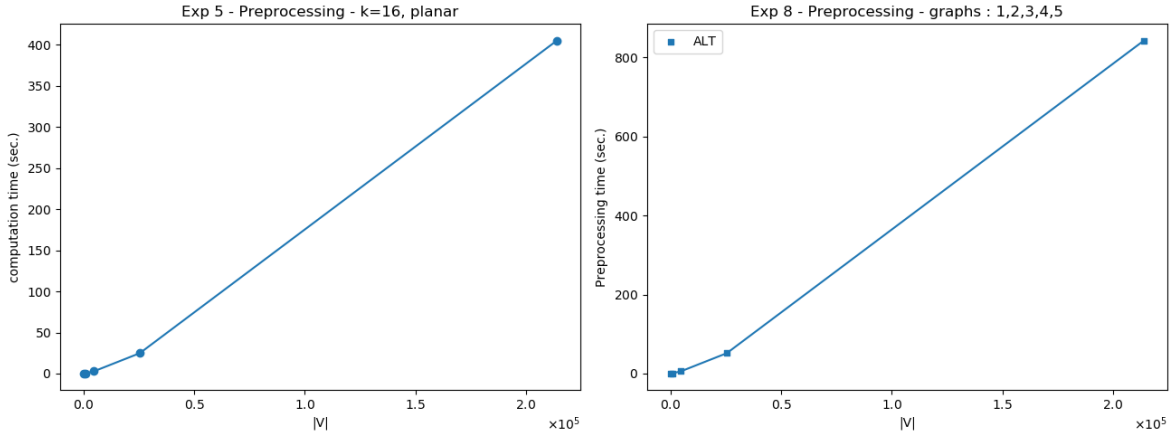



Figure 6.26: Exp 8 - **Left**: Preprocessing times single-modal network, graphs 1 to 6, **Right**: Preprocessing times station-based multimodal network, graphs 1 to 5

As we can see from fig 6.26, the preprocessing time on the multimodal networks is exactly the double of that of the single-modal networks (graphs 1 to 5). Hence, duplicating the graph yields doubled preprocessing times.

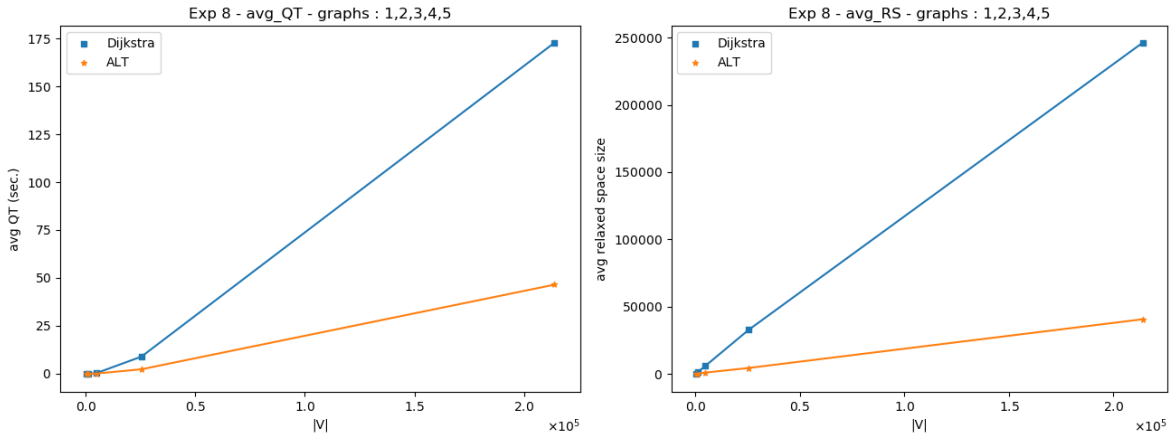



Figure 6.27: Exp 8 - **Left**: Average query times, **Right**: Average relaxed space

As expected, Dijkstra's algorithm (see fig 6.27) produces the same behaviour with the multi-modal networks as with the single-modal networks (i.e. nearly linear). ALT still manages to achieve a speed-up compared to Dijkstra. However, if we look at table 6.2 of experiment 6, ALT speedup for graph 5 was 8, which is the double.

Thus, ALT preprocessing is impacted by the change in graph topology (see fig 6.28 below) but proportionally to the graph size, although the query times are still reasonable considering the number of nodes and edges in the new graphs. Between graph 4 and 5, the speedup even decreases a little, which was not the case in the single-modal scenario of experiment 6. It is due to an increase in the number of relaxed edges in the search as the number of edges has more than doubled.

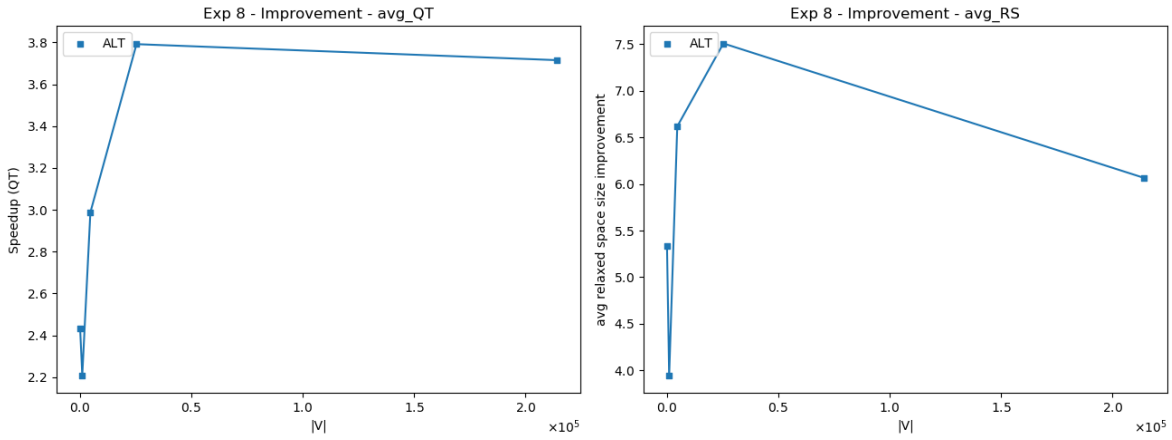



Figure 6.28: Exp 8 - **Left**: Average query times improvement over Dijkstra, **Right**: Average relaxed space improvement

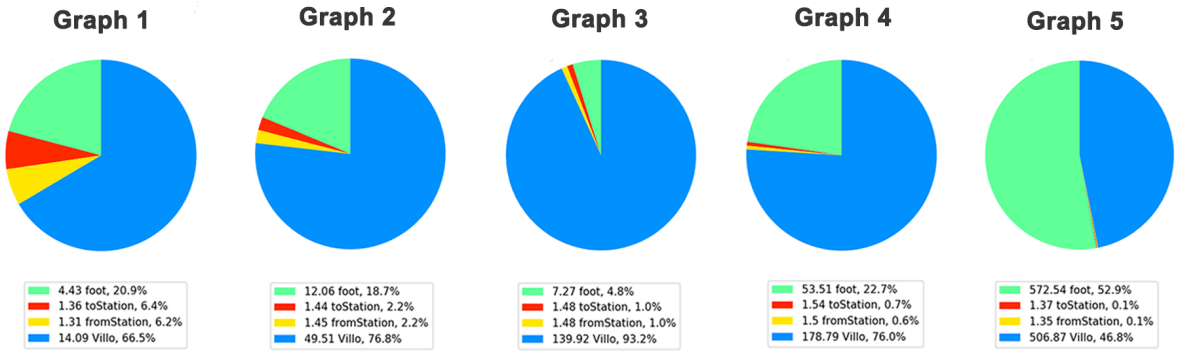



Figure 6.29: Exp 8 : average Modalities repartitions over graphs 1 to 5

fig 6.29 above exhibits the average modalities repartitions over graphs 1 to 5. That is, the average number of each modality in the shortest paths in the multimodal network. Since the bike-layer contains faster edges, ALT tends to choose the second layer more often. In graph 5, we see the opposite behaviour. It is explained by the fact that graph 5 covers a larger area than the Villo stations area, leading to more edges in the foot base layer. (see fig 6.3)

**Experiment 9 - Adding user preferences (part. 1)**

For the 9th experiment, we conduct some experiments on the multimodal network described before : a station-based shared-car base layer $\mathcal{L}_O$ and a second layer composed of villo-stations (see right side of fig 6.25). The particularity of this experiment is the addition of user preferences. We model them using the scalarized weighted sums (as described in section 5.4):

$$\hat{\omega}(e) = \sum_{i=1}^{\mathcal{M}} c_i \omega_i(e)$$

where $\mathcal{M} = 2$ and $\hat{\omega}(e) = c_1 \times Time(e) + c_2 \times Cost(e)$ :

- $\omega_1 = Time$ The traversal time : 3600 * length km / edge speed (depends on the layer).
- $\omega_2 = Cost$ The car gas price : length km / 100 * car consumption * gas price km. We use the following real values (chosen arbitrarily) :
  - The average gas consumption of a "Volkswagen Lupo" : 7 L / 100km. [9]
  - The average gasoline price in Belgium : 1.4 € / L [10]

For the preprocessing phase, we have applied ALT preprocessing on the multimodal network with constant preferences, namely $c_1 = c_2 = 1$ in order to have a neutral preprocessing (we will see in experiment 10 different preprocessing versions). We fix $c_1$ to 1 (associated to timing), and $c_2$ is variable. For both experiments 9 and 10 we use the following range: $[0, 2]$ with a step of 0.2 (we limit the lower bound to 0 to avoid negative edge weights). When $c_2 = 1$, $\omega_2$ remain the same. When $c_2 > 1$, $\omega_2$ increases (more expensive) and when $0 \leq c_2 < 1$, $\omega_2$ decreases (cheaper). user-adapted edge weights are computed on the fly during ALT's execution. We experiment these parameters on graphs 1 to 3, which are well covered by villo-stations (see fig 6.3). Each benchmark is averaged over **200** s-t paths.

