POLITECNICO DI TORINO

Master degree course in Mechatronic Engineering

Master Degree Thesis

Goal recognition design

Artificial intelligence techniques in surveillance problems

Advisors
prof. Fabio Fagnani

prof.ssa Sara Bernardini

Candidate
Alessandra Lo Piano Rametta

October 2019

Summary

This thesis focuses on Goal Recognition, an innovative and still developing field of research in Artificial Intelligence, which involves determining an agent's goal by observing its behaviour. Several real-world applications can be modelled in this context, for example, surveillance problems, operations where it is necessary to preserve user privacy, chat-boxes, and human-robot interaction.

In this dissertation, we consider an environment in which an intelligent agent, the target, performs actions that can be observed by another intelligent agent, the observer (or by the environment itself). We analyze an evasive or fleeing target that moves in the environment to reach its destination, while the observer tries to find it. An example of this scenario is a surveillance drone that flies over a wide geographical area to discover a criminal who is trying to reach a hideout by car as soon as possible. In this context, the observer runs Goal Recognition algorithms that aim to discover where the target is going. In particular, the observer needs to reason about the target's strategy to obscure its goals. To support this kind of reasoning, we propose a technique to calculate a new measure, the "undisclosing index", that represents the maximal length of the prefix of a path that a fleeing agent may take before its goal becomes apparent to the observer. Moreover, we present methods for the observer to reason about how the target might change its behaviour based on the resources that it has available: the target will take shorter or longer paths based on its total travel budget.

We modelled the environment in which the agents move by using a connected graph. For our experiments, we construct the graphs based on data from a real web map provided by OpenStreetMap (OSM). We implemented a web application designed for this purpose. Through it, the user selects and exports an area of interest on the map, identifying the target's possible goals within that area. The web-app provides a viewable version of the graph, implements the algorithms that help the observer reason about the possible paths of the target based on its budget and visualize them.

The experimental evaluation shows how the solutions proposed in this thesis are useful tools for solving Goal Recognition problems.

Acknowledgements

Vorrei ringraziare innanzitutto i relatori di questa tesi, il professor Fabio Fagnani e la professoressa Sara Bernardini, per il tempo, la disponibilità e l'estrema gentilezza riservatimi durante questi mesi di lavoro, oltre che per avermi dato l'opportunità di lavorare presso un'università straniera e in un settore tanto all'avanguardia.

Ringrazio mia madre, che ha sempre creduto in me, insegnandomi che determinazione, grinta e passione non sono mai abbastanza nella vita. Ringrazio mio padre, che con la sua saggezza e pazienza ha saputo tirar fuori il meglio di me, guidandomi durante tutto il percorso accademico. Ringrazio Francesco, mio fratello, che con la sua semplicità e genuinità è riuscito a strapparmi un sorriso in ogni momento di difficoltà.

Ringrazio le persone splendide con cui ho condiviso le mie giornate in questi cinque anni, Manuela e Miriam, la mia piccola famiglia torinese. Un pensiero va anche alla mia cara amica Valeria, che da sempre mi dimostra che anche chi è lontano può essere vicino, e alla mia compagna di viaggio Isabella, per la sua gentilezza e disponibilità, oltre che per il suo essere fonte inestinguibile di sapere.

Grazie infine a tutte le persone che non ho menzionato in queste pagine, ma che hanno avuto un ruolo importante in questo mio percorso.

Contents

1 Introduction 1
- 1.1 Problem Description and Motivation . . . 1
- 1.2 Thesis Aims . . . 1
- 1.3 Organization . . . 2

2 Literature Review 3
- 2.1 Introduction . . . 3
- 2.2 Graph theory . . . 3
  - 2.2.1 Graphs: terminology involving Paths and Cycles . . . 5
  - 2.2.2 Graph: connectivity . . . 6
  - 2.2.3 Graph: main classification . . . 6
  - 2.2.4 Data Structures for Representing Graphs . . . 10
- 2.3 Shortest path problems . . . 12
  - 2.3.1 Dijkstra's algorithm . . . 13
  - 2.3.2 Yen's algorithm . . . 22

3 Goal Recognition 31
- 3.1 Introduction . . . 31
- 3.2 Plan Recognition . . . 31
- 3.3 Goal Recognition and Goal Recognition Design . . . 34

4 Statement of the problem 37
- 4.1 Introduction . . . 37
- 4.2 Abstract formulation of goal recognition . . . 37
- 4.3 Optimization of the undisclosing index . . . 40
  - 4.3.1 Maximum undisclosing index achievable within a set of paths . . . 41

5 Experimental results 53
- 5.1 Graphical user interface design . . . 53
  - 5.1.1 XML file . . . 54
  - 5.1.2 Preprocessing . . . 56
  - 5.1.3 Processing . . . 60
  - 5.1.4 First Test . . . 63
- 5.2 Results, precision and speed . . . 69

6 Conclusion 75
- 6.1 Conclusion and Future Work . . . 75

Appendices 77
.1 Tested Areas . . . 79
.1.1 Map and data: second tested area . . . 79
.1.2 Map and data: third tested area . . . 80
.2 Tables of Tests . . . 81
.2.1 Results: first data set . . . 81
.2.2 Results: second data set . . . 84
.2.3 Results: third data set . . . 90
.3 Computational Time . . . 96

Bibliography 97

Chapter 1

Introduction

1.1 Problem Description and Motivation

Nowadays, intelligent systems are used in a wide variety of applications: there are several realities, more or less consolidated, that make use of them. These intelligent agents are technologically advanced entities that perceive and respond to the world around them: practically, they analyse the surrounding environment, therefore decide on rational actions to achieve their objectives.

In Artificial Intelligence, the specific task of determining an agent's goal by observing its behaviour is called Goal Recognition. The need to identify strategies and goals of a system in a given environment is a very recurrent task in several real-world applications such as surveillance problems, chat-boxes, operations where it is necessary to preserve user privacy, human-robot interaction, etc.

In this dissertation, we consider an environment in which an intelligent agent, the target, performs its movement and actions that can be observed by another intelligent agent, the observer (or by the environment itself). In particular, the target is a fleeing agent, i.e. it moves in the environment to reach its destination and achieve its final goal, while the observer tries to find it. An example of this scenario is a surveillance drone that flies over a wide geographical area to discover a criminal who is trying to reach a hideout by car as soon as possible.

In this context, Goal Recognition techniques allow developing strategies aimed at discovering where the target is going. In particular, the observer runs Goal Recognition algorithm to reasons about the target's strategy to hide its final destination and how this strategy changes based on its available resources (the target will take shorter or longer paths based on its total travel budget).

1.2 Thesis Aims

The purpose of this dissertation is to understand how the actions that an agent can perform in the environment reveal its final goal. In other words, through this work, we aim to devise techniques that allow the creation of an intelligent action plan.

In our specific scenario, the observer, through planning techniques, tries to reason about target's possible plans and strategies, in order to increase its chances of finding it. For this purpose, in this dissertation has been proposed a technique to calculate a new measure, the *undisclosing index*, that represents the maximal length of the prefix of a path that a fleeing agent may take before its goal becomes apparent to the observer. The observer gets this new measure reasoning about how the target might change its behaviour based on the resources that it has available: the target will take shorter or longer paths based on its total travel budget.

In this sense, a relevant goal of this dissertation is to maximize the *undisclosing index*, using a new algorithm based on the well-known Dijkstra and Yen's algorithms, and trying to modify the model of the environment in which the agents move.

We modelled the environment in which the agents move by using a connected graph. For our experiments, we built the graphs starting from real data provided by OpenStreetMap (OSM). We implemented a web application designed for this purpose. Through it, the user selects and exports an area of interest on the map, identifying the target's possible goals within that area. The web-app provides a viewable version of the graph, implements the algorithms that help the observer reason about the possible paths of the target based on its budget and visualize them.

The experimental evaluation at the end of the dissertation shows how the solutions proposed and the algorithm implemented are useful tools for solving related Goal Recognition problems. The results obtained could also constitute the starting point for research in the field of Goal Recognition Design, where the ultimate goal is a real modification of the environment to simplify or hinder the Goal Recognition.

1.3 Organization

The thesis is organized as follows. We start by providing some general concepts about Graph theory and associated terminology and algorithms. We continue by introducing the necessary background to understand the Goal Recognition theory: therefore, we will provide some concepts related to it such as Plan Recognition and Goal Recognition Design. Then we propose a concrete instance of the Goal Recognition problem, considering the scenario mentioned above and analysing it as a precise mathematical optimization problem. We conclude with practical applications of the implemented algorithms and empirical evaluations of the obtained results, then some considerations about related and future works.

Chapter 2

Literature Review

2.1 Introduction

This chapter aims to provide and explain some of the concepts underlying the research carried out by this thesis work. In addition, the terminology and notation used in this work are provided and fixed. First of all, let's start with an introduction to Graph theory.

2.2 Graph theory

Graph theory was introduced by the mathematician Leonhard Euler as a possible solution to the problem called "Seven Bridges of Königsberg". The problem was to look for a path through the city that crossed each bridge once and only once. Starting from the considered problem, a new discipline of mathematics was born and evolved, aimed at modelling and solving decision-making problems. Reconnecting to the aforementioned problem, a graph represents, in mathematical terms, a complex network of the real world.

According to 'A Textbook of Graph Theory' by R. Balakrishnan and K. Ranganathan [1], we provide the following definition. A graph $\mathcal{G}$ is defined as a ordered triple $(\mathcal{V}(\mathcal{G}), \mathcal{E}(\mathcal{G}), \mathcal{I}_{\mathcal{G}})$, where $(\mathcal{V}(\mathcal{G}))$ represents a nonempty set of elements called vertices, nodes or points, $\mathcal{E}(\mathcal{G})$ is the set of edges, or lines, connecting the nodes, and $\mathcal{I}_{\mathcal{G}}$ is an "incidence" relation that associates each edges in $\mathcal{E}(\mathcal{G})$ to an unordered pair of nodes in $\mathcal{V}(\mathcal{G})$.

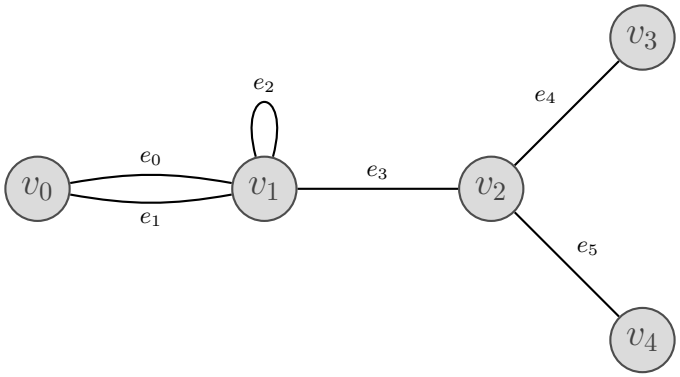
Below is a clarifying example:

Example 2.2.1. Let $\mathcal{V}(\mathcal{G}) = \{v_0, v_1, v_2, v_3, v_4\}$, $\mathcal{E}(\mathcal{G}) = \{e_0, e_1, e_2, e_3, e_4, e_5\}$. According to the above concepts, the set of relationships could be given by $I_{\mathcal{G}}(e_0) = \{v_0, v_1\}$, $I_{\mathcal{G}}(e_1) = \{v_0, v_1\}$, $I_{\mathcal{G}}(e_2) = \{v_1, v_1\}$, $I_{\mathcal{G}}(e_3) = \{v_1, v_2\}$, $I_{\mathcal{G}}(e_4) = \{v_2, v_3\}$, $I_{\mathcal{G}}(e_5) = \{v_2, v_4\}$. So the triple $(\mathcal{V}(\mathcal{G}), \mathcal{E}(\mathcal{G}), \mathcal{I}_{\mathcal{G}})$ is a graph.

For convenience, we will express the triple $(\mathcal{V}(\mathcal{G}), \mathcal{E}(\mathcal{G}), \mathcal{I}_{\mathcal{G}})$ in the more compact form $(\mathcal{V}(\mathcal{G}), \mathcal{E}(\mathcal{G}), \mathcal{I}_{\mathcal{G}})$.

Reconnecting to the previous example, it is possible to provide a graphical representation, or more simply, a planar diagram of the graph considered:

Figure 2.1. Graphical representation of the graph described by the example 2.2.1



In the Figure 2.1 it is possible to distinguish the nodes, with the respective labels $(v_1, v_2, ..., v_4)$, and the edges, $(e_1, e_2, ..., e_5)$, or the lines connecting the vertices to each other. Notice that in a graphical representation of a graph, two or more edges could intersect each other: this insertion does not necessarily identify a node of the aforementioned graph.

We now provide some notions related to the cardinality of nodes and edges of a graph. The number of vertices in a graph $\mathcal{G}$ is defined as its *order*. It is devoted by $|\mathcal{V}|$. We will use n to denote the order of $\mathcal{G}$. The number of edges, on the other hand, is called *size* of $\mathcal{G}$ and denoted by $|\mathcal{E}|$. We will use m to denote the size of $\mathcal{G}$.

Depending on the order, a graph can be $finite$ or $infinite$. For simplicity, in this work, we will always refer to finite graphs. A graph of order 0 or 1 is defined *trivial*. In addition, note that a graph with no vertex or edges is defined *empty*.

A graph with many edges with respect to the number of vertices n is called *dense*, whereas a graph with a few edges is called *sparse*; in general we can say that a graph is $sparse$ if the number of edges is of the same order of magnitude as the number of vertices:

$$\mathrm{m} = \mathrm{O}\ (\mathrm{n})$$

At this point, it is necessary to provide some important theoretical concepts related to nodes and edges in a graph. Below is a basic concept of *end* of an edge.

For a given edge e^*, given the related $I_{\mathcal{G}}(e^*) = \{v_1^*, v_2^*\}$, we define v_1^* and v_2^* as the *ends* or *endpoints* of the considered edge e^*. Each edge *joins* its ends, as well as it is possible to say that each ends is *incident* with its edge. The graphic representation provided in Figure 2.1 and the concept of *end* just explained allow us to give other simple definitions related to Graph Theory.

Two or more edges are called *multiple* or *parallel* if they have the same couple of distinct ends. A trivial example of parallel edges is shown in Figure 2.1, and consists of the set of edges $\{e_0, e_1\}$.

If the ends of an edge consist of the same node, the edge is called *loop* at the considered node. An obvious example is represented by the edge e_2 (Figure 2.1).

Concerning these last definitions, it is possible to state that a graph in which there are no loops or multiple edges is called *simple graph*.

Now consider the concept of *adjacency*, first applied to the nodes of a graph, subsequently to its edges. Suppose we work, for convenience, with undirected graphs (For more details see subsection 2.2.3). Given two vertices v_1^*, $v_2^* \in \mathcal{V}$, they are defined *adjacent* if there is at least one edge $e^* \in \mathcal{E}$ joining them, i.e. the related $\mathcal{I}_{\mathcal{G}}(e^*)$ exist and it is worth that $\mathcal{I}_{\mathcal{G}}(e^*) = \{v_1^*, v_2^*\}$.

Differently, two or more distinct edges are called *adjacent* if they have a common endpoint. Therefore, given for example the vertices v_1^*, $v_2^*, v_3^* \in \mathcal{V}$, the connecting edges $e_1^*, e_2^* \in \mathcal{E}$, with

$e_1^* \neq e_2^*$, and the related $\mathcal{I}_\mathcal{G}(e_1^*)$, $\mathcal{I}_\mathcal{G}(e_2^*)$, if $\mathcal{I}_\mathcal{G}(e_1^*) = \{v_1^*, v_3^*\}$ and $\mathcal{I}_\mathcal{G}(e_2^*) = \{v_2^*, v_3^*\}$ then the two considered edge e_1^*, e_2^* are *adjacent*.