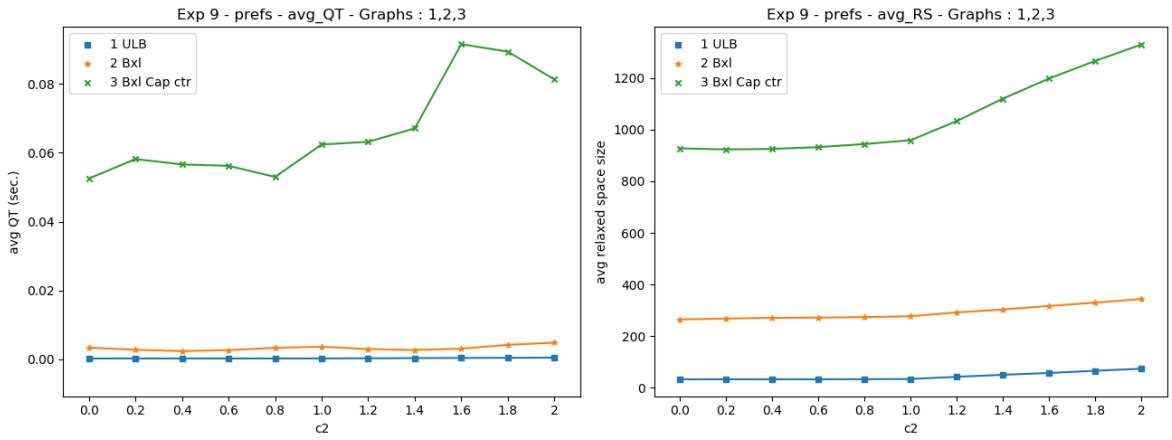



Figure 6.30: Exp 9 - **Left**: Average query times, **Right**: Average relaxed space

In terms of query times, we can see that it increases a little with $c_2$, especially for bigger graphs, which is correlated with the increase in average relaxed edge space size (right). Overall, if we

[9] http://www.fiches-auto.fr/articles-auto/chiffres-consommation/volkswagen.php

[10] https://bestat.statbel.fgov.be/bestat/api/views/74d181b1-7074-4c9f-9a71-85303980d41f/result/PDF

compare those query times with that of experiment 8, we do not see a significant reduction in performance (due to a potential overhead caused by the on-the-fly edge weights computations).

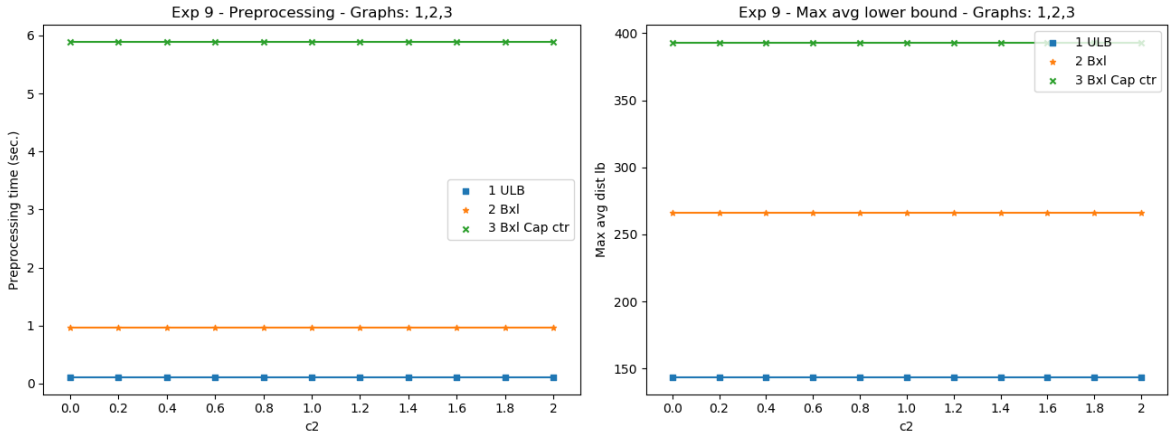



Figure 6.31: Exp 9 - **Left**: Preprocessing times, **Right**: Maximum average lower bound

In terms of preprocessing times, they are also equivalent to those of experiment 8 since we used $c_1 = c_2 = 1$. If we look at the maximum average lower bound in the right plot of fig 6.31, we can see that it does not change that much with the increasing values of $c_2$, which is expected since the landmark distances are not impacted by the user preferences during the preprocessing phase.

In figures 6.32 and 6.33 hereunder, we show the modalities repartition with respect to $c_2$ for each graph. That is, the average percentage of each modality constituting the average shortest path. We notice that the percentage of bikes involved in the shortest path decreases with the graph size. We suppose that the bigger the graph is, the more fast edges there are (varying speeds) and thus, the more interesting the car edges are. Increasing the average bike speed would considerably impact the behaviour of those plots as more bike edge would be considered in the shortest paths.

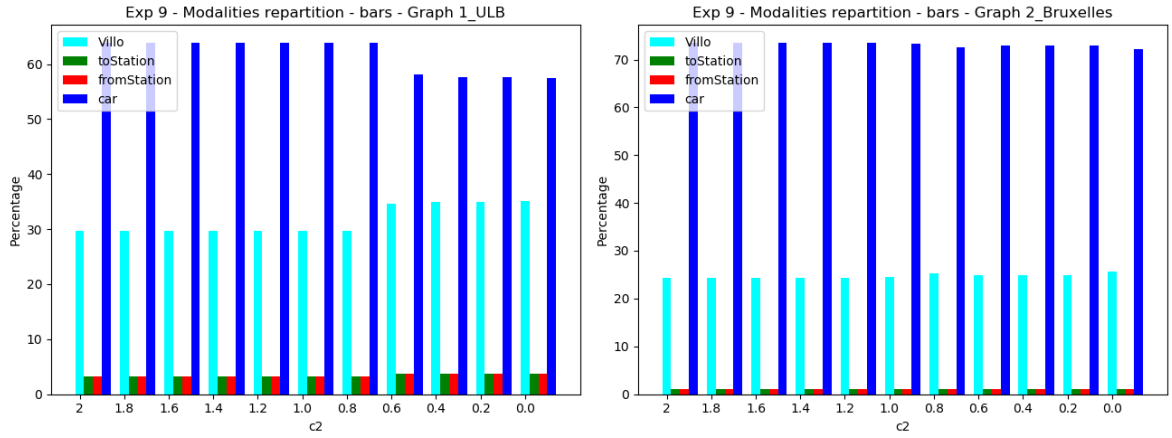



Figure 6.32: Exp 9 - Modalities repartition **Left**: graph 1, **Right**: graph 2

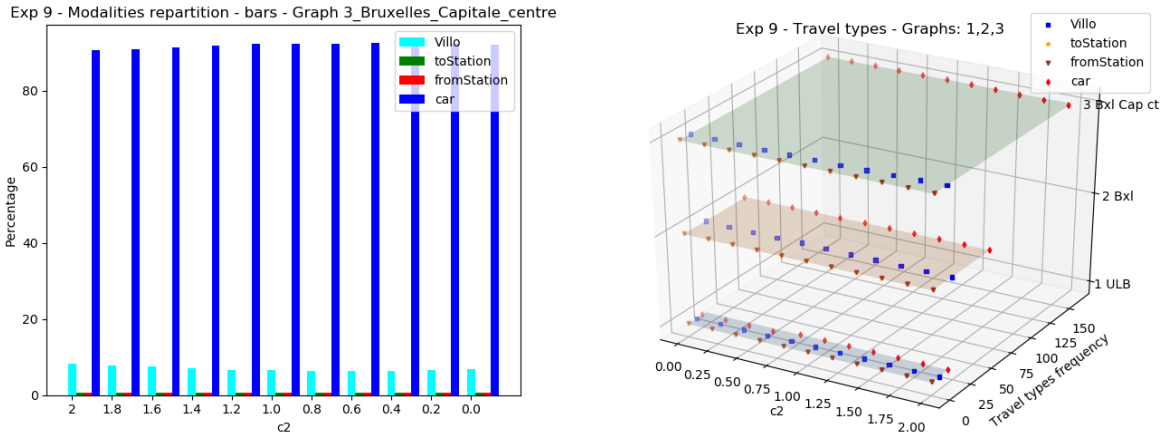



Figure 6.33: Exp 9 - Modalities repartition **Left**: graph 3, **Right**: Modalities frequencies - graphs 1 to 3 in 3D

Fig 6.33 (right), we have a global view on the modalities repartition over each graph (a combinations of figures 6.32 and 6.33 (left)). Each layer represents one graph. The size of the plane in terms of frequency ($y$ axis) is proportional to the average shortest path size within the graph.

### Experiment 10 - Adding user preferences (part. 2)

For the last experiment, we use the same parameters as for experiment 9, except that we only consider graph 3 (Bruxelles Capitale centre). During this experiment, we test different combinations. That is, we fix $c_1$ or $c_2$ and we perform a preprocessing with a worst-case scenario, namely the worst-case users. Given the preference range $[\mathbf{0}, \mathbf{2}]$ We have the 6 following combinations :

- Fix $c_1 = 1$ and preprocessing with $[1, \mathbf{0}]$ (*low*), $[1, \mathbf{2}]$ (*high*), $[1, \mathbf{1}]$ (*middle*),
- Fix $c_2 = 2$ and preprocessing with $[\mathbf{0}, 1]$ (*low*), $[\mathbf{2}, 1]$ (*high*), $[\mathbf{1}, 1]$ (*middle*).

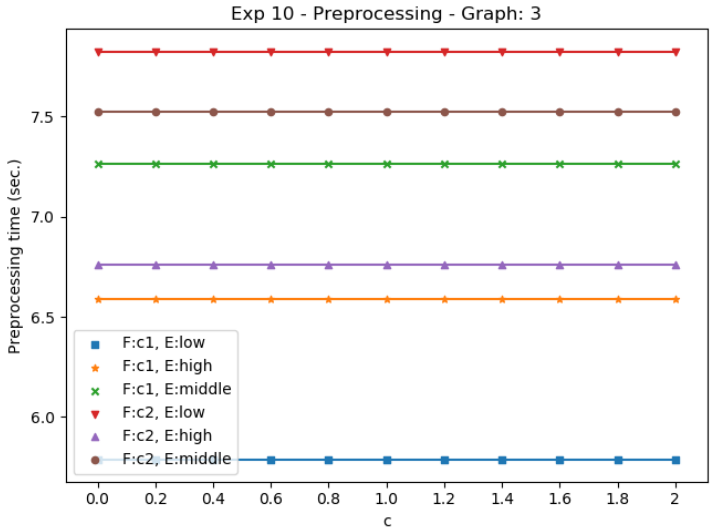



Figure 6.34: Exp 10 : Preprocessing times

In terms of preprocessing times, we can see that with the same graph, different combinations of preferences use cases affect the performance. In experiment 8, the preprocessing time for this exact graph was of 5.87737670000206 seconds. Thus, most of the preprocessing times with user preferences yield poorer preprocessing times, although the decrease in performance is quite low. Besides, we can notice that performing a preprocessing without worst-case users yields poorer results than with those preferences.

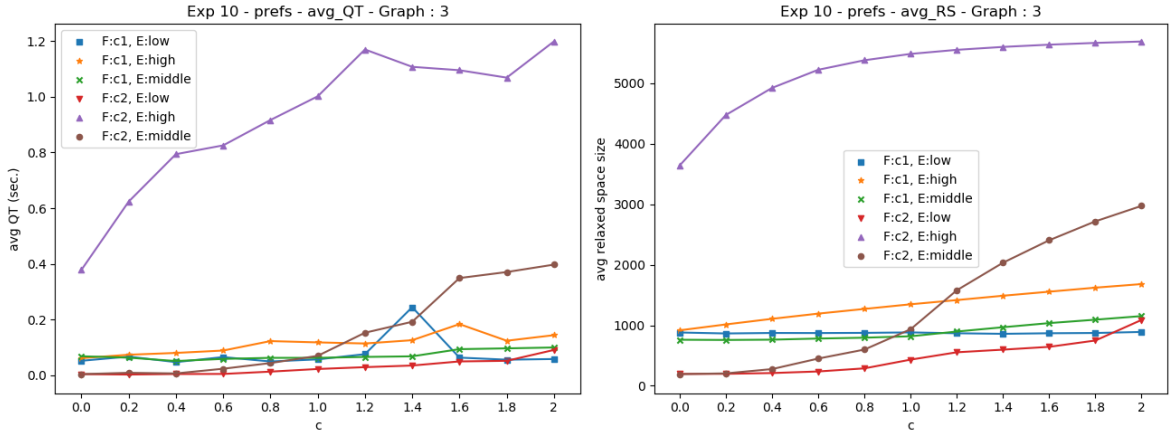



Figure 6.35: Exp 10 - **Left**: Average query times, **Right**: Average relaxed space

In terms of average query computation times, fig 6.35 exhibits interesting results. Except for "F:c2, E:high", most of the use cases yield stable query times over $c$. Also, using a preprocessing with middle case (neutral preprocessing) does not necessarily yield worse query times, except when $c > 1$. Overall, we see that the lines increase with $c$, which can be explained by the increasing average relaxed space size.

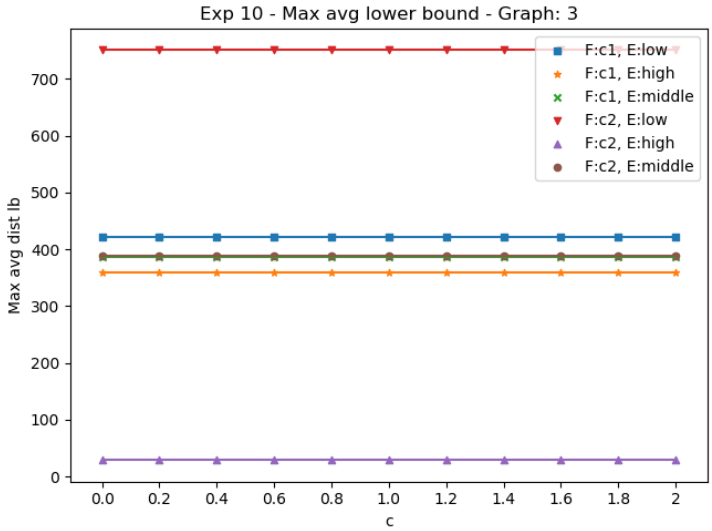



Figure 6.36: Exp 10 : Maximum average lower bounds

Fig 6.36 confirms the query times we obtained in fig 6.35. Indeed, "F:c2, E:high" produces the lowest lower bound, hence worst query times, while "F:c2, E:low" produces the highest lower bound, hence the best query times.

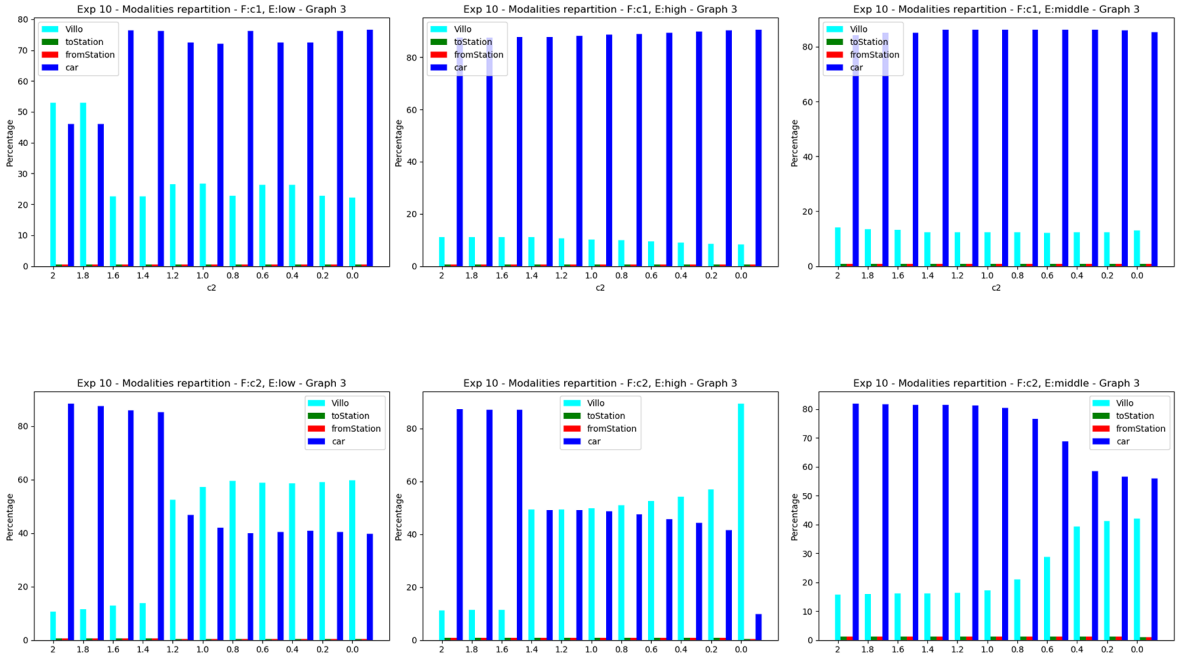



Figure 6.37: Exp 10 : Modalities frequencies

In fig 6.37, we have the average modalities repartition for each combination of preference use case. In the first row, $c_2$ is fixed, and in the second row, $c_1$ is fixed. When $c_1$ is fixed, only the price changes with $c_2$ and in this case, we can see that the more $c_2$ increases, the more bike edges appear in the average shortest path.

This can explain why "F:c2, E:high" performs the worst compared to the other combinations in fig 6.35. Indeed, fixing $c_2$ means that we fix the cost and vary $c_1$ (the time). Consequently, when considering the worst-case user with the highest value of the preference interval, the search has the incentive to pick more Villo edges as it has a cost of 0€ and increasing the time eventually yields higher edges values for the car layer.

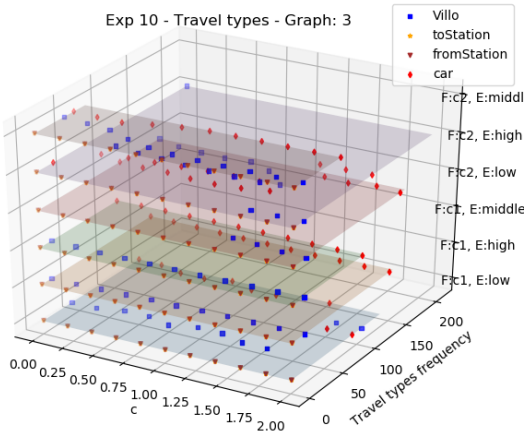



Figure 6.38: Exp 10 : Modalities frequencies - 3D

As in experiment 9, fig 6.38 shows the combinations of plots appearing in fig 6.37.

# Chapter 7

# Conclusion

We have analysed several prominent and promising speed-up techniques. Among them, we focused on Bidirectional search and more particularly on ALT (A*, landmarks and inequality), a heuristic-preprocessing-based speed-up technique. The latter offers a significant speed-up compared to Dijkstra's algorithm and we saw that it applies to multi-modal networks. In addition to introducing new difficulties, multi-modal routing allows the integration of user preferences which can either be modeled as multi-criteria problems or by simply changing edge weights within the graphs.

We conducted experiments on single-modal networks as well as on multi-modal networks. We also integrated user preferences in a multi-modal scenario. In particular, we have performed benchmarks constituted of a preprocessing phase and a query phase on multi-modal scenarios (namely 2) using ALT and analysed its performance with respect to Dijkstra's algorithm. First, we modeled a multi-modal network composed of a foot base layer and multiple public transport layers (6.3.4). We saw that ALT performance is impacted by a little with the number of added edges whilst the preprocessing times remain relatively reasonable. Then, we tested a bi-modal network composed of a foot base layer and a Villo-station-based layer. This model yields interesting results as the query times remain reasonable, but at the expense of doubled preprocessing times. Finally, we modeled a bi-modal network composed of a shared-car station-based base layer and a Villo-station-based layer. We integrated user preferences modeled with scalarized weighted sums and analysed ALT performance. We saw that overall, the overhead created by the preferences does not greatly impact the preprocessing and query times. Thus, it means that ALT remains applicable anyway without great loss in terms of performance. In conclusion, ALT is applicable to multi-modal networks and user-adapted bi-modal networks despite a really small drop in performance during the preprocessing phase.

This master thesis offers many possibilities for improvements and further work. As an example, we could apply ALT on time-dependent user-adapted multi-modal networks. We could also apply other speed-up techniques involving ALT, such as REAL (that we described in 4.5.2) in order to see if they behave similarly or not. Furthermore, we could model different multi-modal scenarios and combine them with user preferences (combination of station-based layers with public-transport layers for instance). We could apply other speed-up techniques on such networks. We saw in 4.5.3 that there exists a lot of such speed-up techniques and many of them are faster than ALT. Last but not least, we could implement the Label Correcting Dijkstra's algorithm (see 5.5.2) and compare it with the scalarized weighted sum variant.

# Appendix A

# Combinatorial Optimisation

## A.1 Integer program formulation

Since the shortest path problem consists in finding the shortest path, it is a combinatorial optimisation problem, where the optimal value we are searching is the shortest path regarding the configuration of the graph we are considering. Since all the quantifiable relationships in the problem are linear and the nodes are constrained in some way, we can use linear integer programming. Let us then define a linear integer programming formulation [78] of the classical shortest path problem using a directed graph.

We consider a directed graph $G = (V, A)$ with a set of vertices $V$ and a set of arcs $A$. Let $n$ and $m$ be the cardinalities of V and E respectively. A path in the graph is the sequence of vertices

$$P = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$$

which is elementary. A path is elementary if no vertex appears more than once in the graph. We denote $\delta^+(i)$ the set of outgoing arcs of vertex $i$ and $\delta^-(i)$ the set of incoming arcs of vertex $i$. $\delta^+(S)$ and $\delta^-(S)$ are the arcs leaving/entering the set $S \subseteq V$. We assume, without loss of generality, that $|\delta^-(s)| = |\delta^+(t)| = 0$. $s$ is the starting vertex of the shortest path we want to find and $t$ is the destination vertex. There is no arcs going into the starting node $s$ and no arcs going out of the destination node $t$.