Two or more vertices of a graph which are adjacent are called *neighbours*. The set of all neighbours of a vertex v^* is called *neighborhood set* of v^*. It is identified by the compact form $N(v^*)$ or $N[v^*]$. The first one represents the *open neighborhood set*, i.e.:

$$N(v^*) = \{u^* \in \mathcal{V} | u^* \text{ adjacent to } v^* \text{ and } u^* \neq v^*\}$$

The second expression is the representation of the *closed neighborhood set*, that is:

$$N[v^*] = N(v^*) \cup \{v^*\}$$

Having defined the concept of neighborhood set, consider the following statement. Given a (non-empty) graph, we define *degree* or *valency* of a vertex v^* the number of edges at v^*. In other words, the degree $d_\mathcal{G}(v^*)$ of a node is equal to the number of its neighbours (this last definition is not valid for multigraphs). A vertex whose degree is zero is called *isolated*. From a notation point of view, we can indicate with $\delta(G) = min\{d_\mathcal{G}(v) | v \in \mathcal{V}\}$ the minimum degree of a graph $\mathcal{G}$. The maximum degree of $\mathcal{G}$ will be expressed by $\Delta(G) = max\{d_\mathcal{G}(v) | v \in \mathcal{V}\}$. In the particular case in which each node has the same valency, then $\mathcal{G}$ is *regular*.

2.2.1 Graphs: terminology involving Paths and Cycles

First of all, we provide the definition of *walk* in a graph $\mathcal{G}$. Trivially, a walk can be seen as a sequence of nodes and edges whose ends are vertices. Truly, a walk is also a sequence of adjacent two-by-two edges. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a walk can be seen as

$$(v_0, e_0, ...v_{k-1}, e_k, v_k)$$

with $\mathcal{I}_\mathcal{G}(e_i) = v_i, v_{i+1}$, for $1 \leq i \leq k$. The number of edges in a *walk* is called *length* of the walk. For convenience, the edges within the sequence will be omitted, so that a walk will appear as follows:

$$(v_0, v_1, ..., v_k)$$

Note that v_0 and v_k are the *origin* and *termination* of the walk respectively. Moreover, if $v_0 = v_k$ the walk is said *closed*, otherwise is *open*.

A *path* p is a particular case of *walk*: in this case all the nodes are different. The considerations previously written and relating to the concept of *walk* are still valid. In addiction, a path of length k can be denoted by p^k.

Figure 2.2. Graphical representation of a path p^3 in a graph $\mathcal{G}$

The concept of path, as widely used in this thesis work, will be repeatedly taken up and expanded in the following sections.

Now we define the concept of *subpath*: given a generic path

$$p = (v_0, v_1, v_3, v_4)$$

let's call sub_p of p the following sequence of elements:

$$sub\_p = (v_1, v_3, v_4)$$

In other words, the subpath sub_p of a path p is an ordered sequence of nodes also contained in p, then:

$$sub\_p \subseteq p$$

A path of length greater than or equal to 3 is a *cycle* if its origin and termination node coincide. A cycle is a type of sequence as shown below:

$$(v_0, v_1, ..., v_{k-1}, v_0)$$

A cycle is said *even* or *odd* according to its length (trivially, if its length is an even number, then the cycle is *even*, otherwise is *odd*).

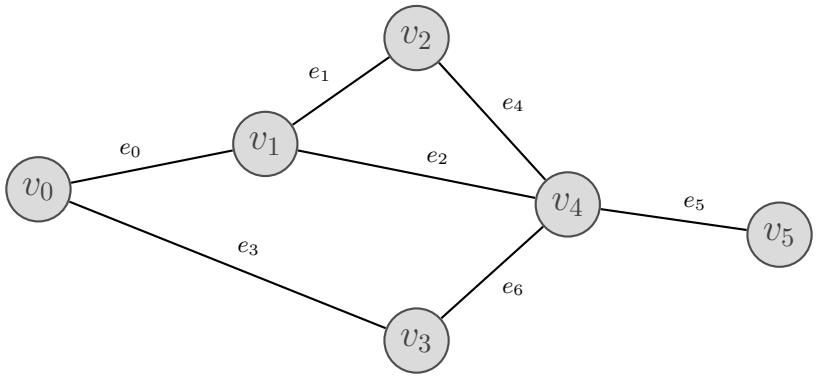
2.2.2 Graph: connectivity

A non-empty graph $\mathcal{G}$ is defined *connected* if every couple of distinct nodes of $\mathcal{G}$ are joined by a path. Otherwise the graph is said *disconnected*. More specifically, a graph $\mathcal{G}$ is called $k-connected$ if there are no vertices in $\mathcal{G}$ separated by fewer than k other nodes. Truly, each graph is at least $0-connected$ (unless it is an empty graph). In this regard we define the connectivity $\kappa(\mathcal{G})$ of a graph: it is the greatest value k such that the graph is $k-connected$, then if the graph is disconnected the connectivity is null. The concept of connection will be taken up in the next chapter, as essential for the formulation of the problem dealt with by this thesis (See chapter 4, section 4.2).

2.2.3 Graph: main classification

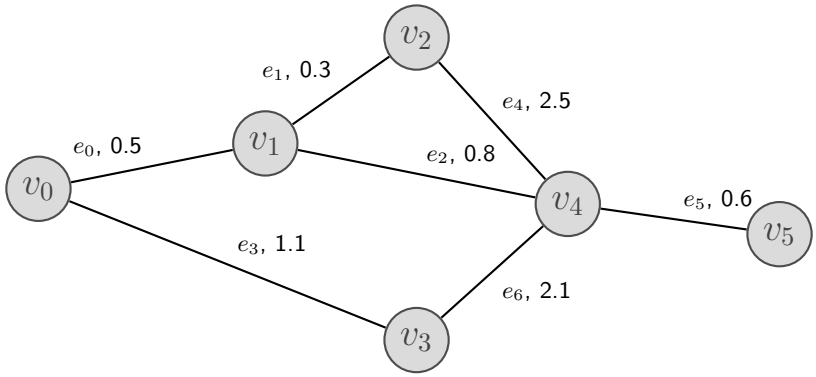
The following are the main classes to which a graph can belong. According to the type of its edges, a graph can be defined *unweighted* or *weighted*. In particular, given two nodes $v_1, v_2 \in \mathcal{V}$, we define the graph as *unweighted*, if the edge connecting the two nodes can exist or not. The type of edge in question is called *unweighted* or *binary*: as for a switch, its value can be 1 (on), if the edge exists, or 0 (off) if it doesn't exist. Notice that for a unweighted graph there is no criterion of importance among the various edges. Below in Figure 2.3 is an example of an unweighted graph:

Figure 2.3. Unweighted Graph



On the other side a graph is *weighted*, if we can attribute to the edge connecting two nodes $v_1, v_2 \in \mathcal{V}$ a real number w_{v_1,v_2}, called *weight* of the edge.

Figure 2.4. Weighted Graph



The figure 2.4 represents a weighted graph: on each edge it is possible to see the weight of the considered edge next to its label. It is also possible to use a more compact form to represent the weights of the graph: let's talk about a weight matrix W. The weight matrix related to the considered example is shown in figure 2.5.

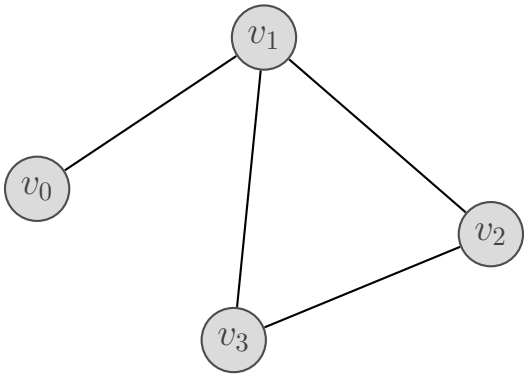
Figure 2.5. Weight matrix W related to graph in Fig. 2.4

$$W = \begin{pmatrix} w_{v_0,v_0} & w_{v_0,v_1} & \cdot & \cdot & \cdot & w_{v_0,v_5} \\ w_{v_1,v_0} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ w_{v_0,v_5} & \cdot & \cdot & \cdot & \cdot & w_{v_5,v_5} \end{pmatrix} =$$

$$= \begin{pmatrix} 0 & 0.5 & 0 & 1.1 & 0 & 0 \\ 0.5 & 0 & 0.3 & 0 & 0.8 & 0 \\ 0 & 0.3 & 0 & 0 & 2.5 & 0 \\ 1.1 & 0 & 0 & 0 & 2.1 & 0 \\ 0 & 0.8 & 2.5 & 0.8 & 0 & 0.6 \\ 0 & 0 & 0 & 0 & 0.6 & 0 \end{pmatrix}$$

Another distinction must be made between *direct* and *undirected* graphs. So far, for convenience, we have considered undirected graphs, since much of the classical theory of the graphs has been developed for this type of structure. Specifically, a graph is defined *undirected* if each of its edges is associated with an unordered pair of vertices in $\mathcal{V}$. From a more practical point of view, given two nodes $v_1, v_2 \in \mathcal{V}$ connected by two edges e_1, e_2, according to the relations $I_{\mathcal{G}}(e_1) = \{v_1, v_2\}$ and $I_{\mathcal{G}}(e_2) = \{v_2, v_1\}$, there is no privileged orientation and $I_{\mathcal{G}}(e_1) = I_{\mathcal{G}}(e_2)$. Below is an example of a undirected graph (Fig. 2.6).

Figure 2.6. Undirected Graph



On the other hand, a graph is said *directed* if the elements of $\mathcal{I}_\mathcal{G}$ are ordered pairs of nodes: considering the previous case in which we have v_1, v_2, e_1, e_2, for a directed graph

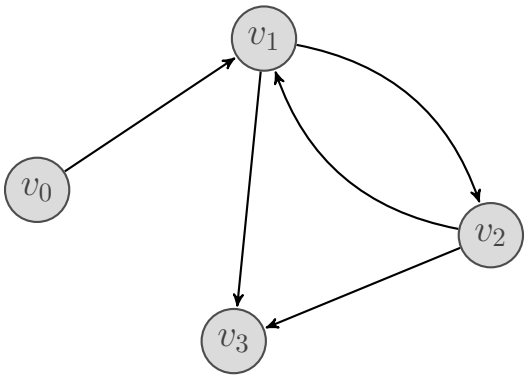
$$I_\mathcal{G}(e_1) \neq I_\mathcal{G}(e_2)$$

since

$$I_\mathcal{G}(e_1) = (v_1, v_2)$$
$$I_\mathcal{G}(e_2) = (v_2, v_1)$$

The figure Fig 2.7 represents an example of a direct graph.

Figure 2.7. Directed Graph



A further classification is that which distinguishes *cyclic* and *acyclic* graphs. A *cyclic* graph is, trivially, a graph containing at least one cycle (See definition of *cycle* in the previous section 2.2.1).

Figure 2.8 is an example of cyclic graph. It is evident that there exists, for example, a path from node v_1 which connects it to itself. The same is for the nodes v_2, v_3, v_4.

On the other hand, an *acyclic* graph is a graph without cycles, that is no node can be traversed back to itself. An example of acyclic graph is shown in Figure 2.9.

Figure 2.8. Example of Cyclic Graph

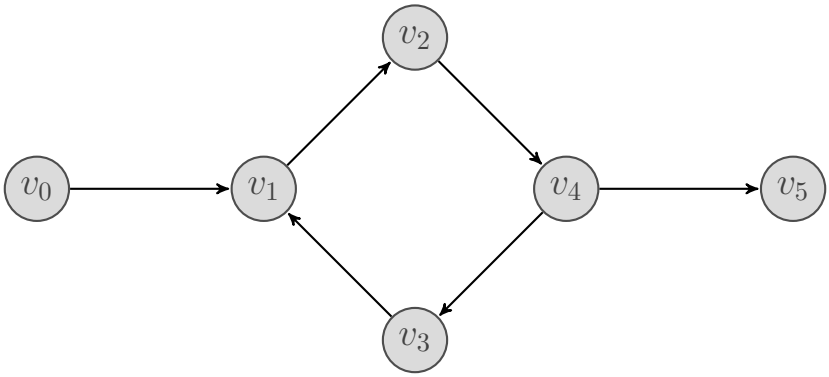
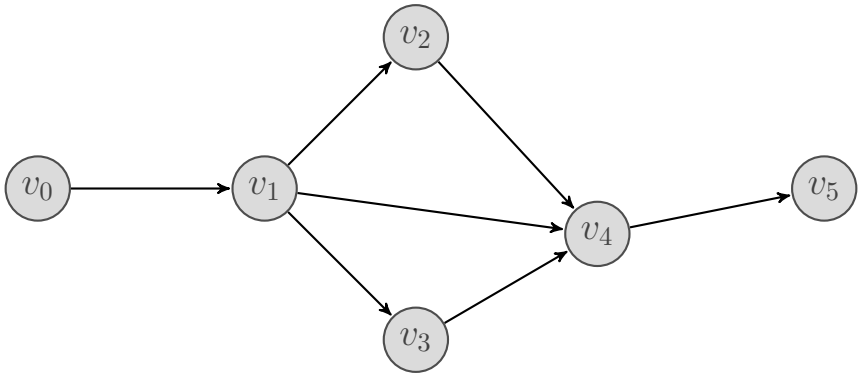


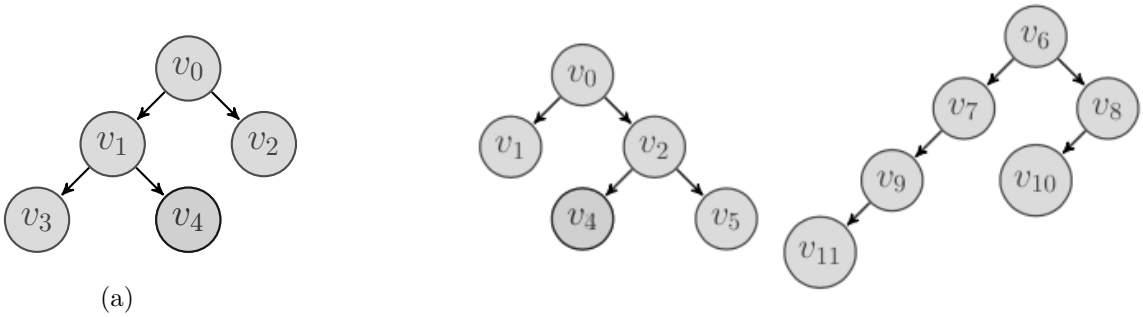
Figure 2.9. Example of Acyclic Graph



In the considered example, there are no paths which connect a vertex back to itself in the graph.

An acyclic graph is also called *forest*. A connected forest is called *tree* [2]. In other words, a forest is composed of more trees. An example of a *tree* and a *forest* are shown in Fig. 2.10.