The classical integral programming formulation to determine a shortest path from vertex $s$ to vertex $t$ can be written like this (primal form):

### A.1.1 Primal

---

$$\text{Minimize} \quad \sum_{(i,j)\in A}^{m} c_{ij}x_{ij}$$

$$\text{Subject to} \quad \sum_{(i,j)\in\delta^+(i)} x_{ij} - \sum_{(j,i)\in\delta^-(i)} x_{ji} = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{si } i = t \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i \in V$$

$$\sum_{(i,j)\in\delta^+(i)} x_{ij} \leq 1 \qquad \forall i \in V$$

$$x_{i,j} \in \{0, 1\} \qquad \forall (i,j) \in A$$

---

where $c_{ij} \in \mathbb{R}$ are the edges costs, and

$$x_{ij} = \begin{cases} 1 & \text{if the edge (i,j) is part of the path} \\ 0 & \text{otherwise.} \end{cases}$$

It is obvious that we want to minimize the total costs, which is the sum of all arcs's costs that constitute the path. The first constraint is the flow conservation constraint. It ensures that for each vertex only has one arc going into it and one arc going out of it, except for the source node $s$ (No arc is entering it) and for the destination node $t$ (No arc is going out of it). The second constraint ensures that the outgoing degree of each node is at most one (it is 0 for the destination node).

### A.1.2 Dual

We can formulate the dual version of this problem.

---

$$\text{Maximize} \quad \pi_t - \pi_s$$

$$\text{Subject to} \quad \pi_j - \pi_i \leq c_{ij} \quad \forall e = (i,j) \in A$$

$$\pi \in \mathbb{R}^{|V|}$$

---

where $\pi_i$ in this problem can be interpreted as being some sort of "distance" (a real value) of each node $i$ towards the starting node $s$. In this view, we can say that we want to put the

most distance between $s$ and $t$ with respect to the cost matrix $c$. To do that, we maximize the difference between the distance towards t and the distance towards s. In this case, $s$ is at distance 0 and when following an arc $(i, j)$, we have $\pi_j \leq \pi_i + c_{ij}$. An interpretation of this linear program is that the graph is being embedded on a line, with $\pi_i$ representing the position of the vertex $i$ on a line, subject to the constraints that for every edge (i,j), there is a "string" preventing vertex $j$ from being put more than $c_{i,j}$ units further along the line than vertex $i$. We can observe in particular that by linear programming duality, if $x_{ij} = 1$ (we choose to take arc (i,j) in the path of the primal), then by complementarity we must have

$$\pi_j = \pi_i + c_{ij}$$

Furthermore, note that we also need to satisfy all the other inequality constraints

$$\pi_j \leq \pi_k + c_{kj}$$

for the nodes that are not chosen in the corresponding primal path, which can only be satisfied when the distances correspond to at most the distance travelled in a shortest path.

### A.1.3 Property

In combinatorial optimisation, if we have a primal problem and its dual, we can have a weak duality or a strong duality.

Given a primal problem $P$

---

$$\text{Maximize} \quad c^T x$$

$$\text{Subject to} \quad Ax \leq b, \quad x \geq 0$$

---

and its dual problem $D$

---

$$\text{Minimize} \quad b^T y$$

$$\text{Subject to} \quad A^T y \geq c, \quad y \geq 0$$

---

The weak duality states [79] that the duality gap (the difference between the primal and the dual solutions) is always greater than or equal to 0. It means that the solution of the primal problem is always greater than or equal to the solution of the associated dual problem : $c^T x \leq b^T y$. So any feasible solution to the dual problem corresponds to an upper bound on any solution to the primal problem. If $(x_1, x_2, \ldots, x_n)$ is a feasible solution for the primal linear program and $(y_1, y_2, \ldots, y_n)$ is a feasible solution for the dual linear program, then the weak duality theorem states that

$$\sum_{j=1}^{n} c_j xj \leq \sum_{i=1}^{m} b_i y_i$$

where $c_j$ and $b_i$ are the coefficients of the objective functions, respectively. Generally, if $x$ is a feasible solution of the primal and $y$ is a feasible solution for the dual, then the weak duality implies $f(x) \leq g(y)$, where $f$ and $g$ are the objective functions of the primal and the dual, respectively.

The strong duality also satisfies the weak duality condition and the additionnal condition that states that the primal optimal objective and the dual optimal objective are equal.

Formally, if there exists an optimal solution $x'$ for $P$, then there exists an optimal solution $y'$ for $D$ and the value of $x'$ in $P$ equals the value of $y'$ in $D$.

This property is very useful and the shortest path problem actually satisfies the strong duality property. Denote by $\pi_1*$ an optimal solution to the dual problem and by $x*$ an optimal solution to the primal problem. Also denote by $v(\pi) = \pi_t$ and by $u(x) = \sum_{(i,j)\in A} c_{ij} x_{ij}$. We could prove strong duality by proving that

$$v(\pi_1*) = u(x*)$$

.

Since the shortest path problem satisfies the strong duality property, it is sufficient to find an optimal solution to the dual in order to obtain the optimal solution to the primal problem.

# Appendix B

# Miscellaneous

## B.1 Negative cycles is NP-Hard

We can show this by using the complementary problem : the `Longest path problem`. We first have to introduce some notions. A path with no repeated vertices is called `simple-path`. The longest-path problem is the problem of finding a simple path of maximum length in a given graph. This problem is NP-hard (it can be proved using a reduction from the Hamiltonian path problem [80]). Its decision problem can be formulated as asking whether a path exists of at least some given length. This decision problem is NP-Complete (all problems which are in NP can be reduced to this problem in polynomial time), and therefore not solvable in polynomial time unless the polynomial class P = NP. What is interesting is that the longest-path problem can be reduced to the shortest-path problem containing negative cycles to prove that it is NP-hard. Indeed, given a graph with only positive-weights, suppose we have a polynomial algorithm for solving our problem. If we negate all the edge-weights and run this algorithm, it would give the longest path in the original graph. Since the longest-path problem is NP-hard, it is not possible to obtain the longest path, and therefore the original problem is NP-hard. Now that we know that solving the shortest path problem in a graph containing negative weights is difficult, we will only consider non-negative weights graphs when using specific algorithms such as Dijkstra's algorithm (that will be described later).

## B.2 Contraction Hierarchies - Algorithm correctness

We can show that algorithm 13 is correct. Let us first show that any distance in the augmented graph is equal to the distance in the original graph after preprocessing phase.

**Lemma 10.** *[59] The distance $d^+(s,t)$ between any pair of nodes $s$ and $t$ in the augmented graph $G^+ = (V, E^+)$ is equal to the distance $d(s,t)$ in the original graph $G = (V, E)$ between the same nodes.*

*Proof.* Intuitively, since the shortcuts added in the augmented graph are combinations of edges in the original graph, their distance must be equal. Formally, first we know that $E$ is a subset of $E^+$, thus $d^+(s,t) \leq d(s,t)$. Some shortest path between $s$ and $t$ in $G$ will also be shortest path in $G^+$. Then, consider any added shortcut $(u,w)$ between $u$ and $w$. Before adding this shortcut, the original graph had the path $(u,v,w)$ of length $\omega(u,v)+\omega(v,w)$. The

added shortcut has length $\omega(u,v)+\omega(v,w)=\omega(u,w)$. It means that if there is some shortest path from $s$ to $t$ going through $(u,w)$ in $G^+$, then there is a path $(s,u,v,w,t$ in $G$ with same length. Thus, $d^+(s,t)$ cannot be shorter than $d(s,t)$, meaning that $d^+(s,t)=d(s,t)$. □

Now, we know that $G$ and $G^+$ are equivalent in terms of distance between nodes. We can now show that bidirectional search in the query phase is correct by proving that it is legitimate to find two increasing paths in forward and backward searches.

**Lemma 11.** *For any $s$ and $t$, the augmented graph $G^+=(V,E^+)$ contains a shortest path $P_{st}$ such that $P_{sv}+P_{vt}=P_{st}$ and $P_{sv}$ is increasing and $P_{v,t}$ is decreasing.*

*Proof.* We reason by contradiction. Let us assume that such $P_{st}$ described in lemma 11 does not exist. Then, it means that for any shortest path from $s$ to $t$, $P=(s,u_1,u_2,\ldots,u_k,t$, the decreasing order of nodes is not satisfied and there exists a node $u_i$ such that $r(u_{i-1})>r(u_i)<r(u_{i+1})$. $u_i$ is a local minimum. For any shortest path $P$ between $s$ and $t$, let us denote by $mr(P)$ the minimum rank of all local minima of $P$, which means that the local minimum will be contracted in priority. Among all shortest path, there must exist the shortest path $P^*$ with the maximum of the minimum ranks $mr(P)$. We then have $r(u_{k-1})>r(u_k)=mr(P)<r(u_{k+1})$ which lies in $P^*$. When we contract node $u_k$, we have 2 cases :

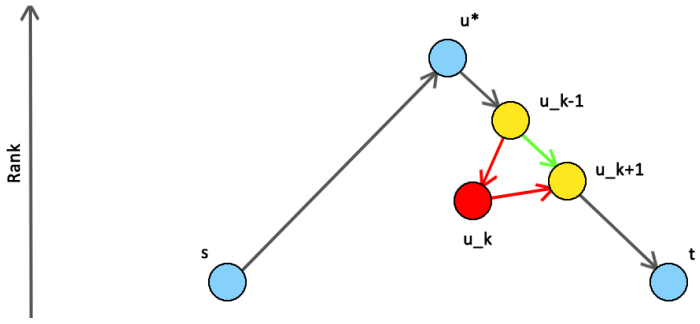



Figure B.1: Illustration of the proof

- If there is no witness path between $u_{k-1}$ and $u_{k+1}$, a shortcut $(u_{k-1},u_{k+1})$ is added and there is a shortest path $P'$ using this shortcut instead of the path $(u_{k-1},u_k,u_{k+1})$. Since $P'$ does not contain $u_k$, its minimum rank is necessarily bigger than that of $u_k$, which means that $mr(P')>mr(P^*)=r(u_k)$, which contradicts the initial hypothesis that $P^*$ has the maximum $mr(P)$.
- If there is a witness path between $u_{k-1}$ and $u_{k+1}$, then this witness path contains nodes with higher ranks than $r(u_k)$ and there exist a shortest path $P''$ which contains this witness path instead of path $(u_{k-1},u_k,u_{k+1})$ and since witness path's nodes have higher ranks than $r(u_k)$, we have that $mr(P'')>mr(P^*)$, which is also a contradiction.