Figure 2.10. Example of Tree (a) and Forest (b)



(a)

(b)

2.2.4 Data Structures for Representing Graphs

Sometimes a graphical representation is not sufficient to provide a complete description of a graph. It is, for example, the case of implementation of graph algorithms. There are two common data structures for representing graphs:

- representation through *adjacency matrix*
- representation through *adjacency lists*

An adjacency matrix is a binary square matrix $\mathcal{M}$ of order $n = |\mathcal{V}|$, such that, given two vertices $v_i, v_j \in \mathcal{V}$:

$$\mathcal{M}_{v_i,v_j} = \begin{cases} 1 & if \{v_i, v_j\} \in \mathcal{I}_\mathcal{G} \\ 0 & otherwise \end{cases}$$

Note that, if the graph is not oriented, $\{v_i, v_j\} \in \mathcal{I}_\mathcal{G} \iff \{v_j, v_i\} \in \mathcal{I}_\mathcal{G}$, then the matrix $\mathcal{M}$ is symmetric. For the sake of greater clarity, the following example presents the adjacency matrix $\mathcal{M}$ related to the unoriented graph in the Figure 2.6:

$$\mathcal{M} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

If the graph is weighted, that is each edge of the graph is associated to a non-zero weight w_{v_i,v_j}, then it is possible to modify the representation of the adjacency matrix $\mathcal{M}$, just putting $\mathcal{M}_{i,j} = w_{v_i,v_j}$ if $\{v_i, v_j\} \in \mathcal{V}$ belongs to the set $\mathcal{I}_\mathcal{G}$, $\mathcal{M}_{i,j} = 0$ otherwise.

Another graph representation technique is based on the use of adjacency lists. This kind of representation is widely used in computer science. It involves the use of a collection of n lists of vertices, with n the *order* of the graph $\mathcal{G}$: in other words, there exist an adjacency list for each vertex in the graph. Each list contains the nodes belonging to the *closed neighborhood set* (See section 2.2) of the corresponding vertex in the graph. In other words, the two sets coincide.

So, for example, consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{I}_\mathcal{G})$ s.t. $\mathcal{V} = \{v_k, v_{k1}, v_{k2}\}$. With the following relationships between its nodes:

$$I_\mathcal{G}(e_1) = \{v_k, v_{k1}\}$$
$$I_\mathcal{G}(e_2) = \{v_k, v_{k2}\}$$

We determine the closed neighborhood set of the v_k node, for example, in order to obtain the corresponding adjacency list L_{v_k}. So:

$$L_{v_k} = N[v_k] = \{v_k, v_{k1}, v_{k2}\}$$

or writing L_{v_k} differently

$$L_{v_k} : v_k \to v_{k1} \to v_{k2}$$

In the more complex case of a weighted graph, it is possible to use the representation by adjacency lists, simply by changing their structure.

In particular, consider, as in the previous case, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ s.t. $\mathcal{V} = \{v_k, v_{k1}, v_{k2}\}$. Then consider the following relationships between its nodes:

$$I_{\mathcal{G}}(e_1) = \{v_k, v_{k1}\}$$
$$I_{\mathcal{G}}(e_2) = \{v_k, v_{k2}\}$$

Moreover, consider the following weight indicators, each one relating to a specific edge in the graph:

$$w_{v_k,v_{k1}} \text{ w.r.t edge } e_1$$
$$w_{v_k,v_{k2}} \text{ w.r.t edge } e_2$$

As in the previous case, let us consider node v_k and compute its adjacency list L_{v_k}. We get, this time, the *open neighborhood set* of v_k, $N(v_k)$: for each node in $N(v_k)$ we consider the weight relative to the edge connecting itself and v_k. Then, considering the elements in $N(v_k)$, let's build the following pairs:

$$(v_{k1}, w_{v_k,v_{k1}})$$
$$(v_{k2}, w_{v_k,v_{k2}})$$

At this point it is easy to define the adjacency list searched for, L_{v_k}, as a sequence whose head is the v_k node and whose successive elements are pairs *node – weight*, similar to the one shown above: to underline the concept, the first element of the pair represents the node adjacent to v_k, while the second one is the weight of the edge that connects v_k to the considered adjacent node.

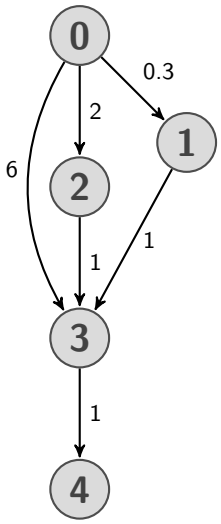
So, in the considered case, the adjacent list L_{v_k} is:

$$L_{v_k} : v_k \to (v_{k1}, w_{v_k,v_{k1}}) \to (v_{k2}, w_{v_k,v_{k2}})$$

It is worth to notice that in this discussion we will consider weighted graphs. Therefore we propose below an example of a weighted graph, represented by adjacency lists.

Consider the weighted directed graph in Figure 2.11. A representation of the graph using adjacency lists is as follows:

Figure 2.11. Weighted directed graph and the related adjacency lists.



$L_0 : 0 \to (1, 0.3) \to (2, 2) \to (3,6)$

$L_1 : 1 \to (3, 1)$

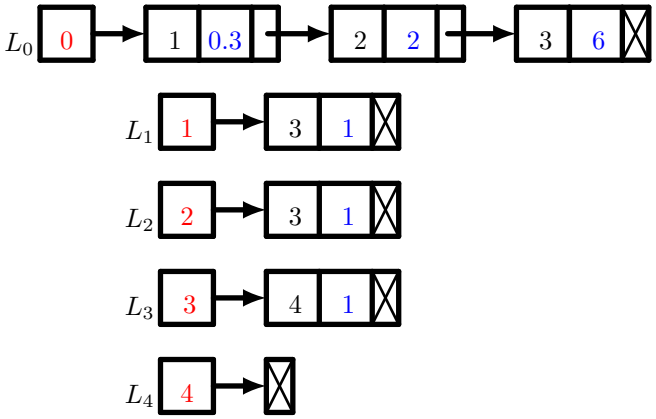
$L_2 : 2 \to (3, 1)$

$L_3 : 3 \to (4, 1)$

$L_4 : 4$

The set of adjacency lists can also be represented as shown in figure 2.12:

Figure 2.12. Graphical representation of a weighted directed graph (Fig. 2.12) by using adjacency lists. The head node is shown in red, while the weight value for each edge is represented in blue



Note the graphical representation just mentioned(figure 2.12), the last element of each list points to the $NULL$ element (represented by the box with an X). This representative choice is closely linked to the computer science concept of a list, according to which a list is a container whose last element points to the special value $NULL$.

The representation by adjacency lists is preferable to that by adjacency matrices in case of a sparse graph. Conversely, if the graph is dense, a representation by adjacency matrices is preferred. These two data structures are useful when we implement an algorithm based on graphs. Depending on the type of algorithm, one of the two representation techniques is chosen. For example, consider an algorithms that, having fixed a vertex $v_i \in \mathcal{V}$, requires iterating a certain number of steps on all vertices $v_j \in N(v_i)$: the cost of this operation is $|N(v_i)|$ if the list of nodes adjacent to v_i is represented with a list of adjacency L_{v_i}. The same operation has a cost equal to $n > |N(v_i)|$, with n the order of $\mathcal{G}$, if the representation of the graph occurs through an adjacency matrix.

We shall see the usefulness of these concepts and representation methods in sections 2.3.1 and 2.3.2, where we use them to analyse and explain shortest path algorithms such as Dijkstra and Yen algorithms. In the next section, we provide some hints on the concept of the shortest path.

2.3 Shortest path problems

The search and calculation of the shortest path over a network, or in our specific case, on a road network, is a very topical problem, target of many and different research fields over the years. The shortest path problem consists in identifying the shortest, fastest or cheapest path in a given network, starting from a source node. Numerous algorithms have been implemented for this purpose. Of course, the choice of the algorithm depends on the type of problem: depending on the application, in fact, the algorithm runtime could be a relevant consideration. This discussion, however, does not analyze in detail aspects related to the comparison between execution times of different algorithms.

In this thesis work, we use algorithms aimed at solving the problem of finding the shortest path. Keep in mind, however, that the ultimate goal of this thesis is not to find the shortest path on a network.

So, based on studies external to this dissertation, we choose to use the most well-known and used Dijkstra algorithm, which is mentioned in the next section. It is a good choice in case we

only want to find the shortest path from a source vertex to a destination vertex (this is also called "*one – to – one* shortest path problem", [3]). Practically, it carries out its research by attributing a permanent label to each node that constitutes the shortest path tree. In other words, once the node is permanently labelled, its optimal distance from the source node is known (See section 2.3.1 for more details about *distance* definition and Dijkstra's algorithm explanation).

2.3.1 Dijkstra's algorithm

The objective, as already mentioned in the previous section, is to find in a given graph $\mathcal{G}$ a path of minimum length, connecting a source o to a destination d. In other words, the algorithm looks for a path such that the distance from source and destination is the minimum.

Before proceeding with the explanation of the algorithm, it is necessary to provide the definition of *distance*, closely related to the concept of *weight* of a path.

Definition 1. Given two vertices v_0, v_l in a weighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a Weight matrix W related to the graph's edges, a path p from v_0 to v_l defined by the vertex sequence

$$p = (v_0, v_1, ..., v_l)$$

so that the relation between its nodes is

$$I_{\mathcal{G}}(e_j) = \{v_{j-1}, v_j\},\ j = 1, 2, ..., l$$

then the weight of the path p is:

$$\sum_{i=1}^{l} w_{v_{j-1}, v_j}$$

We now provide the definition of distance.

Definition 2. Given two vertices a and b in a weighted graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, we define a distance $d_{a,b}$ from a to b as:

$$d_{a,b} = min\{w(\gamma) | \gamma\ is\ a\ path\ from\ a\ to\ b\}$$

Note that $w(\gamma)$ is the *weight* of path γ connecting a and b.

Go back to the algorithm explanation, the used approach is based on building the solution in order of increasing length of the wanted optimal path.

The practical implementation, in accordance with the dedicated discussion [4], involves the division of both node and edge sets in $\mathcal{G}$ into other distinct subsets. Concerning the node-set, the split consists of dividing the elements into the following three subsets:

N.I set for which the shortest path starting from the source is known.

N.II set in which possible candidates for the N.I set are inserted. Practically, for each cycle a specific node in N.I is considered, according to the algorithm criteria: its neighbours are placed in N.II.

N.III set of the remaining nodes.

The edges of the graph are split into three different sets:

E.I set of edges that occur in the shortest path from the source node to the nodes belonging to set N.I.

E.II set in which those edges that are possible candidates for the E.I are inserted.

E.III set of the remaining edges.

The initialization of the sets is the following:

- all nodes are inserted in N.III.
- all edges are inserted in E.III.

Starting from this consideration, we put the source node o in the set N.I. The algorithm then proceeds, repeating cyclically the two following steps, as reported in Dijkstra's scientific publication [4].

Step 1 Considering a node v in N.I, we analyze its neighbours $N(v)$. For simplicity, we identify the element belonging to N(v) with the symbol n_v. Each n_v can belong to set N.II or N.III. In the first case, we consider all edges connecting v to n_v. Among these branches, consider edge e: if it generates a shorter path than the corresponding edge, belonging to set E.II, then it becomes part of the shortest path. Moreover, the edge e replaces the corresponding edge in set E.II. In the opposite case, the edge e is rejected. In the case in which, instead, the node n_v belongs to the set N.III, the approach is different: this is added to set N.II and, in parallel, the edge e is put in the E.II set.

Step 2 Selecting the set of edge E.I and a single edge belonging to E.II, there is only one path that connects the source o to each node in N.II. Therefore, we choose the node in N.II whose distance from the source is the shortest and we move it to N.I. We also move the corresponding edge from the E.II set to the E.I set. The algorithm repeats the entire process from step 1 until the destination node d is put in N.I. Therefore, the execution ends and the shortest path from o to d is returned.

Note, however, some relevant observations regarding the algorithm just analyzed: the implementation is valid for both direct and undirected graphs. Furthermore, in the case of weighted edges, it is necessary to have non-negative weights. Finally, to calculate the shortest path between any two nodes of the graph, the graph must be connected.

Below is a more practical explanation of Dijkstra's algorithm. The implementation involves the use of basic concepts not mentioned in the theoretical explanation. These concepts were used to better understand the explanation of the algorithm. Let's starting by defining a vector of nodes sh_p, representing the shortest path between source o and destination d. The algorithm has been implemented to return this vector. The sh_p is an essential result as it will be used in the following sections as input for the implementation of a more complex algorithm (see chapter 4).

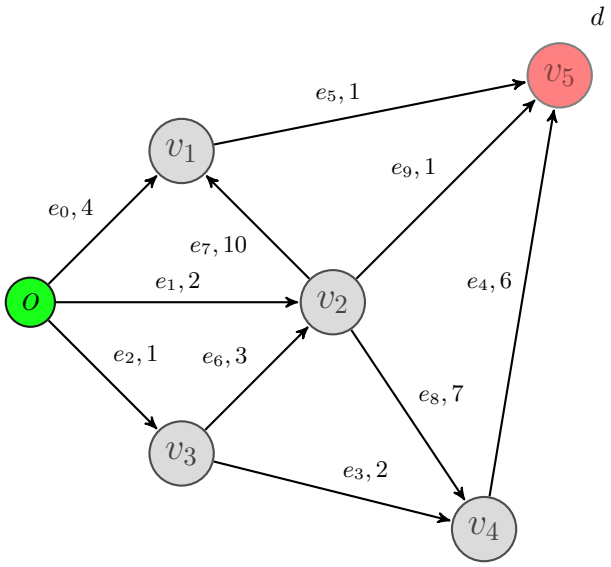
We also recall the concept of a "permanently labeled" node [3]. It represents a node for which the shortest distance from the source node is known. So, trivially, the nodes in N.I are vertices belonging to this category. The aforementioned concept is used in order not to repeatedly analyse the same node.

Under these assumptions, let us consider an explanatory example. We are given a weighted, direct graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ in Fig. 2.13 and a weight set $\mathcal{W}$. As shown in figure, the source node o is in green and the final destination v_5 (or d) is in red.

The correspondent weight matrix is:

$$\mathcal{W} = \begin{pmatrix} 0 & 4 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 10 & 0 & 3 & 7 & 1 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.13. Directed and weighted graph used to explain the Dijkstra algorithm



The algorithm starts, initializing the system: as previously written, we insert all the nodes and edges in sets N.III and E.III respectively. Then,

$$\begin{aligned} N.I &= \{\} \\ N.II &= \{\} \\ N.III &= \{o, v_1, v_2, v_3, v_4, v_5\} \\ E.I &= \{\} \\ E.II &= \{\} \\ E.III &= \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\} \end{aligned}$$

From a practical point of view, and for utility purposes, we use a *state* indicator for each node v. Each indicator consist of two elements: the first term d_v indicates the distance of the considered node v from the source node o, the second one identify the node u that precedes the analyzed one v. Note that, in general, the distance of the node v from the origin is:

$$d_v = d_u + w_{u,v}$$

with $w_{u,v}$ the weight of the edge connecting the two node u and v.

So let's start by initializing all the nodes' states, except the one related to the source node, as:

$$(d_v, u) = (\infty, /)$$

The state of the source node o is initialized as $(0, /)$ and is moved from set N.III to set N.I, becoming a permanent labeled node.

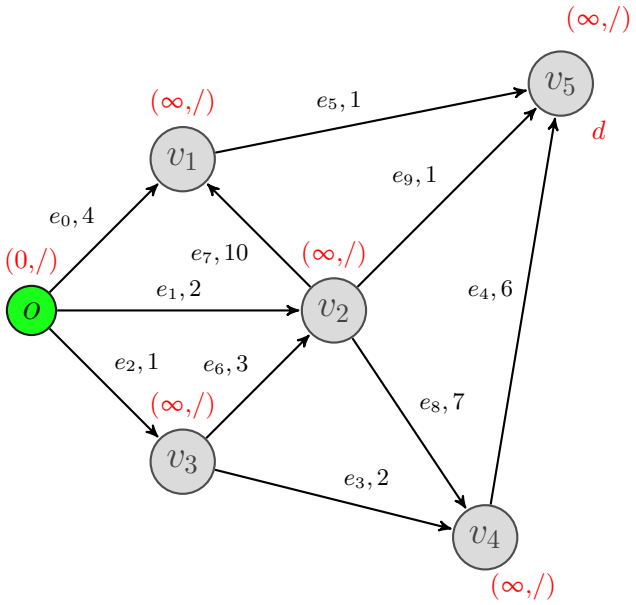


Figure 2.14. Graph with labels

$$
\begin{array}{lcl}
N.I & = & \{o\} \\
N.II & = & \{\} \\
N.III & = & \{v_1, v_2, v_3, v_4, v_5\} \\
E.I & = & \{\} \\
E.II & = & \{\} \\
E.III & = & \{e_0, e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}
\end{array}
$$

Let's start, therefore, from considering the last node added in N.I (the source node): it is denoted as a *current* node. So we find the set of its neighbours. Remember that the search for neighbouring nodes must be done between the nodes of the N.II and N.III sets. In this first case, in which the considered node is the source o, the neighbourhood set is

$$N(o) = \{v_1, v_2, v_3\}$$

We start, for example, by considering the node v_1 that belong to set N.III. The distance label of the generic adjacent node v_j with respect to v_i is update according to the following equation:

$$new \;\; d_{vj} = min\{d_{vj}, d_{vi} + w_{v_i,v_j}\} \tag{2.1}$$

So, in out specific case:

$$new \; d_{v1} = min\{d_{v1}, d_o + w_{o,v_1}\} = min\{\infty, 0 + 4\} = 4$$

Note that the formula used is directly linked to that written in *Step* 1. If d_{v1} is update by its new value, the node v_1 is moved from the set N.III to the set N.II and, at the same time, the corresponding edge in E.II, if exist, is replaced by the new one. In our case, we have no corresponding edge, so e_0 is simply shifted from E.III to E.II. Note that, if *new* d_{v1} had taken its old value, the edge e_0 would have been removed from E.III and no other operations would have been performed.

Practically:

$$\begin{array}{lcl} N.I & = & \{o\} \\ N.II & = & \{v_1\} \\ N.III & = & \{v_2, v_3, v_4, v_5\} \\ E.I & = & \{\} \\ E.II & = & \{e_0\} \\ E.III & = & \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\} \end{array}$$

Because at the beginning the distance from o of each node is initialized to ∞, all nodes adjacent to the source node during the first cycle are moved from N.III to N.II (the same reasoning is true for the corresponding edges).

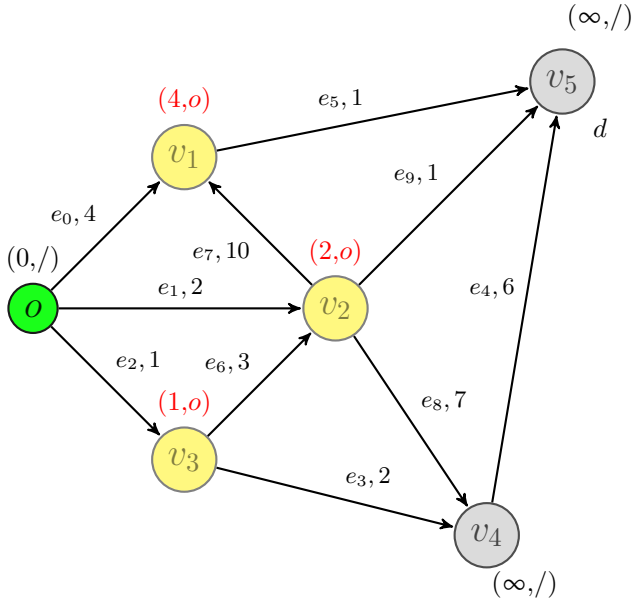
In any case, we calculate the new distances for the remaining nodes in $N(o)$, v_2, v_3.

Then

$$\begin{array}{l} new \;\; d_{v2} = min\{d_{v2}, d_o + w_{o,v_1}\} = min\{\infty, 0+2\} = 2 \\ new \;\; d_{v3} = min\{d_{v3}, d_o + w_{o,v_1}\} = min\{\infty, 0+1\} = 1 \end{array}$$

Updating all labels, we obtain the graph as shown in figure 2.15.

Figure 2.15. The adjacent nodes of o (in yellow) are labelled. The table shows the update of their status label.



	o	v_1	v_2	v_3	v_4	v_5
d:	0	∞	∞	∞	∞	∞
u:	/	/	/	/	/	/
d:	0	4	2	1	∞	∞
u:	/	o	o	o	/	/

The final sets will be equal to:

$$\begin{aligned} N.I &= \{o\} \\ N.II &= \{v_1, v_2, v_3\} \\ N.III &= \{v_4, v_5\} \\ E.I &= \{\} \\ E.II &= \{e_0, e_1, e_2\} \\ E.III &= \{e_3, e_4, e_5, e_6, e_7, e_8, e_9\} \end{aligned}$$

When all the nodes in the neighbourhood set have been considered and analysed, we proceed with *Step* 2. We chose the node in N.II whose distance from the source node is minimal. Then, among the three newly calculated distances, the minimum value is given by *new* d_{v_3}. Then we move the corresponding node v_3 from N.II to N.I (the corresponding e_2 is moved from E.II to E.I): v_3 has been permanently labelled. So:

$$\begin{aligned} N.I &= \{o, v_3\} \\ N.II &= \{v_1, v_2\} \\ N.III &= \{v_4, v_5\} \\ E.I &= \{e_2\} \\ E.II &= \{e_0, e_1\} \\ E.III &= \{e_3, e_4, e_5, e_6, e_7, e_8, e_9\} \end{aligned}$$

At this point we designate the node just moved v_3 as the *current* node and repeat the proposed procedure. Let analyse the nodes belonging to $N(v_3) = \{v_2, v_4\}$ and compute the new distances from the source o. Note that edges e_3 and e_6, referring to v_4 and v_2 respectively, are moved from E.III to E.II. v_2, instead, is still in N.II, so no operation about nodes are performed. So:

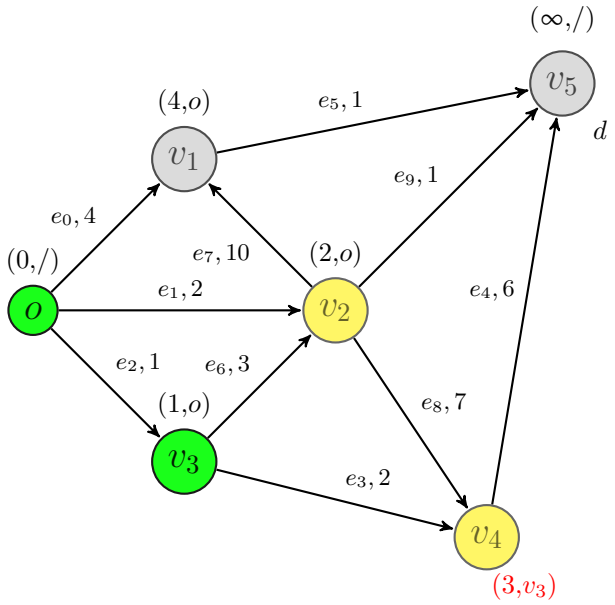
$$\begin{aligned} N.I &= \{o, v_3\} \\ N.II &= \{v_1, v_2, v_4\} \\ N.III &= \{v_5\} \\ E.I &= \{e_2\} \\ E.II &= \{e_0, e_1, e_3, e_6\} \\ E.III &= \{e_4, e_5, e_7, e_8, e_9\} \end{aligned}$$

Computation of new distances:

$$\begin{aligned} new\ d_{v2} &= min\{d_{v2}, d_{v3} + w_{v_3,v_2}\} = min\{2, 1+3\} = 2 \\ new\ d_{v4} &= min\{d_{v4}, d_{v3} + w_{v_3,v_4}\} = min\{\infty, 1+2\} = 3 \end{aligned}$$

Also referring to the table in the figure 2.17, we determine which of the nodes in N.II presents minimal distances from the source.

Figure 2.16. .



	o	v_1	v_2	v_3	v_4	v_5
d:	0	∞	∞	∞	∞	∞
u:	/	/	/	/	/	/
d:	0	4	2	1	∞	∞
u:	/	o	o	o	/	/
d:	0	4	2	1	3	∞
u:	/	o	o	o	v_3	/

The node, among the possible ones, whose distance from the source is the minimum is v_2. It is the new *current* node and is moved from N.II to N.I, becoming a permanently labelled node. Since the value of d_{v2} has not been modified, we choose not to move, simply deleting the corresponding edge e_6 from E.II. So we have the following sets:

$$\begin{aligned} N.I &= \{o, v_3, v_2\} \\ N.II &= \{v_1, v_4\} \\ N.III &= \{v_5\} \\ E.I &= \{e_2\} \\ E.II &= \{e_0, e_1, e_3\} \\ E.III &= \{e_4, e_5, e_7, e_8, e_9\} \end{aligned}$$

At this point, starting from the new *current* node v_2, we analyse its neighbours v_1, v_5 and move them, if necessary, from N.III to N.II (e_7, e_9 are moved from E.III to E.II).

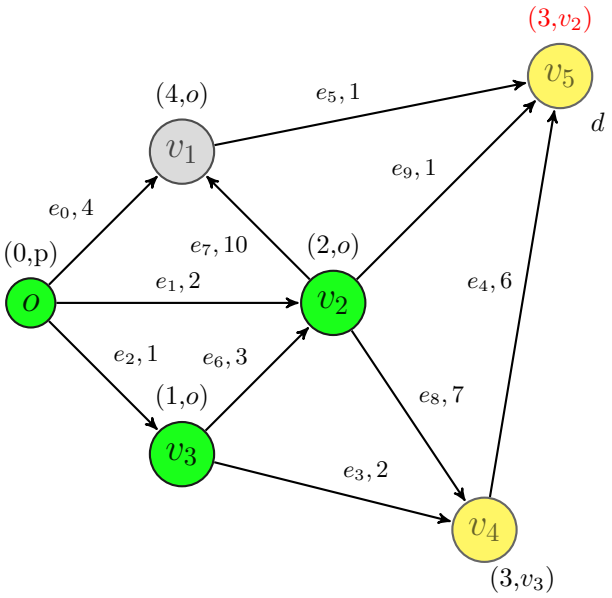
$$\begin{aligned} N.I &= \{o, v_3, v_2\} \\ N.II &= \{v_1, v_4, v_5\} \\ N.III &= \{\} \\ E.I &= \{e_2\} \\ E.II &= \{e_0, e_1, e_3, e_7, e_9\} \\ E.III &= \{e_4, e_5, e_8\} \end{aligned}$$

We update the status of the nodes belonging to $N(v_2)$, calculating the new distances from the source:

$$\begin{aligned} new\ d_{v1} &= min\{d_{v1}, d_{v2} + w_{v_2,v_1}\} = min\{4, 2+10\} = 4 \\ new\ d_{v5} &= min\{d_{v5}, d_{v2} + w_{v_2,v_5}\} = min\{\infty, 2+1\} = 3 \end{aligned}$$

Graphically (Fig. 2.17):

Figure 2.17. .



	o	v_1	v_2	v_3	v_4	v_5
d:	0	∞	∞	∞	∞	∞
u:	/	/	/	/	/	/
d:	0	4	2	1	∞	∞
u:	/	o	o	o	/	/
d:	0	4	2	1	3	∞
u:	/	o	o	o	v_3	/
d:	0	4	2	1	3	3
u:	/	o	o	o	v_3	v_2

So we select the node, among the possible ones, whose distance is the minimum. That node is v_5. Note that it coincides with the destination node d, so the algorithm stops: we found the length of the shortest path from the origin to the destination node.

At this point, it is necessary to go back over the nodes that constitute the shortest path, to return the searched node vector sh_p: remember that this represents the sequence of nodes that constitute the shortest path from the source node to the destination node. The vector is built using a temporary variable *temp*: at each step, *temp* stores and adds at the beginning of the vector sh_p an element constituting the predecessor of the node observed in the previous step. The process continues until the source node o is added. So, starting from the destination node v_5 and considering the table in figure 2.17:

1. $temp = v_5 \rightarrow sh_p = (v_5)$

2. $temp = v_2 \rightarrow sh_p = (v_2, v_5)$

3. $temp = o \rightarrow sh_p = (o, v_2, v_5)$

We can conclude, therefore, that the shortest path is

$$sh_p = (o, v_2, v_5)$$

and that the minimum distance of the destination v_5 from the source node o is 3.
The pseudocode of the algorithm up to now explained is provided below.

Pseudocode: Dijkstra's algorithm

Algorithm 1 Dijkstra

Input: $nodesVector$ (a vector containing all the nodes of the graph), $weighMatrix$ (weight matrix of the considered graph), $source$ (source node), $destination$ (destination node)
Output: $shortestPath$ (list representing the shortest path from the source node to the destination node)

```
1:  shortestPath ← NULL
2:  tempNode ← destination
3:  for all vertex v in nodesVector do
4:      dist[v] ← infinite
5:      prev[v] ← undefined
6:  end for
7:  dist[source] ← 0
8:  while nodesVector is not empty do
9:      u ← vertex in nodesVector with minimum dist[u]
10:     remove u from nodesVector
11:     for all neighbour v of u do
12:         tempDist ← dist[v] + weight(u, v)
13:         if tempDist < dist[v] then
14:             dist[v] ← tempDist
15:             prev[v] ← u
16:         end if
17:     end for
18: end while
19: while tempNode ≠ source do
20:     shortestPath.add(tempNode)
21:     tempNode ← prev[tempNode]
22: end while
23: return shortestPath, dist[destination]
```

Further details, such as the proof of correctness of the algorithm, can be found in dedicated scientific documents and publications.

2.3.2 Yen's algorithm

Based on what was written in the previous section, a more complex problem is now defined, related to the search for the K shortest paths, KSP, in a given network. KSP are algorithms mainly used in the field of communications, research operations, computer and transportation science.

As explained above, the problem of finding the shortest path in a graph arises from the objective of finding a minimum path between two nodes called respectively *source* and *destination*. However, there could be eventualities such as, for example, route interruption or damage: it is, therefore, necessary to search for an alternative path that links the *source* and *destination*, through paths that are as short as possible between those allowed.

In this section, we will focus on finding the K shortest paths in a network without loops. The Yen's algorithm [5] seems to be a good choice when compared with other algorithms solving the KSP problem. As reported in the paper by Yen [5], in fact, the computational upper bound of this algorithm increases linearly with the value of K.

Below are some preliminary definitions and considerations useful for explaining the Yen's algorithm.

Definitions and basic considerations

Recalling the concepts of *path* and *distance* already discussed in the previous sections (sections 2.2.1 and 2.3.1 respectively), the following definitions are provided.