□

# Appendix C

# Search algorithms

Another approach to the shortest path problem is as a search problem using agents. Indeed, an agent can establish goals that result in solving a problem by considering different sequences of actions that might achieve those goals. A goal is a desired state of the world. This goal can be thought as a set of world states in which it is satisfied or there could be multiple goals in the same world. The agent has the possibility to think about which action he can take in order to get him to the goal state. In our case, the world can be represented as a graph and the goal state is simply a path in this graph which is the shortest regarding the weights that constitute it. As said before, the goal could also be the shortest path from one node to all other nodes (single-source shortest path) or the shortest path from all nodes to one specific node (single-destination shortest path) or even the shortest path between all pairs of nodes in the graph (all-pairs shortest path).

**Algorithm 14** Agenda Search

```
1:  function AgendaSearch(start_state, actions, goal_test)        ▷ Returns an action sequence or failure
2:      seq ← ∅                                                  ▷ an action sequence, initially empty
3:      agenda ← ∅                                               ▷ a state sequence, initially contains start_state
4:      while True do
5:          if Empty(agenda) then
6:              return failure
7:          end if
8:          if goal_test(First(agenda)) then
9:              return seq
10:         end if
11:         agenda ← Queuing_Fn(agenda, Expand(First(agenda), actions))
12:         seq ← Append(seq, Action(First(agenda)))
13:     end while
14: end function
```

An action could be to pick up a node when searching for the shortest path in a graph. An agent must assess several possible sequences of actions leading to the goal state(s) and choose the best one. To do so, we can use search algorithms that take a problem as input, formulated as

a set of goals to be achieved and a set of possible actions to be applied and returns a sequence of action which is a solution. We need an initial state, which is the starting node(s), operators, which describe actions in terms of states reached by applying them, and a goal state, which is the destination node(s). The initial state and operators together define the `state space` of the problem which is the set of all states reachable from the initial state and any sequence of actions by applying the operators. We can describe a generic search algorithm framework [25] as follows (see agorithm 14) :

- `Queuing_Fn` is a function that determines what kind of search we are doing, `Expand` is a function that generates a set of states achieved by applying all possible actions to the given state and
- `First` is a function that gives the first element of a set of elements (action of state).
- The `goal test` function is a boolean function over the states that gives true if the input is the goal state and false otherwise.

We will describe several basic search algorithms that implement different `Queuing_Fn` functions and compare their performances in terms of complexity and capability to find the optimal solution of a given problem. The first three methods are `uninformed strategies`, i.e the way they expand their nodes is by only using information in the problem formulation. Then, there are `informed` or `heuristic search strategies` that use a quality measure not strictly in the problem formulation to generate solutions more effectively.

## C.1 Uninformed search

The order of node expansion defines a strategy. Uninformed strategies only use information in the problem formulation (initial state, operator, goal state, path cost). it can be seen as a blind search or a brute force search.

### C.1.1 Depth-first search (DFS)

The state space of the problem can be structured as a tree where actions correspond to arcs and states correspond to nodes. The Depth-first search **(Charles Pierre Trémaux, 1859–1882, [81] [82] )** [14] constructs the state space tree by going down and going across when a leaf is encountered. Thus, it starts at the root node and explores as deep as possible along each branch and backtracks when it is not possible anymore.

The problem is that the search can be impossible if the space contains cycles or if the space is generated from continuous values because the algorithm will indefinitly going down. However, to avoid this infinite depth due to cycles in the graph, we simply have to add a label to the nodes to explore. That way, we avoid exploring already visited nodes, which guarantees the algorithm to terminate for any finite graph. This change could potentially change the time complexity of the algorithm slightly. The depth-first search also depends on the order of the operators applications on nodes, since it could find wrong action sequences before good ones.

In the agenda search algorithm, the `Queuing_Fn` function would be implemented as a stack (Last in first out). It will remove the current node from the agenda and append its children to the front of the agenda. The algorithm terminates whenever it encounter the destination node during its search.

**Algorithm 15** Depth-first Search

```
1: function DFS_SEARCH(G, start_node, end_node)                    ▷ Returns a path
2:     S ← Stack()                                                  ▷ a stack
3:     S.push(start_node)
4:     explored ← ∅                                                 ▷ empty set
5:     while S not empty do
6:         c ← S.pop()
7:         if c == end_node then
8:             return path from explored nodes
9:         end if
10:        if c not labeled as explored then
11:            explored.add(c)
12:            for all edges from c to w in G.neighbours(c) do
13:                S.push(w)
14:            end for
15:        end if
16:    end while
17:    if q empty then return failure
18:    end if
19: end function
```

### C.1.2 Breadth-first search (BFS)

An alternative method to DFS is the Breadth-First Search **(Konrad Zuse, 1945 and Edward F. Moore, 1959, [83])**, which constructs the search tree by exploring its layers repeatedly and replacing each node by its children. This algorithm guarantees to find the solution if there exists one since it will explore all the nodes in the graph. It also guarantees to find the shortest action sequence. However, BFS can only be used to find the shortest path in a graph if :

- there is no loops. Like Depth-first search, BFS will find the shortest action sequence even if there are loops by avoiding already visited nodes.
- all edges have same weight or no weight at all. BFS do not consider the weights, so it finds the shortest action sequence which is equivalent to "shortest path" if all weights are equal.

If those two conditions are satisfied, then the algorithm just have to start from the initial node, apply the search and stop whenever it reaches the destination node. The shortest path found is the shortest in terms of number of edges in it. The problem is the tremendous amount of nodes that have to be explored if the search space becomes large, as it is usually the case in most real problems. In this case, the time needed to find the optimal sequence of actions is huge. In the agenda search algorithm, the `Queuing_Fn` function is implemented as a queue (First in first out). It will remove the current node from the agenda and append its children to the back of the agenda. At each step, each node of a given layer of the graph is being expanded.

**Algorithm 16** Breadth-first Search

```
1: function BFS_SEARCH(G, start_node, end_node)        ▷ Returns a solution or failure
2:     p ← Queue()                                                          ▷ a Queue
3:     p.push(start_node)
4:     explored ← ∅                                                      ▷ empty set
5:     while p not empty do
6:         c ← p.remove()
7:         if c == end_node then
8:             return path from explored nodes
9:         end if
10:        if c not labeled as explored then
11:            explored.add(c)
12:            for all edges from c to w in G.neighbours(c) do
13:                p.add(w)
14:            end for
15:        end if
16:    end while
17:    if q empty then return failure
18:    end if
19: end function
```

Note that BFS is equivalent to the Dijkstra's algorithm (described in 3.1.1) if all weights are equal (=1 for instance). The shortest path found by BFS is the shortest in term of number of edges and the shortest path found by Dijkstra is the shortest in term of the sum of edge weights.

### C.1.3 Iterative-deepening search

The compromise between Depth first search and Breadth first search is the Iterative-deepening search. It consists in applying DFS but with a depth bound (cutoff), forcing to behave like BFS. In the agenda search algorithm, we store the depth of each node at first (starting with a depth cutoff of 1), then only expand nodes with depths that are smaller to the depth cutoff. If it happened to be unsuccessful, we increment the depth and repeat the process again.

### C.1.4 Uniform-cost search (UCS)

The Uniform-cost search [84], also called Best-First Search (actually, best-first search is a class of algorithms and UCS is a special case.), constructs the search tree by choosing the best node in the agenda after each expansion, according to the utility function. This function is computing the utility of a route, it can be the distance between two nodes for instance. As for the Breadth-first search, UCS guarantees to find the solution if there exists one and also guarantees to find the optimal solution (the best sequence of actions) in terms of utility. Nevertheless, as BFS, it also suffers from the lack of efficiency when the search spaces become very large. It cannot guarantee to use less memory than BFS. In terms of agendas, `Queuing_Fn` function is a priority queue initialised with the source node. At a given point in the execution of the algorithm, a node is selected so that its cost is lower than the currently known shortest path's cost. Thus, it will extract the node with the lowest cost from the agenda and append its children to the agenda and then sort the whole agenda according to the utility function.

To avoid recomputation of the cost so far at each node, this value is also stored in the agenda. At the end, the algorithm will return the first optimal path encountered if the destination has been found, and will not search for other paths.

**Algorithm 17** Uniform-Cost Search

```
1:  function UFS_SEARCH(G, start_node, end_node)          ▷ Returns a path
2:      q ← priorityQueue()                               ▷ a priority queue
3:      q.push(start_node)
4:      explored ← ∅                                      ▷ empty set
5:      while q not empty do
6:          c ← s.extract_minimum()
7:          if c == end_node then
8:              return path from explored nodes
9:          end if
10:         if c not labeled as explored then
11:             explored.add(c)
12:             for all edges from c to w in G.neighbours(c) do q.insert(w)
13:             end for
14:         end if
15:     end while
16:     if q empty then return failure
17:     end if
18: end function
```

Note that the way nodes are ranked in the priority queue determine different kinds of variants to the best-first search algorithm. Namely, Breath-first search (described in C.1.2) is a special case of best-first search where the node's cost is defined by its depth in the search tree. Thus, UCS is a generalisation of BFS since the nodes's weight is the sum of the weights from the start node to the current node. In addition, it is worth to mention that UCS is very similar to Dijkstra's algorithm (3.1.1). Indeed, both has a priority queue and the cost function defined for each node is the sum of weights of the edges from the source node to another node along a path that is currently the shortest one known. As stated in [84], the only difference is that Dijkstra initialises its queue with all nodes of the graph, while UCS insert nodes progressively during the execution of the algorithm. This change is the advantage of UCS compared to Dijkstra. At any given time of the algorithm execution, the queue must contain a node with distance $\infty$. Thus, Dijkstra will never choose a node with distance $\infty$ for expansion. The execution time induced by the priority queue operations (insert, extract minimum or decrease priority) is greater for Dijkstra because it has to consider useless nodes in it, while UCS benefits from the small overhead caused by the queue size.

## C.2 Informed search

Informed search strategies rely on `heuristics`, a quality measure that is not in the problem formulation and that allows to guide the order in which nodes are expanded. The informed search methods give particular information about the state space in order to generate the solutions in a more efficient way. The heuristic describes the desirability of expanding nodes. It is an approximation since we cannot know in advance which node to expand in order to find

the best solution. Hence, unlike uninformed search algorithm which only consider information contained in the initial problem formulation, the heuristics involved in the informed search algorithm will help the algorithm to choose between several nodes. The most desirable nodes will be processed first, leading to the optimal solution.

### C.2.1 Hill-climbing search

Hill-climbing search, also called `Greedy search`, is the simplest kind of informed search. It uses a cost estimate from the current node to the solution. To solve the shortest path problem, this estimate is an heuristic that could be the straight-line distance to a given goal state and is based on the idea of geometrical distance estimates of distance in a graph (if the graph allows the notion of straight-line distance).

In terms of implementation, the agenda is sorted according to this heuristic and can be of any length. It is different from the Uniform Cost search (C.1.4) that uses the actual cost so far. The problem is that we do not take the cost so far into consideration, and that can lead to a sub-optimal solution. If we want the search to be as greedy as possible, we can use an agenda of size 1, where we only store the best node at each step. Every other node are thrown away. However, it can lead to a local minimum.

### C.2.2 A

The A algorithm is a combination of Uniform Cost search and Hill-Climbing search, described hereabove (C.1.4 and C.2.1). But now, it takes the cost so far (denoted $g(n)$) and the estimated cost remaining heuristic (denoted $h(n)$) into account. Together, it forms the overall utility function $f(n) = g(n) + h(n)$.

### C.2.3 A*

Like the A algorithm above, A* **(Peter Hart, Nils Nilsson and Bertram Raphael, 1968 [26])** [27] is an informed search algorithm and aims to find the optimal path from a specific starting node in a graph to a given goal state (single-pair). At each iteration of its main loop, it determines which of its node to extend by evaluating the cost of the path and a specific heuristic required to extend the path all the way to the goal. A*, as A, selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where $n$ is the next node on the path, $g(n)$ is the cost of the path from the start node to $n$, and $h(n)$ is a heuristic function that estimates the cost of the shortest path from $n$ to the goal state. A* ends when the path it chooses to extend is a path that contains the start and the goal state or if there are no eligible path to be extended anymore.

The difference with A algorithm is that now, A* uses a heuristic which is **admissible**, meaning that it never overestimates the costs to get to the goal. For example, given a node $B$, if the estimated cost is $X$ while the real cost to reach the goal state from $B$ is $Y < X$, then the heuristic is inadmissible because the heuristic $X$ over-estimates the real cost $Y$.

**Lemma 12.** *A* guarantees to return the optimal solution if the heuristic function is admissible.*

*Proof.* We assume that there is an optimal solution $G*$ with an optimal cost $f(G*) = C*$.

Suppose A* returns a non-optimal solution. It means that a non-optimal solution $G2$ must be in the agenda at some point of the algorithm. If $G2$ is non-optimal, it means that its cost function

$$f(G2) > f(G*) = C*$$

Further suppose that there is a node $v$ in the agenda which is on the path towards the optimal solution. If the heuristic $h(v)$ is admissible, then $h(v)$ does not overestimate the actual cost to the goal state, by definition. Therefore,

$$f(v) < C*$$

Now, if we combine the two formulas above, we obtain the following inequality :

$$f(v) < C* < f(G2)$$

Consequently, we can see that every node on the path towards the optimal solution has a cost that is less than the cost of the non-optimal solution $G2$. Hence, we conclude that $G2$ will never be chosen (expanded) from the priority queue during the algorithm execution. $G2$ will never be a solution returned by A*. Therefore, A* will always return an optimal solution if its heuristic $h$ is admissible. □

**Algorithm 18** A*

```
1: function aStar(start, goal, h)                          ▷ h is the heuristic function
2:     openSet ← {start}
3:     pred                                                ▷ pred[n] = node preceding the node n
4:     g ← ∅                                               ▷ g[n] = cost so far, default values set to Infinity
5:     g[start] ← 0                                        ▷ f[n] = g[n] + h(n) = utility function
6:     f ← ∅                                               ▷ default values set to Infinity
7:     f[start] ← h(start)
8:     while openSet ≠ ∅ do
9:         c ← node in openSet with lowest f value         ▷ c = current
10:        if c == goal then
11:            return reconstruct_path(pred, current)
12:        end if
13:        openSet.remove(c)
14:        for each neighbour n of current do
15:            n_g ← g[c] + 𝒲(c, n)      ▷ distance from start to neighbour through current
16:            if n_g < g[n] then
17:                pred[n] ← c
18:                g[n] ← n_g
19:                f[n] ← g[n] + h(n)
20:                if n ∉ openSet then
21:                    openSet.add(n)
22:                end if
23:            end if
24:        end for
25:    end while
26:    return failure                                      ▷ open set empty, goal not reached
27: end function
```

To implement that algorithm, we use a priority queue, which is called `open set`, in order to store the repeated selection of minimum cost nodes to expand. This open set can be implemented with a min-heap (binary heap), a priority queue or a hash set. At each iteration, we remove the node with the lowest $f(n)$ value from the queue and the $f$ and $g$ values of its neighbours are updated. these neighbours are added to the queue. The algorithms terminates whenever a goal node has a lower $f$ value than any node in the queue, or the queue is empty.

The solution found is the shortest path and its cost is the $f$ value of the goal state we end up with since the heuristic $h(t) = 0$ by definition of admissible heuristic. In order to get the sequence of actions leading to the shortest path, the algorithm should also keep track of all precedessors of each node. At the end of the execution, we could go back from the last node to the first node by following the predecessors using Algorithm 2.

An alternative approach would be to use a monotone, consistent heuristic. If the heuristic $h$ satisfies the additional condition

$$h(x) \leq \mathcal{W}(x, y) + h(y)$$

for every arcs $(x, y)$ of the graph, where the weight function $\mathcal{W}$ represents the distance between

nodes $x$ and $y$, then $h$ is "consistent". Note that this heuristic is equivalent to the definition of admissible heuristic. Indeed, given nodes $x$ and $y$, if the heuristic is consistent, it means that $x$'s heuristic must be lower than or equal to the distance from $x$ to $y$ added to $y$'s heuristic. Since $(x, y)$ is a directed edge from $x$ to $y$, $x$ has to reach node $y$ before reaching the goal state. Thus, the heuristic $h(x)$ do not overestimate the real cost from $x$ to the goal state assuming $y$'s heuristic is admissible.

The advantage of this heuristic is that A* guarantees to find an optimal solution without preprocessing any node more than once. This version of A* is equivalent to running Dijkstra's algorithm with the reduced cost

$$\mathcal{W}'(x, y) = \mathcal{W}(x, y) + h(y) - h(x)$$

In another point of view, Dijkstra's algorithm (3.1.1), as another example of uniform-cost search algorithm can be viewed as a special case of A* algorithm with a heuristic function $h(x) = 0$.

## C.3 Analysis

Now that we have an overview of the different algorithms, we can compare them in terms of performances. To do so, we can analyse their time and space complexity (we can find this analysis in [25]). However, there are several ways of defining the complexity of an algorithm. We focus here on the general complexity (time and space complexity in terms of number of nodes and edges) as well as the complexity induced by the specific graph topology. Each algorithm has been designed to tackle one or several aspect(s) of the shortest path problem (single-source, all-pairs, ...) and each one of them is dealing with the search space differently.

In order to compare those algorithms described in section B, we can assess these following criteria:

- *Completness* : If a solution exists for a given problem, the algorithm is always able to find a solution
- *Time complexity* : Can be measured :
  - in terms of execution time : how much time did the algorithm actually take
  - in terms of search space : number of nodes explored
- *Space complexity* : How much memory did the algorithm use (maximum number of nodes in memory at once)
- *Optimality* : The algorithm is always able to find the least-cost solution.

We can also assess the complexity of the algorithms with the following information about the search tree topology:

- the maximum *branching factor* of the search tree, denoted $b$ (if the tree is dense or sparse depending on the average number of successors per node)
- the *depth* of the optimal solution, denoted $d$
- the *maximum depth* of the state space, denoted $m$

In some cases, the complexity of an algorithm can not be easily characterized in terms of $b$ and $d$. Instead, the performance will be computed using *path costs* and *action costs* (costs of the operations in the algorithm).

### C.3.1 Depth-first search

- Depth-first search is not complete due to loops. Indeed, it fails to find the solution if there are loops in the graph because it is stuck in a infinite depth search space or if the search space is infinite. However, as stated in the Introduction (1) and in the algorithm description (C.1.1), we are assuming that our graphs are finite. We also avoided the problem of loops by introducing a simple additionnal avoid explored node condition in the algorithm. Therefore, we can consider the Depth-first search to be complete since it always terminates, and thus able to find a solution (not necessarily the optimal one) at some point, if there exists one.
- Its time complexity is $\mathcal{O}(b^m)$, which is huge if the maximum depth of the state space $m$ is larger than $d$. In practice, if there are a lot of possible solutions in the search space, DFS is faster than BFS since it has better chance of finding a solution with a small amount of nodes in the search space.
- It takes $\mathcal{O}(bm)$ in terms of space complexity. It only needs to keep track of one unique path from the root to the leaf node, along with unexpanded neighbours nodes for each node in the path.
- This algorithm is not optimal. It does not guarantee to always find the optimal solution.

### C.3.2 Breadth-first search

- Breadth-first search is complete. All nodes are examined if the branching factor $b$ is finite.
- Its time complexity is exponential in term of the branching factor $b$ :

$$1 + b + b^2 + b^3 + \ldots + b^d = \mathcal{O}(b^d)$$

  .
- Its space complexity is $\mathcal{O}(b^d)$ since it keeps track of every node in memory and that is an issue if the number of nodes is big.
- It is optimal only if the step cost equals 1. The step cost being the "edge" cost between two nodes, when an agent go from one state to another.

### C.3.3 Uniform-cost search

- As for the breadth-first search, UCS is complete and guarantees to find a solution if there is one.
- It runs in $\mathcal{O}(n)$ in the number of nodes with path cost less than the optimal solution. The time complexity of the algorithm can not be expressed using $b$ and $d$. We need the help of path cost rather than depth $d$. Suppose $C^*$ is the optimal path cost, and each

step cost is at least $\epsilon$, then the time complexity is

$$\mathcal{O}(b^{1+(C^*/\epsilon)})$$

where $(C^*/\epsilon)+1$ is the number of steps taken to reach the destination. The $+1$ is here because we start at distance 0 and end at $C/\epsilon$, so the steps are taken at distances

$$0, \epsilon, 2\epsilon, \ldots, (C^*/\epsilon)\epsilon$$

which is much greater than Breadth-first search if the step costs are not the same. Indeed, Uniform-cost search is the same as BFS when all step costs are identical.

- Its space complexity is identical to its time complexity. Namely, $\mathcal{O}(n)$ in the number of nodes with path cost less than the optimal solution. In another point of view, we can express the space complexity using path cost. In the algorithm, the node with smallest cost is removed at each step, and its successors are added. We need to store all explored nodes until the goal node is found since we need the least cost of all of these nodes. The space complexity is therefore $\mathcal{O}(b^{1+(C*/\epsilon)})$.
- It is optimal since it always find the optimal solution as long as the step cost is positive.