Let $\mathcal{G}$ be a network, or graph, consisting of n nodes. Let

- $d_{i,j} \leq 0$ or $d_{i,j} > 0$, ij, be the distance between two adjacent nodes, i, $j \in \mathcal{G}$. Note that, trivially, if the edge doesn't exist, the distance is considered equal to infinity;
- $A^k = (1^k, 2^k, 3^k, ..., Q_k^k, N)$ be the k-th shortest path connecting its node at the first position to its destination node (at the last position N). So $1^k, 2^k, ..., Q_k^k$ are respectively the node in position $1, 2, ..., Q_k$ of the k-th shortest path. Note also that $k = 1, 2, ..., K$;
- $A_i^k, i = 1, 2, ..., Q_k$ be a set of deviations from path A^{k-1} at node i. Practically, a deviation is accomplished by searching for the shortest path that coincides with A^{k-1} from the first node to the i-th node of the path and then deviates to a node different from any of the $(i+1)$st nodes in A^j, considering $j = 1, 2, .., k-1$, that have the same sequence of nodes, from the first position (source node) to the i-th one as does A^{k-1}. The deviation must, therefore, lead to the destination d via the shortest path which must not pass through any nodes already included in the first part of the path;
- R_i^k be the *root* of A_i^k: it is the subpath consisting of the sequence of nodes that A_i^k and A^{k-1} have in common, i.e. from the source node to the i-th node;
- S_i^k be the *spur* of A_i^k: it is the subpath coinciding with the last part of A_i^k, from the i-th node to the last one (i.e. the destination).

Moreover, in this dissertation, we will assume to consider a $non - negative - distance$ and *loopless* network.

Description of the algorithm

We assume to work with two different containers, or *Lists*, of paths, A and B. In the container A, we hold the K shortest paths from the source o to the destination d, whereas the List B represents the set of possible candidates from which we can choose, to find A^k.

For greater clarity, the description of the algorithm is divided into two parts: the first one related to the first iteration, so for $k = 1$, the second one related to the generic k-th iteration [5].

Iteration 1

Starting from $k = 1$, we find $A^k = A^1$. Note that any efficient shortest path algorithm can be used to solve this first step: in this case, we use the aforementioned Dijkstra's algorithm (See section 2.3.1). Using the Dijkstra's algorithm, and remembering that we consider a no negative loops network, we should get at least a shortest path. If the Dijkstra's algorithm returns some shortest paths greater than or equal to K, we are done because the Yen solution coincides with the K paths just obtained. In this case, the execution stops and the resulting A^1 is stored in List A. Otherwise, if the Dijkstra algorithm produces a number of paths less than K and greater than 0, we assign any arbitrary one of these paths to be A^1. Then it can be stored in A, while the rest of the resulting paths can be hold in List B. So the execution continues, until set A is full (Capacity of B equal to K paths).

Iteration $k = 2, 3, ..., K$

The computation of A^k is strongly dependent on $A^1, A^2, ..., A^{k-1}$, calculated during the previous iterations. It is, in turn, divisible into two steps, the first one consists in determining all the deviations A_i^k, the second one in choosing the shortest path which will become A^k.

Process I. For each $i = 1, 2, ..., Q_{k-1}$:

(a) we identify the *root* R_i^k: it is the subpath in A^{k-1} whose nodes in sequence coincide with the sequence of nodes of length i in A^j, $j = 1, 2, ..., k-1$. So we set the distance $d_{i,i+1} = \infty$: it represent the cost of the edge between the i-th node and the $(i+1)$-th node of A^j $(j = 1, 2, ..., k-1)$. This setting coincides with the actual removal of the edge in question and is valid only during the k-th iteration. Once the k-th iteration is finished, the modified distance assumes its initial value.

(b) we use a shortest path algorithm (in our case Dijkstra's algorithm 2.3.1) to obtain the shortest path from the i-th node to destination d, avoiding to pass through the node already contained in the path. In other, we find the *spur* path of A_i^k previously defined, S_i^k. We can, in addition, define i as the *spur node*.

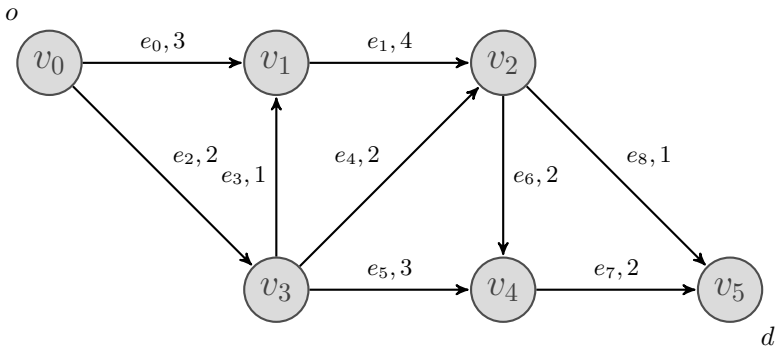
(c) we add the path A_i^k, resulting by joining the two set R_i^k and S_i^k, to List B.

Process II. We have to select, among the possible paths in List B, the one(s) with minimum length, in order to denote it (or them) as A^k. So we move it (or them) from B to A. If set A reaches its maximum capacity of K paths, execution ends, otherwise it is necessary to reset the cost of the edge removed during the considered k-th iteration and increase the value of k by one, starting again from the process I.

To provide a more practical explanation, an example is proposed.

Example 2.3.1. Let $\mathcal{G}$ be a loopless network with non-negative edge cost, as shown in Fig. 2.18. For simplicity, we will consider a 2-SP problem, i.e. in which we have to find the two shortest paths that connect the source o to the destination node d in $\mathcal{G}$.

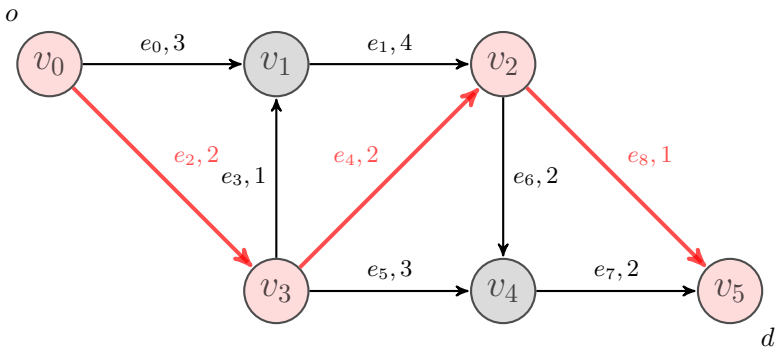
Figure 2.18. Loopless graph with no negative edges cost, used to explain Yen's algorithm



Starting from $k = 1$, let determine A^1 (See Fig. 2.19), by using the Dijkstra's algorithm:

$$A^1 = (v_0, v_3, v_2, v_5)$$

Figure 2.19. Nodes and edges that make up path A^1 are shown in red



and the cost of the path is 5.
The obtained path is moved in the A set, so that:

$$A.push\_back((v_0, v_3, v_2, v_5))$$

At this point, we increase the value of k, so that $k = 2$ (equivalent to the upper limit K): the sought set is A^2. For each $i = 1, 2, ..., Q_{k-1}$, the algorithm follow the three steps previously written, i.e. computing the root path R_i^k, the spur path S_i^k, and the resulting final path A^k. Note also that, in this case, $Q_{k-1} = 3$.

For i =1:

- The subpath, consisting of the first node ($i = 1$) of $A^{k-1} = A^1$, coincides with the subpath composed by the first node of A^j ($j = 1, ...,1$), that is:

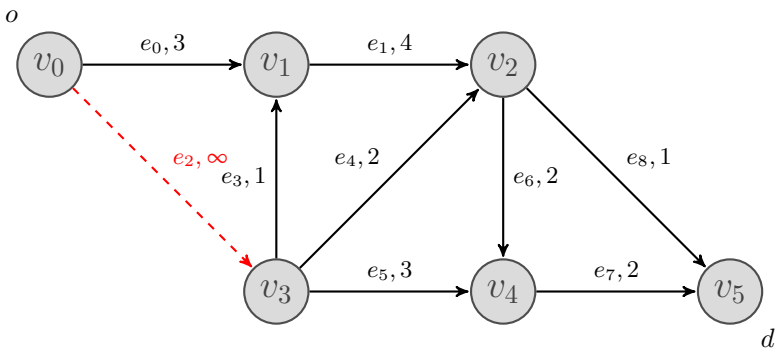
$$v_0 \equiv v_0$$

Note that this first step makes more sense if $K > 2$. In that case, we can compare the first i node of A^{k-1} with the first i node of A^j, taking into account that

$$1 \leq j \leq k < K$$

So we remove the edge between the i-th node and the $(i+1)$-th node of A^1, i.e. e_2: it also means that the distance d_{v_0,v_3} between the two nodes just mentioned is set equal to infinity (See Fig. 2.20).

Figure 2.20. Loopless graph with no negative edges cost, used to explain Yen's algorithm



Moreover we state that the *root* path R_i^k is $R_1^2 = \{v_0\}$.

- We compute the *spur* path S_1^2 by using the Dijkstra's algorithm to find the shortest path from the 1-st node ($i = 1$) to the destination v_5. So we obtain:

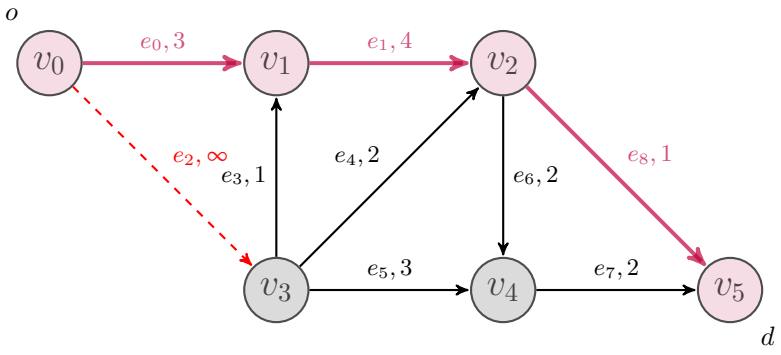
$$S_1^2 = (v_0, v_1, v_2, v_5)$$

- The last step joins the two founded sets, R_1^2 and S_1^2, obtaining A_1^2 (See Fig. 2.21), that is:

$$A_1^2 = R_1^2 + S_1^2 = (v_0, v_1, v_2, v_5)$$

Its cost is 8.

Figure 2.21. Nodes and edges that make up path A_1^2 are shown in pink



So the obtained path A_1^2 is added to List B:

$$B.push\_back((v_0, v_1, v_2, v_5))$$

Note also that the cost of the edge e_2 is reset to its initial value 2.

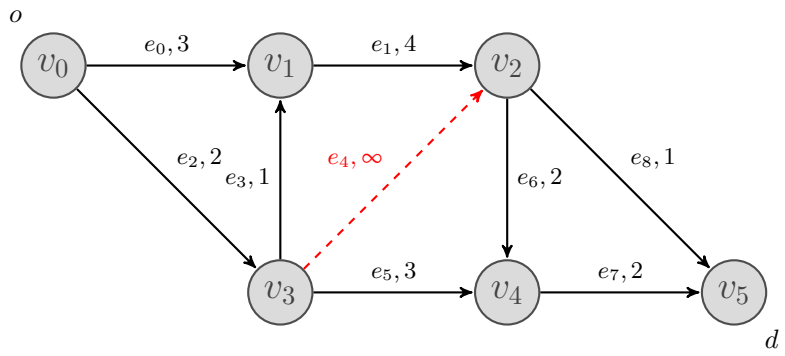
For i =2: The same steps are followed.

- The first two nodes (i = 2) of A^1 coincides with the first two nodes of A^j (j = 1, ...,1). In fact,

  $$\{v_0, v_3\} \equiv \{v_0, v_3\}$$

  So the edge e_4 is removed, i.e. the distance d_{v_3,v_2} is set to infinity (See Fig. 2.22) .

Figure 2.22. Removing the e_4 edge from the network



Moreover the *root* path will be:

$$R_2^2 = (v_0, v_3)$$

- By using the Dijkstra algorithm, the *spur* path is

  $$S_2^2 = (v_3, v_4, v_5)$$

- In the last step we joins the two founded sets, R_2^2 and S_2^2, obtaining A_2^2 (shown in green in Fig. 2.23), that is:

  $$A_2^2 = R_2^2 + S_2^2 = (v_0, v_3, v_4, v_5)$$

  Its cost is 7.

The obtained path A_2^2 is added to List B:

$$B.push\_back((v_0, v_3, v_4, v_5))$$

The cost of the removed edge e_4 is reset to its initial value 2.

For i =3:

- The first three nodes (i = 3) of A^1 coincides with the first three nodes of A^j (j = 1, ...,1). So the edge e_8 is removed, i.e. the distance d_{v_2,v_5} is set to infinity (See Fig.2.24).

Figure 2.23. Nodes and edges that make up path A_2^2 are shown in green

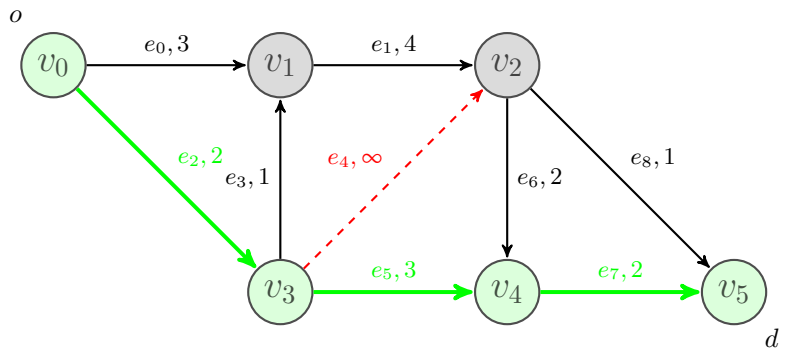
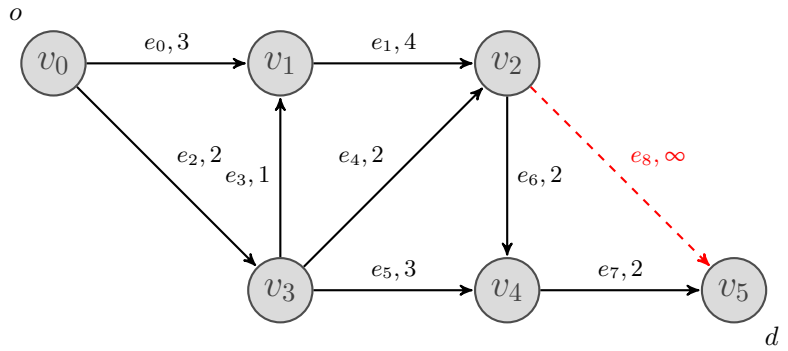


Figure 2.24. Removing the e_8 edge from the network



Moreover the *root* path will be

$$R_3^2 = (v_0, v_3, v_2)$$

- By using the Dijkstra algorithm, the *spur* path is

$$S_3^2 = (v_2, v_4, v_5)$$

- In the last step we joins the two founded sets, R_3^2 and S_3^2, obtaining A_3^2 (shown in Fig. 2.25), that is:

$$A_3^2 = R_3^2 + S_3^2 = (v_0, v_3, v_2, v_4, v_5)$$

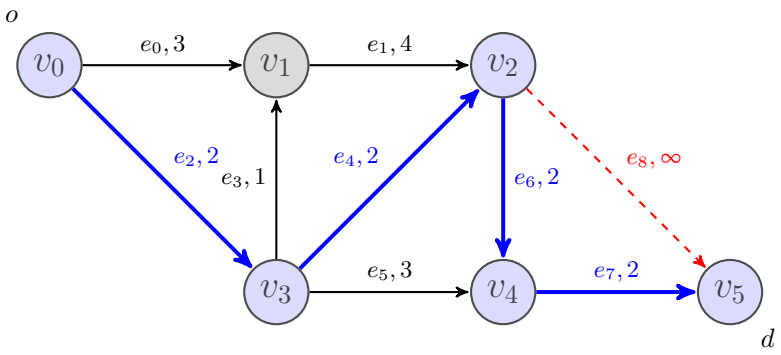
  Its cost is 8.