### C.3.4 Iterative-deepening search

- As this algorithm is a Depth-first search but with depth bound, its behaviour is closed to Breadth-first search. Therefore, this algorithm share the same performances as the Breadth-first search algorithm.
- It is complete
- It runs in $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = \mathcal{O}(b^d)$ time, which is asymptotically the same as breadth-first search.
- At any given time, the algorithm is performing a depth-first search, and never searches deeper than depth $d$. Thus, this algorithm benefits from the same advantage (and is even more efficient) as DFS which is the modest memory requirements due to the depth of the paths in the graph and it actually runs in $\mathcal{O}(d)$ space. It means that iterative-deepening search simulates Breadth-first search with only linear space complexity. [85]
- It is optimal if the step cost equals 1

### C.3.5 A*

- A* is complete and guarantees to terminate on finite graphs with non negative edge weights, it will always find a solution if one exists. Even though we are not working with infinite graphs, it is worth to mention that if the graph is inifinite with a finite branching factor $b$ and edges costs satisfying

$$\mathcal{W}(x,y) >_{\epsilon} 0$$

for a fixed $\epsilon$, then A* guarantees to terminate if there is a solution.

- The time complexity of A* depends on the chosen heuristic. If the heuristic function is well chosen, it can have a great impact on the performances of the algorithm since

it prevent A* from expanding many of the $b^d$ nodes that an uninformed search would have expanded. To measure the efficiency induced by a given heuristic (in addition to the search space), we can measure the *effective* branching factor $b^*$, which can be calculated empirically by measuring the number of expanded nodes, $N$, and the depth of the solution $d$.

$$N = 1 + b* + (b^*)^2 + \ldots + (b^*)^d$$

For example, if A* finds a goal at depth 5 using 52 nodes, the effective branching factor is 1.91. A good heuristic is characterised by a low *effective* branching factor. The optimal branching factor is 1 ($b^* = 1$). In the worst case, A* runs in $\mathcal{O}(b^d)$ time, which is exponential in the depth of the solution when the search space is unbounded. If the goal state is not reachable from the starting node and the state space is infinite, then A* will never terminate. But it can terminate with a "failure" output if the goal state is unreachable and the graph is finite. If we do not look at the worst case scenario of A* algorithm, its time complexity would be polynomial if the search space is a tree and the goal state is reachable. One additional condition for the complexity to be polynomial is that the heuristic function satisfies :

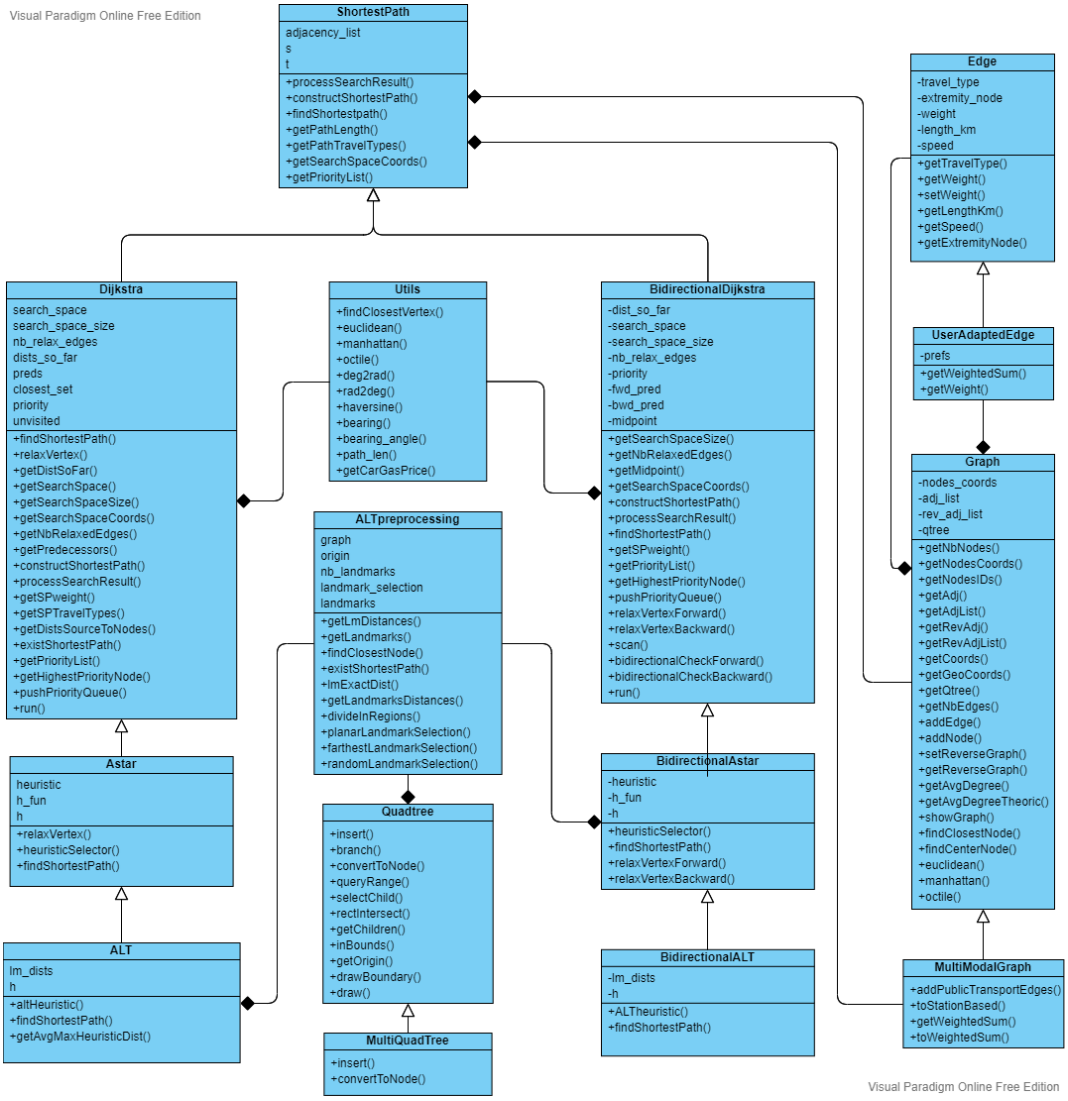
$$|h(x) - h(x^*)| = \mathcal{O}(\log h^*(x))$$

where $h^*$ is the optimal heuristic, meaning the exact cost to reach the goal state from a node x. That formula means that the error of the heuristic $h$ will not grow faster than the logarithm of the optimal heuristic $h^*$. It gives us a bound on the time complexity of the algorithm.

- Concerning the space complexity, it is similar to the other search algorithms since it stores all generated nodes in memory. Thus, the worst-case space complexity is $\mathcal{O}(b^d)$
- A* is optimal as long as the heuristic is admissible, which is supposed in the algorithm description (it never overestimates the costs to get to the goal state).

# Appendix D

# Class diagram



# Bibliography


[1] Paul E. Black. *Shortest Path, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/shortestpath.html. ed. 15 July 2019.

[2] Paul E. Black. *single-source shortest-path problem, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/singleSourceShortestPath.html. ed. 1 February 2005.

[3] Paul E. Black. *single-destination shortest-path problem, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/singleDestShortestPath.html. ed. 17 December 2004.

[4] Paul E. Black. *all pairs shortest path, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/allPairsShortestPath.html. ed. 1 February 2005.

[5] Victor Goossens. "Contraction Hierarchies and its applicability to Multi-Modal Routing". MA thesis. Université Libre de Bruxelles (ULB), 2019.

[6] Ittai Abraham Amos Fiat Andrew V. Goldberg Renato F. Werneck. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms". In: . (2010). URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2010/01/soda10.pdf.

[7] Ittai Abraham Daniel Delling Andrew V. Goldberg Renato F. Werneck. "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks". In: . (2010).

[8] Paul E. Black. *sublinear time algorithm, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/sublinearTimeAlgo.html. ed. 3 September 2019.

[9] Paul E. Black. *Path, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/path.html. ed. 24 August 2017.

[10] S. Gill Bender Edward A.; Williamson. *Lists, Decisions and Graphs. With an Introduction to Probabillity.* ., 2010.

[11] Paul E. Black. *simple path, in Dictionary of Algorithms and Data Structures [online].* URL: https://www.nist.gov/dads/HTML/simplepath.html. ed. 24 August 2017.

[12] Christopher M. Dellin and Siddhartha S. Srinivasa. "A Unifying Formalism for Shortest Path Problems with Expensive Edge Evaluations via Lazy Best-First Search over Paths with Edge Selectors". In: (2016).

[13] E. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[14] Cormen Thomas H. Leiserson Charles E. Rivest Ronald L. Stein Clifford. *Introduction to Algorithms (Second edition).* MIT Press and McGraw Hill, 2001. ISBN: 0-262-03293-7.

[15] M. L. Fredman and R. E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the Association for Computing Machinery* (1987).