The obtained path A_3^2 is added to List B:

$$B.push\_back((v_0, v_3, v_2, v_4, v_5))$$

The cost of the removed edge e_8 is reset to its initial value 1.

Figure 2.25. Nodes and edges that make up path A_3^2 are shown in blue



Then, among the three obtained paths,A_1^2, A_2^2, A_3^2, stored in B, we choose the one with the lowest cost and move it to A. So, given B:

$$\begin{array}{lllll} B\text{:} & (v_0, v_1, v_2, v_5) & A_1^2 & \text{cost } 8 \\ & (v_0, v_3, v_4, v_5) & A_2^2 & \text{cost } 7 \\ & (v_0, v_3, v_2, v_4, v_5) & A_3^2 & \text{cost } 8 \end{array}$$

we choose the path A_2^2, due to its minimum cost of 7, moving it from B to A. So we obtain:

$$\begin{array}{lll} A\text{:} & (v_0, v_3, v_2, v_5) & \text{cost } 5 \\ & (v_0, v_3, v_4, v_5) & \text{cost } 7 \end{array}$$

and it is the solution of the proposed problem.

The pseudocode of the Yen algorithm is reported and discussed below.

Pseudocode: Yen's algorithm

Algorithm 2 Yen

Input: $nodesVector$ (vector of nodes in the given graph), $weightMatrix$ (weight matrix of the considered graph), $source$ (source node), $destination$ (destination node), K (number of shortest paths we are looking for)

Output: A (container of the K shortest paths from the source node to the destination node)

$A[0] \leftarrow Dijkstra(nodesVector, weightMatrix, source, destination)$ //See function 1
$B \leftarrow []$ //Set of possible candidates
for k from 1 to K **do**
  for i from 0 to $size(A[k-1]) - 2$ **do**
    $spurNode = A[k-1].node(i)$
    $rootPath = A[k-1].nodes(0, i)$ //nodes of $A[k-1]$ from 0 to i
    for all path p in A **do**
      if $rootPath$ equal to $p.nodes(0, i)$ **then**
        remove $p.edge(i, i+1)$ from $weightMatrix$
      end if
    end for
    for all node $rootPathNode$ in $rootPath$ except $spurNode$ **do**
      remove $rootPathNode$ from $nodesVector$
    end for
    $spurPath = Dijkstra(nodesVector, weightMatrix, spurNode, destination)$
    $totalPath = rootPath + spurPath$
    $B.append(totalPath)$
    restore nodes in $rootPath$ to $nodesVector$
    restore edges to $weightMatrix$
  end for
  if B is empty //If there are no $spurPath$ **then**
    break
  end if
  $B.sort()$ //Sort the potential k-shortest paths by cost
  $A[k] = B[0]$
  $B.pop()$
end for
return A

Chapter 3

Goal Recognition

3.1 Introduction

In this chapter, we provide some concept concerning Goal Recognition (GR) and Goal Recognition Design(GRD), starting from a brief overview on Plan Recognition, of which the GR is a sub-problem, and Classical Planning techniques.

3.2 Plan Recognition

Plan recognition (PR) fits into different contexts and applications related, for example, to multi-agent systems, assisted cognition, natural language ([6], [7], [8]). It is necessary to distinguish planning and Plan Recognition. In fact, in a typical planning problem, the goals are known: the problem involves the planning of a strategy useful for achieving the final goals. Differently, in Plan Recognition, part of the plan is provided: the task of the PR is to identify the goals and the complete plan. Plan Recognition tries to predict the goal of an agent, based on the observations of its behavior and actions.

Still object of study and research, Plan Recognition can be treated by exploiting techniques based on classic planning algorithms, as suggested by modern papers (Ramirez and Geffner 2009 [9]): recent planning algorithms are used to recognize set $\mathcal{G}^*$ of goals G that, given a domain theory, comply a series of observations. A goal G belongs to set $\mathcal{G}^*$ if exists an action sequence π that constitutes an optimal plan for both the goal G and the goal G extended with additional goals representing the observations (Ramirez and Geffner, 2009 [9]). The calculation of set $\mathcal{G}^*$ is based on the use of modified versions of optimal and suboptimal planning algorithms, as well as on the polynomial heuristics. Using these tools, it is possible to state that the set $\mathcal{G}^*$ can be calculated *exactly* or *approximately*, obtaining good but slightly different performances concerning speed and efficiency [9].

We first present a basic planning background, in order to better understand how a PR problem can fit into a similar context. A Strips planning problem is a tuple

$$P = \langle F, I, A, G \rangle$$

where F is a defined as set fluents, $I \subseteq F$ and $G \subseteq F$ are the initial and goal situations, A is the set of actions a, s.t. $Pre(a), Add(a), Del(a)$ are *preconditions*, *add* and *delete* operations respectively. All of these elements are subsets of F [9]. For each $a \in A$, it is possible to associate a cost whose value, for assumption, is non-negative: so, for a plan $\pi = a_1, a_2, ..., a_n$, it is therefore worth that

$$c(\pi) = \sum c(a_i), i = 1, ..., n$$

If the cost is the minimum, π is defined as optimal plan [9]. Note that, if we assume that each action has a unit cost, the optimal plans coincide with the shortest ones.

The concept of cost of a plan is closely related to the probability that a given agent follows that plan: the higher the cost of the plan, the less, of course, the probability that the agent chooses it. In order to find a plan that is as probable as possible, the planning looks for a strategy to calculate a set of optimal paths, using algorithms based on heuristic search, for example. Note that this calculation is very hard and expensive. In particular, the research aims to find a heuristic $h(s)$ such that

$$h(s) \leq h^*(s)$$

with $h^*(s)$ the optimal cost from s [9]. In order to make this research less hard, planning techniques can instead search for suboptimal plans, using suboptimal planners: these planners return good results and scale up much better than optimal planners [9].

As previously written, the new Plan Recognition approaches base their problems over a domain theory: they replace the set of plans for a specific goal G in a library by the set of optimal plans for G given the domain P. It is possible to represent the *planning domain* $P = \langle F, I, O \rangle$: it is a planning problem without a goal [9]. A *planning problem* is, in fact, defined as $P[G]$: it is the concatenation of planning domain and goal G. To solve a typical planning problem, it is possible to use existing planning tools: the planners are invoked not on the problem $P[G]$ that does not take into account of the observations. They use, instead, a modified, transformed version of the problem: the solution will be a set of plans for $P[G]$ that satisfy the observations [9]. In particular, given a plan recognition problem $T = \langle P, \mathcal{G}, O \rangle$, where $P = \langle F, I, A \rangle$ is a planning domain, $\mathcal{G}$ is the set of possible goals $G \cup F$ and $O = o_1, ..., o_m$ is a sequence of observations (each o_i begin an action in A), we transform it into a different planning recognition problem $T' = \langle P', \mathcal{G}', O' \rangle$. Note that the sequence of transformed observations is an empty set. By using this transformed version of the problem, we can read off from the solution $\mathcal{G}_T^{*\prime}$ for T' the sought set $\mathcal{G}_T^*$ (solution with respect to T). The transformation from T to T', according to (Ramirez and Geffner, 2009), is such that

- $P' = \langle F', I', A' \rangle$: $F' = F \cup F_o$, $I' = I$, $A' = A \cup A_o$, $F_o = \{p_a | a \in O\$$, $A_o = \{o_a | a \in O\}$.
- $\mathcal{G}'$ is constituted by goals $G' = G \cup G_o$ such that $G \in \mathcal{G}$.
- O' is empty.

Note that the o_a actions in P' have the same *precondition*, *add* and *delete* settings as the action a in P, being a the first observation in O. The only difference is the presence of the new fluents p_a and p_b, being b the action that immediately precedes a in O. They are added to $Add(o_a)$ and $Pre(o_a)$ respectively.

So, by using this transformed planning recognition problem T', we obtain additional fluents $p_a \in F_o$, additional actions $o_a \in A_o$ and additional goals $p_a \in G_o$ [9]. In particular, this extra goals can only be achieved by the additional actions o_a: they can be applied only after all the actions in O preceding a have been considered (think of precondition p_b in $Pre(o_a)$). So the result is a correspondence between the plan P' and the plan P satisfying the observations O. Specifically, given a plan $\pi = a_1, ..., a_m$, it is a plan for G in P that satisfies the observations $O = o_1, ..., o_m$ if and only if $\pi' = b_1, ..., b_n$ is a plan for G' in P' with $b_i = o_{a_i}$, if exist a function f such that $i = f(j)$ and $j \in [1, m]$, otherwise $b_i = a_i$ [9]. So, to find an optimal plan π that satisfies the observations, it is necessary to find an optimal plan π' in P', such that it achieves two goals, G and $G' = G \cup G_o$. Moreover, defining $c^*_{P'}(G)$ as the optimal cost of achieving G in P', it is possible to state that the goal G belongs to optimal set $\mathcal{G}_T^*$ iff $c^*_{P'}(G) = c^*_{P'}(G')$ [9]. This means that in the transformed problem, the extra goals replacing the observations have no extra cost.

Some exact and approximate algorithms to compute the set of optimal goal $\mathcal{G}^*$ are proposed (Ramirez and Geffner, 2009 [9]). The empirical evaluation shows that the approximate methods

are faster than optimal ones. Obviously, the optimal algorithms make no error at all, as they are used to compute the optimal goal exactly. The suboptimal and heuristic ones, instead, work well with some domains, a little worse with others. Anyway, it is possible to state that the approximate method are good enough for larger problems [9]. A critical aspect of this model is that it does not weight the possible goals and it just filter them.

The concept of weight associated with the possible hypotheses (goals) was introduced by a more recent paper [10]: the approach assigns a weight $\Delta(G)$ to each possible goal G equal to the difference between the cost $c^*_{P'}(G')$ of solving the goals $G' = G \cup G_o$ (goals of the transformed set P') and the cost $c^*_{P'}(G)$ of solving the goals G alone. The greater difference means less probability that goal G is achieved by satisfying observations: $\Delta(G)$ is, in fact, a measure of how far the agent has to move away from the best plans to reach G, in order to comply with the observations. So, such weight is used to express the probability distribution over the goals, given the observations [9].

The preliminary concepts used so far in Ramirez and Geffner's discussion (2009) [9], are still valid for the new one: however, the transformation method used to map the domain P into a new domain P' differs slightly. $P' = \langle F', I', A' \rangle$ is a new domain such that:

- $F' = F \cup \{p_a | a \in O\}$
- $I' = I$
- $A' = A$

The new transformation does not add new actions. This new approach is based on considering the plans for goal G that satisfy and do not satisfy the observation sequence O ($G + O$ and $G + \bar{O}$).

The same considerations as before are taken into consideration: in particular

- π is a plan for $P[G]$ that satisfies the observations O iff π is a plan for $P'[G + O]$.
- π is a plan for $P[G]$ that does not satisfy the observations O iff π is a plan for $P'[G + \bar{O}]$.

Moreover, assuming that $c(G), c(G, O), c(G, \bar{O})$ are the optimal cost of the planning problems $P'[G], P'[G + O], P'[G + \bar{O}]$ respectively, if we applied the old approach, according to which the goals selected G are the most likely ($\Delta(G) = 0$), we should select the goals $G \in \mathcal{G}$ for which $c(G) = c(G, O)$ holds [10]. The new approach focuses, instead, on cost difference $c(G, O) - c(G, \bar{O})$: this implies a probability distribution over the goals and not just a partition [10].

Under these assumptions, it is possible to define probabilistic distribution. A *probabilistic plan recognition* problem is a tuple $T = \langle P, \mathcal{G}, O, Prob \rangle$ [10]: all this elements have been discussed previously; the just introduced $Prob$ element represents a probability distribution over $\mathcal{G}$.

By using the Bayes Rule and some information about *goal priors*, it is possible to compute the posterior probabilities $P(G|O)$

$$P(G|O) = \alpha P(O|G) P(G)$$

where α is a normalizing constant, $P(G)$ is $Prob(G)$, $P(O|G)$ represents the probability of observing O when the goal is G. It is possible to shows and verify that the most likely goals G are the ones that minimize the difference cost proposed above ($c(G, O) - c(G, \bar{O})$) [10].

Through experimental evaluations in [10], it is possible to state that the just presented approach to Plan Recognition leads to good and flexible solutions. Moreover, in this new formulation, the agents do not assume as perfectly rational agents, so they can follow the plan with no minimum cost. This is certainly a more realistic scenario than the previous one in [9].

The two approaches to Plan Recognition ([9] and [10]) are defined as generative. Obviously, there are other types of non-generative approaches based on the use of libraries of plans or policies.

From a computational point of view, instead, a type of approach based on Bayesian Dynamic Networks is spreading. In these approximate algorithms are used (Rao-Blackwellised particle filtering is an example). The advantage is that these techniques manage and handle stochastic actions and sensors, and incomplete information. On the other hand, the applicability of such techniques in applications where the planning horizon is large is still unclear.

3.3 Goal Recognition and Goal Recognition Design

As written below, Goal Recognition (GR) is a sub-problem of Plan Recognition: the objective of GR is to find out what goal the agent is trying to pursue by observing his actions collected online. The main difference is that, unlike the PR, the Goal recognition only looks for the terminal goal, while the plan used to achieve this is somewhat irrelevant.

Goal Recognition Design (GRD) is a different task: it offers an offline evaluation of the action field of an agent, providing a solution that does not include a specification on observations. It provides solutions for assessing and decreasing the number of observations needed to recognize the goal of any optimal agent. Moreover, GRD is relevant in any application where goal recognition needs to be performed quickly.

From a more practical point of view, the task of GRD is to study how the actions of an intelligent agent moving in a certain environment reveal its final goal. Therefore GRD tries to modify the environment so that the agent reveals its goals as soon as possible or as late as possible, depending on the application and without distorting the environment. In this way, by controlling the model design, GRD facilitates (or hinders) Goal Recognition.

Recent contributions to the GRD have been made: a new measure has been introduced, the *worst case distinctiveness* (*wcd*), that assesses the Goal Recognition Design models [11]. This new measure represents the "maximal length of a prefix of an optimal path an agent may take within a system before it becomes clear at which goal it is aiming" [11].

The GRD problem can be solved by using tools and methods belonging to the automated planning field: in particular, two methods, based on planning tools, are useful to calculate the *wcd* and are presented in Keren et al. discussion (2014) [11]. Recalling the concepts of planing theory presented above, we affirm that the goal is to find a plan π whose sequence of actions brings an agent from an initial state to a state that satisfies the goal. The cost of the plan, as before, corresponds to the sum of the costs of the individual actions of the plan. For simplicity, it is often assumed that the cost of each action is unitary.

By starting to three assumptions on which the considered model is based, we can formulate the problem of GRD. The three assumptions we have to take into account are that the agents act optimally in the system, the results of the actions are deterministic and the model is fully observable to the system and agents [11].

So, we now explain how to find the *wcd* index. Before to present the methods, it is necessary to define some concepts.

In particular, given a problem $D = \langle P, \mathcal{G} \rangle$, we define a plan π as a non-distinctive path in D iff $\exists G', G'' \in \mathcal{G}$ such that the two goals are different and $\exists \pi' \in \prod^*(G')$ and $\pi'' \in \prod^*(G'')$ such that π is a prefix of π' and π''. Notice that $\prod^*(G)$ is the set of optimal paths to goal G [11].

So using the above definition, we can define the *wcd* of a model D as

$$wcd(D) = \max_{\pi \in \prod_D} |\pi| \tag{3.1}$$

where $\prod_D = \{\pi | \pi \text{ is a non-distinctive path of } D\}$ and $|\pi|$ is the length of a path π [11].

It is possible to affirm that given an initial state, any optimal agent elaborates its starting movement strategy by following a non-distinctive path and ends it with a distinctive path that

leads to its goal [11]: it is evident the correspondence between goal recognition problem defined by a model $D = \langle P, \mathcal{G}$ and plan recognition problem presented in section 3.2 ([9]) whose objective is to add in D the set of observations O.

Going back to the calculation of the *wcd* index, two ways have been analyzed; the first one consist on exploring the state space using a modified version of BFS tree search, the second one is called latest-split [11].

BFS search base its strategy on pruning the nodes that represent distinctive paths. Practically, if the analysed node does not represent a distinctive path, it is added to a queue (the items in the queue represent nodes that still need to be explored), otherwise, it is marked as solved and it is not expanded. When the queue is empty, the search stops. The *wcd* value will be the length of the sub-paths expanded during the last iteration of the execution. Concerning the traditional version, this method uses pruning in order to avoid the expansion of useless states. On the other hand, using the planing techniques for each node in order to understand if it is necessary to expand it or not can be expensive from an efficiency point of view. So limits are set on the model's wcd.

The second method used is the $latest - split$ method: it performs a single search for each pair of goals. This allows compiling the original GRD problem into a classical planning problem. In particular, each problem P is solved by using classical planning tools that produce the optimal plan in with each agent i has to achieve its goal G_i. The method looks for the maximal non-distinctive path that two or more agents, aiming at different goals, may share. The splitting action has no additional cost. The resulting *wcd* of the model is the length of the action sequence until the splitting action occurs [11].

To modify the design of the model, it is possible, for example, to reduce reduce the *wcd* index. Practically, we can modify *wcd* only by removing paths from the optimal set of paths, so by removing possible actions from the model [11].

Two methods can be used to reduce *wcd*: they are the $exhaustive - reduce$ method and the $pruned - reduce$ method. Both are based on the idea that the only actions that can be removed from the original set are those that appear in optimal plans to the goals. Moreover, it has been demonstrated that, by removing the actions, it is possible to reduce the value of *wcd* according to the methods mentioned, without increasing the value of the optimal cost of the plan to any goals.

What has been written so far is an example of how Goal Recognition Design can act on Goal Recognition, facilitating or hindering it. There are a lot of applications in which the task is to hinder the GR, for example when you want to preserve user privacy. Assuming, for example, the problem of searching for an agent that moves in an environment, according to its intentions. It is the particular case in which a moving agent, called *target*, is *uncooperative* with another agent, called observer, i.e. it chooses efficient paths, trying to keep hidden its final destination.

On the contrary, in other applications we want to facilitate Goal Recognition, i.e. in all situations in which an agent interacts with a human user (think of a robot that works with a human operator, or a chatbox that interacts with a user), etc. In this case, it is necessary to use different approaches.

Based on the studies on GR and GRD, this thesis takes into account the first scenario, so we have two agents, a *fleeing target* and an *observer*. In this context, the observer runs Goal Recognition algorithms that aim to discover where the target is going. In particular, the observer needs to reason about the target's possible strategies. To support this kind of reasoning, we proposed a new measure, the "undisclosing index", which represents the maximum length of the path that a fleeing agent may take before its goal becomes apparent to the observer. Further details are presented in the next chapter.

Chapter 4

Statement of the problem

4.1 Introduction

In this section, we propose a concrete instance of a goal recognition problem formulated as a precise mathematical optimization problem.

We analyze an evasive or fleeing target that moves in the environment to reach its destination, while the observer tries to find it. An example of this scenario is a surveillance drone that flies over a wide geographical area to discover a criminal who is trying to reach a hideout by car as soon as possible. In this context, the observer runs Goal Recognition algorithms that aim to discover where the target is going. In particular, the observer needs to reason about the target's strategy to obscure its goals. To support this kind of reasoning, we propose a technique to calculate a new measure, the *undisclosing index*, that represents the maximal length of the prefix of a path that a fleeing agent may take before its goal becomes apparent to the observer. Moreover, we present methods for the observer to reason about how the target might change its behaviour based on the resources that it has available: the target will take shorter or longer paths based on his total travel budget. In this sense, a way to maximize the *undisclosing index* is described, using an algorithm based on the well-known Dijkstra and Yen's algorithms.

Note also that we modelled the environment in which the agents move by using a connected graph.

4.2 Abstract formulation of goal recognition

In this section, we provide a first mathematical representation of the environment in which each agent moves.

In particular, to represent the topology of the system, we build a directed graph $\mathcal{G}$ represented by the triple $(\mathcal{V}, \mathcal{E}, \mathcal{I}_{\mathcal{G}})$. We are given an initial node $o \in \mathcal{V}$ and a set of destinations nodes $\mathcal{D} \subseteq \mathcal{V}$. Consider now a set $\mathcal{P}$ of paths on $\mathcal{G}$ from o to the various destination nodes. We say that $\mathcal{P}$ is $(o, \mathcal{D})$-connecting if for every $d \in \mathcal{D}$ there is at least one path in $\mathcal{P}$ connecting o to d.

We consider the map $\delta : \mathcal{P} \to \mathcal{D}$ such that $\delta(\gamma)$ is the destination node of γ. For every node $d \in \mathcal{D}$, let $\mathcal{P}(d) = \delta^{-1}(d)$ be the subset of paths in $\mathcal{P}$ connecting o to d. We denote by $\mathcal{P}^k$ the set of prefixes of length k of all the paths in $\mathcal{P}$. If $\gamma \in \mathcal{P}$, the prefix of γ of length k is denoted by γ^k

Definition 3. Given a path $\gamma \in \mathcal{P}$ and a positive integer t, we say that γ is t-undisclosing $\mathcal{D}$ if there exists $\gamma' \in \mathcal{P}$ such that

- $\gamma^t = \gamma'^t$

- $\delta(\gamma) \neq \delta(\gamma')$

The definition just presented is useful to introduce the key concept of our research, that is the *undisclosing index*.

Definition 4. We define the following indices:

- The undisclosing index of γ is the maximum t for which γ is t-undisclosing $\mathcal{D}$ and is denoted by $t_{\gamma,\mathcal{P}}$
- The undisclosing index of $d \in \mathcal{D}$ is

$$t_{d,\mathcal{P}} = \min_{\gamma \in \mathcal{P}(d)} t_{\gamma,\mathcal{P}} \tag{4.1}$$

- The undisclosing index of $\mathcal{D}$ is

$$t_{\mathcal{P}} = \min_{\gamma \in \mathcal{P}} t_{\gamma,\mathcal{P}} \tag{4.2}$$

Remark 5. We notice that $t_{\gamma,\mathcal{P}}$ is monotone with respect to $\mathcal{P}$. Namely, if $\mathcal{P}_1 \subseteq \mathcal{P}_2$ are two $(o,\mathcal{D})$-connecting set of paths, then

$$t_{\gamma,\mathcal{P}_1} \leq t_{\gamma,\mathcal{P}_2} \tag{4.3}$$

However, $t_{\mathcal{P}}$ does not exhibit any type of monotonicity. Increasing $\mathcal{P}$ the undisclosing index could increase but also decrease as it is shown in the following example (Fig. 4.1):

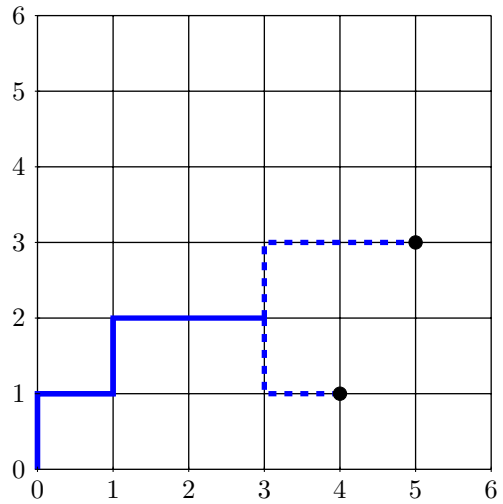


Figure 4.1. .

From the origin (0,0), two possible paths branch out, one for each destination $d \in \mathcal{D}$. Using the definitions given above and considering the simple case analyzed, it is trivial to infer that $t_{\mathcal{P}}$ is equal to 5. In fact, it is evident that for a number of steps greater than 5, the final destination is revealed.

Now we can imagine adding a further path to the set $\mathcal{P}$ (Fig. 4.2):

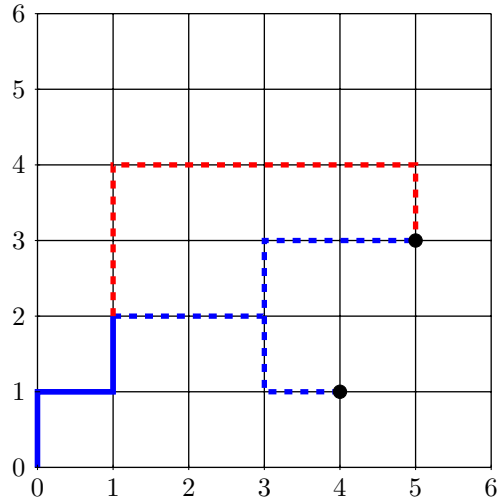


Figure 4.2. .

In this second case, it is clear that the minimum index for which the final destination remains unrevealed is 3, i.e. $t_{\mathcal{P}} = 3$. For this reason, it is possible to state that the function t does not show any type of monotonicity.

The following simple fact holds true:

Proposition 6. Let $\mathcal{P}_1, \mathcal{P}_2$ be two $(o, \mathcal{D})$-connecting set of paths. Then,

$$t_{\mathcal{P}_1 \cup \mathcal{P}_2} \geq \min\{t_{\mathcal{P}_1}, t_{\mathcal{P}_2}\} \tag{4.4}$$

Proof : We prove the proposition using the previously reported definition 4 of undisclosing index of $\mathcal{D}$, with $\mathcal{D}$ a set of destinations nodes in $\mathcal{V}$:

$$t_{\mathcal{P}_1 \cup \mathcal{P}_2} \triangleq \min_{\gamma \in \mathcal{P}_1 \cup \mathcal{P}_2} t_{\gamma, P_1 \cup \mathcal{P}_2} = \min\left\{\min_{\gamma \in P_1} t_{\gamma, \mathcal{P}_1 \cup \mathcal{P}_2}, \min_{\gamma \in P_2} t_{\gamma, \mathcal{P}_1 \cup \mathcal{P}_2}\right\} \tag{4.5}$$

It is possible to note that, by joining the two sets of paths $\mathcal{P}_1$ and $\mathcal{P}_2$, the obtaining undisclosing index can only increase, ie:

$$\min_{\gamma \in P_1} t_{\gamma, \mathcal{P}_1 \cup \mathcal{P}_2} \geq \min_{\gamma \in P_1} t_{\gamma, \mathcal{P}_1} \tag{4.6}$$

and

$$\min_{\gamma \in P_2} t_{\gamma, \mathcal{P}_1 \cup \mathcal{P}_2} \geq \min_{\gamma \in P_2} t_{\gamma, \mathcal{P}_2} \tag{4.7}$$

Therefore, using the inequalities just obtained, we derive the following formulation:

$$\min\left\{\min_{\gamma \in P_1} t_{\gamma, \mathcal{P}_1 \cup \mathcal{P}_2}, \min_{\gamma \in P_2} t_{\gamma, \mathcal{P}_1 \cup \mathcal{P}_2}\right\} \geq \min\left\{\min_{\gamma \in P_1} t_{\gamma, \mathcal{P}_1}, \min_{\gamma \in P_2} t_{\gamma, \mathcal{P}_2}\right\} \tag{4.8}$$

Furthermore, recalling the nomenclature already used previously

$$\min_{\gamma \in \mathcal{P}_1} t_{\gamma,\mathcal{P}_1} = t_{\mathcal{P}_1} \tag{4.9}$$

$$\min_{\gamma \in \mathcal{P}_2} t_{\gamma,\mathcal{P}_2} = t_{\mathcal{P}_2} \tag{4.10}$$

it is trivial to state the above proposition, namely:

$$t_{\mathcal{P}_1 \cup \mathcal{P}_2} \geq \min\{t_{\mathcal{P}_1}, t_{\mathcal{P}_2}\} \tag{4.11}$$

4.3 Optimization of the undisclosing index

In this section we try to modify the model of the environment in which the agents move, trying to maximize the probability that the target has to escape. In fact, it tries to keep its destination hidden as long as possible, making some deviations from the shortest path. The objective of this section is, therefore, to calculate a set of optimal paths, in which, in addition to the shorter paths, possible deviations are also inserted, taking into account, however, the limited budget of the fleeing agent.

Suppose the graph $\mathcal{G}$ has no parallel edges and is weighted, namely we have a weight matrix $W \in \mathbb{R}_+^{\mathcal{V}\times\mathcal{V}}$: if i, j are two adjacent nodes in $\mathcal{V}$, W_{ij} is the weight of the edge connecting the two nodes i, j: it denotes a Euclidean distance. Starting from W, we choose to use an array of adjacency lists[1] to store the relationships between nodes. Assuming to analyze a sparse graph with millions of nodes, representation through adjacency lists can be considered an efficient choice in terms of storage.

To each path $\gamma = (v_1, v_2, \ldots, v_k)$ we can associate the global weight: $|\gamma| = \sum_{i=1}^{k-1} W_{v_i,v_{i+1}}$. As previously written, we can imagine a moving target that needs to reach any of the destinations in $\mathcal{D}$ starting from o and wants to maintain its destination un-revealed as long as possible at the price of possibly deviating from the shortest path.

Given o and $d \in \mathcal{D}$, we define $\lambda(o, d)$ as the geodesic distance from these two nodes, namely the shortest weight of any path from o to d. So we consider the set of shortest paths from o to the various destinations

$$\mathcal{P}^{opt} = \{\gamma \,\text{path from}\, o \,\text{to}\, d, \, ||\gamma| = \lambda(o, d)\} \tag{4.12}$$

The goal is, as already specified, to expand the set of optimal path, in order to simplify the escape of the target. It will, therefore, be able to choose between a wider range of escape routes, taking into account its limited resources, indicated using an index r. Below is a more rigorous formulation of what has just been said.

$$\mathcal{P}^{r-opt} = \{\gamma \,\text{path from}\, o \,\text{to}\, d, \, ||\gamma| \leq r\lambda(o, d)\} \tag{4.13}$$

where $r \geq 1$ and $\mathcal{P}^{r-opt}$ is a suboptimal families of paths.

In this section, two specific optimisation problems are of interest. In the first problem, we fix a budget $\bar{r}$ and we compute the maximum undisclosing index achievable within that budget, while in the second problem we fix a desired undisclosing index $\bar{t}$ and we compute the minimum budget needed to achieve such performance.

[1] See the section on Graph theory 2.2

Note that only the first problem will be treated and analyzed in this discussion, considered the cornerstone of much other research work related to the Goal Recognition Design.