[16] Peter Melhlhorn Hurt; Sanders. *Algorithms and Data Structures. The Basic Toolbox.* Springer, 2008. ISBN: 978-3-540-77977-3.
[17] R. Bellman. "ON A ROUTING PROBLEM". In: *Quarterly of Applied Mathematics* 16 (1958), pp. 87–90.
[18] Michael J. Bannister and David Eppstein. "Randomized Speedup of the Bellman–Ford Algorithm". In: *Computer Science Department, University of California, Irvine* (2011).
[19] R. W. Floyd. "Algorithm 97: Shortest path". In: *Communications of the ACM* 5 (1962), p. 345.
[20] S. Warshall. "A Theorem on Boolean Matrices". In: *J. ACM* 9 (1962), pp. 11–12.
[21] Stefan Hougardy. "The Floyd-Warshall algorithm on graphs with negative cycles". In: *esearch Institute for Discrete Mathematics, University of Bonn, Lenn eestr. 2, 53113 Bonn,Germany* (2010).
[22] Paul E. Black. *Floyd-Warshall algorithm, in Dictionary of Algorithms and Data Structures [online].* URL: `https://www.nist.gov/dads/HTML/floydWarshall.html`. ed. 17 December 2004.
[23] Donald B. Johnson. "Efficient Algorithms for Shortest Paths in Sparse Networks". In: *Journal of the ACM (JACM)* 24 (1977), pp. 1–13.
[24] Paul E. Black. *Johnson's algorithm, in Dictionary of Algorithms and Data Structures [online].* URL: `https://www.nist.gov/dads/HTML/johnsonsAlgorithm.html`. ed. 17 December 2004.
[25] Stuart J. Russell. *Artificial Intelligence: A Modern Approach (2nd ed.)* Upper Saddle River, New Jersey, Prentice Hall, 2003. ISBN: 0-13-790395-2.
[26] P. Hart, N. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Trans. Syst. Sci. Cybern.* 4 (1968), pp. 100–107.
[27] Zeng W. Church R. L. "Finding shortest paths on real road networks : the case of A*". In: *International Journal of Geographical Information Science* (2009).
[28] Paul E. Black. *Manhattan distance, in Dictionary of Algorithms and Data Structures [online].* URL: `https://www.nist.gov/dads/HTML/manhattanDistance.html`. ed. 11 February 2019.
[29] Daniel D. Peter S. Domink S. Dorothea W. "Engineering Route Planning Algorithms". In: *Universitat Karlsuhe, Germany* (2009).
[30] Parmanand Singh. "The So-called Fibonacci numbers in ancient and medieval India". In: *Historia Mathematica* (1985).
[31] Stein C. Fredman M. Sedgewick R. "The Pairing heap: A New Form of Self-Adjusting Heap". In: (1986).
[32] Mikkel Thorup. "Undirected Single Source Shortest Paths with positive integer weights in linear time". In: *Journal of the ACM* (1999).
[33] H. Bast. "Car or Public Transport - Two Worlds". In: *Efficient Algorithms.* 2009.
[34] I. Pohl. "Bi-directional and heuristic search in path problems". In: *Stanford University* (1969).
[35] A. V. Goldberg. "Point-to-point Shortest Path Algorithms with Preprocessing". In: *Microsoft Research - Silicon Valley* (2007).
[36] Karinthy Frigyes. "Chain-Links". In: (1929).
[37] Lenie de Champeaux Dennis; Sint. "An improved bidirectional heuristic search algorithm". In: *journal of the ACM* (1977).
[38] A. V. Goldberg C. Harrelson. "Commputing the Shortest Path : A* Search Meets Graph Theory". In: *Microsoft Research, Microsoft Corporation* (2003).

[39] Paul E. Black. *Euclidean distance, in Dictionary of Algorithms and Data Structures [online].* URL: `https://www.nist.gov/dads/HTML/euclidndstnc.html`. ed. 17 December 2004.

[40] Paul E. Black. *triangle inequality, in Dictionary of Algorithms and Data Structures [online], Algorithms and Theory of Computation Handbook, CRC Press LLC.* URL: `https://www.nist.gov/dads/HTML/trianglnqlty.html`. 1999.

[41] Richard E. Korf. "Recent Progress in the Design and Analysis of Admissible Heuristic Functions". In: *Computer Science Department, University of California, Los Angeles Los Angeles, CA 90095* (2000).

[42] T. Ikeda Min-Yao Hsu H. Imai S. Nishimura H. Shimoura T. Hashimoto K. Tenmoku and K. Mitoh. "A Fast Algorithm for Finding Better Routes by AI Search Techniques". In: *Proc. Vehicle Navigation and Information Systems Conference. IEEE* (1994).

[43] W. Pijls H. Post. "Bidirectional A*: Comparing balanced and symmetric heuristic methods". In: *Econometric Institute Report EL2006-41* (2006).

[44] Fabian Fuchs. "On Preprocessing the ALT-Algorithm". MA thesis. 2010.

[45] Bauer R. Columbus T. Katz B. Krug M. Wagner D. "Preprocessing speed-up techniques is hard." In: *International Conference on Algorithms and Complexity* (2010).

[46] Ulr ich Lauther. "An extremely fast, exact algorithm for finding shor test paths in static networks with geographical background". In: (2006).

[47] R. Möhring et al. "Partitioning Graphs to Speed Up Dijkstra's Algorithm". In: *WEA*. 2005.

[48] G. V. Batz. "Time-Dependent Route Planning with Contraction Hierarchies". In: 2014.

[49] R. J. Gutman. "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks". In: *ALENEX/ANALC*. 2004.

[50] P. Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries". In: *ESA*. 2005.

[51] P. Sanders and Dominik Schultes. "Engineering highway hierarchies". In: *ACM J. Exp. Algorithmics* 17 (2012).

[52] Reinhard Bauer, D. Delling, and D. Wagner. "Experimental study of speed up techniques for timetable information systems". In: *Networks* 57 (2007), pp. 38–52.

[53] Dominik Schultes and Peter Sanders. "Dynamic Highway-Node Routing". In: *Universitat Karlsruhe, Germany* (2007).

[54] H. Bast et al. "Fast Routing in Road Networks with Transit Nodes". In: *Science* 316 (2007), pp. 566–566.

[55] H. Blast D. Delling A. Goldberg M. M. Hannemann T. Pajor P. Sanders D. Wagner R. F. Werneck. "Route Planning in Transportation Networks". In: *University of Freiburg, Sunnyvale USA, Amazon USA, Martin-Luther Universitat Halle-Wittenberg Germany, Microsoft Research, Karlsruhe Institute of Technology* (2015).

[56] Julian Arz, Dennis Luxen, and P. Sanders. "Transit Node Routing Reconsidered". In: *ArXiv* abs/1302.5611 (2013).

[57] Dominik Schultes. "Route Planning in Road Networks". In: *Ausgezeichnete Informatikdissertationen*. 2008.

[58] Robert Geusberger Peter Sanders Christian Vetter. "Exact routing in large road networks using contraction hierarchies". In: *Transportation Science* (2012).

[59] R. Geisberger P. Sanders D. Schultes D. Delling. "Contraction Hierarchies : Faster and Simpler Hierarchical Routing in Road Networks". In: *Universitat Karlsruhe Germany* (2008).

[60] B. Reinhard D. Delling P. Sanders D. Schieferdecker D. Schultes D. Wagner. "Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm". In: *Institut fur Theoretische Informatik (ITI)* (2008).

[61] Daniel Delling Andrew V. Goldberg Thomas Pajor Renato F. Werneck. "Robust Exact Distance Queries on Massive Networks". In: *Technical Report MSR-TR-2014-12* (2014).

[62] D. Delling et al. "Highway Hierarchies Star". In: *The Shortest Path Problem*. 2006.

[63] Reinhard Bauer and D. Delling. "SHARC: Fast and robust unidirectional routing". In: *ACM J. Exp. Algorithmics* 14 (2010).

[64] A. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Efficient Point-to-Point Shortest Path Algorithms". In: *ALENEX*. 2006.

[65] Reinhard Bauer et al. "Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm". In: *ACM J. Exp. Algorithmics* 15 (2010).

[66] Christian Sommer. "Shortest-path queries in static networks". In: *ACM Computing Surveys (CSUR)* 46 (2014), pp. 1–31.

[67] Thomas Pajor. "Multi-Modal Route Planning". MA thesis. Fakultät für Informatik, Mar. 2009.

[68] Yue Lu et al. "A Fuzzy Intercontinental Road-Rail Multimodal Routing Model With Time and Train Capacity Uncertainty and Fuzzy Programming Approaches". In: *IEEE Access* 8 (2020), pp. 27532–27548.

[69] Manuel Braun. "Multi-Modal Route Planning with Transfer Patterns". In: 2012.

[70] Jianwei Zhang et al. "A multimodal transport network model for advanced traveler information systems". English. In: *Procedia : Social and Behavioral Sciences* 20 (Sept. 2011). 14th Meeting of the Euro Working Group on Transp. - In Quest for Adv. Models, Tools, and Methods for Transp. and Logist., EWGT, 26th Mini-EURO Conf. - Intelligent Decis. Making in Transp. and Logist., MEC, 1st Eur. Sci. Conf. on Air Transp. - RH ; Conference date: 06-09-2011 Through 09-09-2011, pp. 313–322. ISSN: 1877-0428. DOI: 10.1016/j.sbspro.2011.08.037.

[71] Kerstin Dächert, Jochen Gorski, and Kathrin Klamroth. "An augmented weighted Tchebycheff method with adaptively chosen parameters for discrete bicriteria optimization problems". English. In: *Computers and Operations Research* 39.12 (2012), pp. 2929–2943. DOI: 10.1016/j.cor.2012.02.021.

[72] E. Jafari and S. Boyles. "Multicriteria Stochastic Shortest Path Problem for Electric Vehicles". In: *Networks and Spatial Economics* 17 (2017), pp. 1043–1070.

[73] L. Galand, P. Perny, and Olivier Spanjaard. "Choquet-based optimisation in multiobjective shortest path and spanning tree problems". In: *Eur. J. Oper. Res.* 204 (2010), pp. 303–315.

[74] I. Ibraheem and Fatin Hassan Ajeil. "Multi-Objective Path Planning of an Autonomous Mobile Robot in Static and Dynamic Environments using a Hybrid PSO-MFB Optimisation Algorithm". In: *Appl. Soft Comput.* 89 (2020), p. 106076.

[75] Dominik Kirchler, Leo Liberti, and R. W. Calvo. "A Label Correcting Algorithm for the Shortest Path Problem on a Multi-modal Route Network". In: *SEA*. 2012.

[76] F. Guerriero and R. Musmanno. "Label Correcting Methods to Solve Multicriteria Shortest Path Problems". In: *Journal of Optimization Theory and Applications* 111 (2001), pp. 589–613.

[77] Reinhard Bauer et al. *Preprocessing Speed-Up Techniques is Hard*. Tech. rep. 4. Karlsruher Institut für Technologie (KIT), 2010. 13 pp. DOI: 10.5445/IR/1000016080.

[78] Leonardo Taccari. “Integer programming formulations for the elementary shortest path problem”. In: . (2015). URL: http://www.optimization-online.org/DB_FILE/2014/09/4560.pdf.

[79] Teofilo F. Gonzalez. *Handbook of Approximation Algorithms and Metaheuristics*. CRC Press, 2007. ISBN: 9781420010749.

[80] Yushi Uno Ryuhei Uehara. “Efficient Algorithms for the Longest Path Problem”. In: . (2004).

[81] S. Even. “Graph Algorithms”. In: 1979.

[82] Robert Sedgewick. “Algorithmen in C++ - Teile 1 - 4, Grundlagen, Datenstrukturen, Sortieren, Suchen (3. Aufl.)” In: 2002.

[83] E. F. Moore. “The shortest path through a maze”. In: *Harvard University Press, Cambridge* (1957).

[84] Ariel Felner. “Position paper: Dijkstra’s Algorithm versus Uniform Cost Search or a Case Against Dijkstra’s Algorithm”. In: *Information Systems Engineering. Ben-Gurion University.* (2011).

[85] Richard E. Korf. “Depth-First Iterative-Deepening: An Optimal Admissible Tree Search”. In: *Department of Computer science, Columbia University* (1985).