4.3.1 Maximum undisclosing index achievable within a set of paths

We first formulate the optimisation problem in a slightly more general form than that proposed above. Precisely, we are given a set of paths $\bar{\mathcal{P}}$ that is $(o - \mathcal{D})$-connecting and we define:

$$T_{\bar{\mathcal{P}}} := \max\{t_{\mathcal{P}} \,|\, \mathcal{P} \subseteq \bar{\mathcal{P}}\ (o, \mathcal{D}) - \text{connecting}\} \tag{4.14}$$

In general, the maximum in (4.14) is not unique. If $\mathcal{P}_1, \mathcal{P}_2 \subseteq \bar{\mathcal{P}}$ are two set of paths $(o, \mathcal{D})$-connecting and achieving the maximum above, then $\mathcal{P}_1 \cup \mathcal{P}_2$ is also $(o, \mathcal{D})$-connecting and, by Proposition 6:

$$t_{\mathcal{P}_1 \cup \mathcal{P}_2} \geq \min\{t_{\mathcal{P}_1}, t_{\mathcal{P}_2}\} = T_{\bar{\mathcal{P}}} \tag{4.15}$$

Hence, $t_{\mathcal{P}_1 \cup \mathcal{P}_2} = T_{\bar{\mathcal{P}}}$ that means that also using $\mathcal{P}_1 \cup \mathcal{P}_2$ we achieve the maximum in (4.14). A straightforward consequence of this fact is that among the maxima, there exists a largest set of paths achieving the maximum: it will be denoted $\mathcal{P}_*$. Note that $\mathcal{P}_*$ represents the set of possible paths the target can choose. Expanding this set means increasing the set of available actions the target can perform: this makes its movement more difficult to predict.

In order to pursue the goals set in this section, we apply the ideas presented so far, starting from $\bar{\mathcal{P}} = \mathcal{P}^{r-opt}$. Coherently, we use the notation $\mathcal{P}_*^{r-opt}$ to denote the largest subset of $\mathcal{P}^{r-opt}$ maximizing the undisclosed index and we define $T(\bar{r}) = T_{\mathcal{P}^{r-opt}}$ that is the maximum undisclosing index obtainable with the budget $\bar{r}$.

Below, we present an algorithm that computes $\mathcal{P}_*^{r-opt}$ and $T(\bar{r})$. The algorithm starts its search by considering the entire network of nodes. Therefore, algorithms for finding the shortest path are applied to the whole graph. Once the optimal set of paths has been obtained, the model is modified by removing optimal paths that are not considered relevant in order to maximize the undisclosing indexing of the model. In other words, the largest sub-set of optimal paths is created, whose undisclosing index is at least equal to the maximum undisclosing index of the obtained subset.

We, therefore, proceed with a clearer explanation of the algorithm in question. This is composed of two parts:

(1) Given the graph $\mathcal{G}$ representing the analysed environment, the weight matrix W, the starting node (the *source*) o and the set of destinations $\mathcal{D}$, we compute the sets $\mathcal{P}^{opt}$ and $\mathcal{P}^{r-opt}$.

The set $\mathcal{P}^{opt}$ of shortest paths from o to the various destinations are easily obtained using the Dijkstra's algorithm [2].

The suboptimal families of paths, $\mathcal{P}^{r-opt}$, given a specific maximum budget $\bar{r}$, are returned by the implementation of the Yen's algorithm [3]. It is a generalization of the shortest path routing problem in a given network, also called K-shortest path problem: it provides not only the shortest path k but also the next $k1$ shortest paths (which may be longer than the first shortest path). Essentially, the Yen's algorithm finds the k shortest paths by extending the Dijkstra algorithm.

Both algorithms are widely used to solve route planning problems in a network.

[2] See the section on Dijkstra's algorithm 2.3.1

[3] See the section on Yen's algorithm 2.3.2

(2) Given any $\bar{\mathcal{P}}$, that is $(o\text{–}\mathcal{D})$-connecting and was obtained from the previous step ($\bar{\mathcal{P}} = \mathcal{P}^{r-opt}$), we compute $\bar{\mathcal{P}}_*$ and $T_{\bar{\mathcal{P}}}$.

The implemented algorithm is an iterative one, therefore, using an index k, we define a generic set of paths $\mathcal{Q}_{k-1}$ as follows:

$$\mathcal{Q}_{k-1} = \{\gamma \in \bar{\mathcal{P}} | t_{\gamma,\bar{\mathcal{P}}} \geq k-1\} \tag{4.16}$$

It is trivial to note that, starting from $k = 1$:

$$\mathcal{Q}_0 \equiv \bar{\mathcal{P}} \tag{4.17}$$

For each k, the algorithm implements the two following steps:

1. Identify the subset of paths in $\mathcal{Q}_{k-1}$ whose undisclosing index is equal to $k-1$, that is

$$\mathcal{Q}'_{k-1} = \{\gamma \in \mathcal{Q}_{k-1} | t_{\gamma,\bar{\mathcal{P}}} = k-1\} \tag{4.18}$$

2. Remove from $\mathcal{Q}_{k-1}$ the paths whose undisclosing index is equal to $k-1$, so

$$\mathcal{Q}_k = \mathcal{Q}_{k-1} \setminus \mathcal{Q}'_{k-1} \tag{4.19}$$

Two cases arise:

(i) If $\mathcal{Q}_k$ is not $(o-\mathcal{D})$-connected

$$\max_{\substack{\mathcal{S}\subseteq\mathcal{Q}_{k-1} \\ \mathcal{S}(o-\mathcal{D})conn.}} t_{\mathcal{S}} = k-1 \tag{4.20}$$

In this case, the execution stops. Therefore it is necessary to identify the set of paths $\mathcal{R} \subseteq Q_{k-1}$ whose removal made the graph not connected. From a mathematical point of view, the set $\mathcal{R}$ at the generic iteration k is

$$\mathcal{R} = \{\gamma \in \mathcal{Q}'_{k-1} | \mathcal{Q}_k(d_\gamma) = \varnothing\} \tag{4.21}$$

considering d_γ the destination node of the generic path $\gamma \in \mathcal{Q}'_{k-1}$.
So we get P_*^{r-opt} as follows:

$$\mathcal{P}_*^{r-opt} = \mathcal{Q}_k \cup \mathcal{R} \tag{4.22}$$

(ii) If $\mathcal{Q}_k$ is $(o-\mathcal{D})$-connected

$$\max_{\substack{\mathcal{S}\subseteq\mathcal{Q}_{k-1} \\ \mathcal{S}(o-\mathcal{D})conn.}} t_{\mathcal{S}} \geq k \tag{4.23}$$

and the largest subset inside $\bar{\mathcal{P}}$ that is $(o-\mathcal{D})$ – *connected* and whose undisclosing index is at least k is $\mathcal{Q}_k$. In this last case, the value of k is increased and the algorithm is repeated starting from the step 1..

It is essential to describe step 1. in more detail, explaining how to calculate the undisclosing index $t_{\gamma,\mathcal{Q}_{k-1}}$ in a more practical way.

In this context, a few additional definitions will be helpful.

Let $col_k[\gamma]$ the k^{th} element of the path γ, with γ contained in the set $\mathcal{Q}_{k-1}$, i.e.

$$col_k[\gamma] = \gamma[k], \forall \gamma \in \mathcal{Q}_{k-1}, k \in \mathbb{N} \tag{4.24}$$

Thus, the column col_k is the vector of nodes contained in $\mathcal{Q}_{k-1}$ at the k^{th} position.

Moreover, let $\mathcal{V}_k(\gamma)$ be the prefix of length k of path γ, i.e. the subsets of nodes of the path γ visited at the k^{th} step.

Taking into consideration the notation just presented, we evaluate the generic cycle k: the algorithm extrapolates and analyses the corresponding column col_k, evaluating node by node of the aforementioned column.

For each element contained in col_k, we look for the $col_k[\gamma']$ such that:

$$col_k[\gamma] = col_k[\gamma'] \tag{4.25}$$

being γ' any path in $\mathcal{Q}_{k-1}$ connecting o to d', $\forall d' \in \mathcal{D}$ and $d' \neq d$. If there is no γ' such that the aforementioned equation is valid, then

$$t_{\gamma,\mathcal{Q}_{k-1}} = k - 1 \tag{4.26}$$

Otherwise, if the (4.25) is valid, it is necessary to verify that

$$\mathcal{V}_k(\gamma) \equiv \mathcal{V}_k(\gamma') \tag{4.27}$$

As written above, the subset $\mathcal{V}_k(\Gamma)$, with Γ a generic path in $\mathcal{Q}_{k-1}$, is the prefix of path Γ of length k: it is returned by the function $calculatePref$ (See pseudocode of $calculatePref$ function (1)).

So if the equivalences (4.25) and (4.27) are verified, $t_{\gamma,\mathcal{Q}_{k-1}}$ will be greater than or equal to k, so the algorithm proceeds with the analysis of the next element in col_k, until the entire column is analysed.

For greater clarity, the following example is presented, assuming to consider the case for $k = 1$:

Input $\mathcal{Q}_{k-1} = \mathcal{Q}_0 \equiv \bar{\mathcal{P}}$ ($\mathcal{Q}_{k-1}$ computed in the previous cycle)	0 1 3 8 0 2 4 5 8	γ conn. o to d_1
	0 2 4 9 0 7 10 9	γ conn. o to d_2
	0 1 4 6	γ conn. o to d_3
Identification of the set $Q'_{k-1} = Q'_0$: 1. consider the k-th column, $col_k = col_1$	0 **1** 3 8 0 **2** 4 5 8	γ conn. o to d_1
	0 **2** 4 9 0 **7** 10 9	γ conn. o to d_2
	0 **1** 4 6	γ conn. o to d_3
2. for each element $col_k[\gamma]$ in col_k, look for the node $col_k[\gamma']$ such that:		

• $col_k[\gamma] = col_k[\gamma']$	0 **1** 3 8 0 **2** 4 5 8	γ conn. o to d_1
	0 **2** 4 9 0 **7** 10 9	γ conn. o to d_2
	0 **1** 4 6	γ conn. o to d_3
• $\mathcal{V}_k(\gamma) \equiv \mathcal{V}_k(\gamma')$ $\forall \gamma'$ connecting o to d', $\forall d' \in \mathcal{D}$ and $d' \neq d$	0 **1** 3 8 0 **2** 4 5 8	γ conn. o to d_1
	0 **2** 4 9 0 **7** 10 9	γ conn. o to d_2
	0 **1** 4 6	γ conn. o to d_3
So $Q'_{k-1} = Q'_0$ is: (The definition of Q'_{k-1} was given previously in (4.18))	0 7 10 9	

Output $Q_1 = Q_0 \setminus Q'_0$ $(Q_k = Q_{k-1} \setminus Q'_{k-1})$	0 1 3 8 0 2 4 5 8	γ conn. o to d_1
	0 2 4 9	γ conn. o to d_2
	0 1 4 6	γ conn. o to d_3

Check if the resulting output $Q_k = Q_1$ is connected:
$\rightarrow Q_1$ is connected: $\Longrightarrow \mathcal{T}_{\bar{\mathcal{P}}} \geq 1$
(Q'_k is connected $\Longrightarrow \mathcal{T}_{\bar{\mathcal{P}}} \geq k$)
Repeat all the steps for $k = 2$ **(**$k = k + 1$**)**

Now we consider the case for $k = 2$:

Input $\mathcal{Q}_{k-1} = \mathcal{Q}_1$	0 1 3 8 0 2 4 5 8	γ conn. o to d_1
	0 2 4 9	γ conn. o to d_2

	0 1 **4** 6	γ conn. o to d_3
Identification of the set $Q'_{k-1} = Q'_1$:		
1. consider the k-th column, $col_k = col_2$	0 1 **3** 8 0 2 **4** 5 8	γ conn. o to d_1
	0 2 **4** 9	γ conn. o to d_2
	0 1 **4** 6	γ conn. o to d_3
2. for each element $col_k[\gamma]$ in col_k, look for the node $col_k[\gamma']$ such that:		
• $col_k[\gamma] = col_k[\gamma']$	0 1 **3** 8 0 2 **4** 5 8	γ conn. o to d_1
	0 2 **4** 9	γ conn. o to d_2
	0 1 **4** 6	γ conn. o to d_3
• $\mathcal{V}_k(\gamma) \equiv \mathcal{V}_k(\gamma')$ $\forall\gamma'$ connecting o to d', $\forall d' \in \mathcal{D}$ and $d' \neq d$	0 1 **3** 8 0 2 **4** 5 8	γ conn. o to d_1
	0 2 **4** 9	γ conn. o to d_2
	0 1 **4** 6	γ conn. o to d_3
So $Q'_{k-1} = Q'_1$ is:	0 1 2 8 0 1 4 6	
Output $Q_2 = Q_1 \setminus Q'_1$	0 2 4 5 8	γ conn. o to d_1
	0 2 4 9	γ conn. o to d_2
	\	γ conn. o to d_3
Check if the resulting output $Q_k = Q_2$ **is connected:** $\rightarrow Q_2$ is not connected: $\Longrightarrow \mathcal{T}_{\bar{\mathcal{P}}} = 1$ ($\mathcal{T}_{\bar{\mathcal{P}}} = k-1$) **Stop the execution.**		
Identify the set $\mathcal{R}$ (see section (i)) and obtain $\bar{\mathcal{P}}_* \rightarrow \bar{\mathcal{P}}_* = \mathcal{Q}_k \cup \mathcal{R} = \mathcal{Q}_2 \cup \mathcal{R}$	0 1 4 6	$\mathcal{R}$
	0 2 4 5 8	

0	2	4	9	$\bar{\mathcal{P}}_*$
0	1	4	6	

If execution is not interrupted, the algorithm iterates k up to a maximum value k_{max}, limited by the number of vertices $\mathcal{V}$ in $\mathcal{G}$.

In particular

$$k_{max} = \mathcal{V} - 3 \tag{4.28}$$

This formulation is due to the following: the best case, which is represented in the figure 4.3 and which coincides with the maximisation of $T_{\bar{\mathcal{P}}}$, consists of a graph in which there are only two destinations and that branches off only at the end.

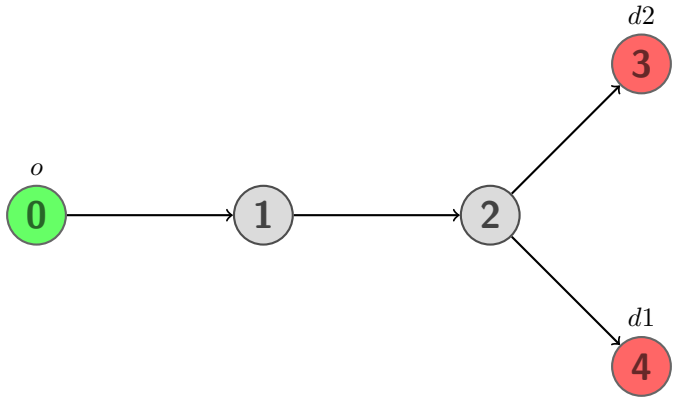


Figure 4.3. .

In the simple considered case

$$k_{max} = \mathcal{V} - 3 = 5 - 3 = 2 \tag{4.29}$$

Pseudocode

The following is the pseudocode of the implemented algorithm. It consists of three functions:

(1) *calculatePref* (Alg. (1)): given a destination $d \in \mathcal{D}$, and the initial set of optimal paths, $\bar{\mathcal{P}}$, it computes the sets of prefixes *prefList* of the paths belonging to $\bar{\mathcal{P}}(d)$. In particular, the computed output *prefList* is an array where in position i there is a list of prefixes of length $i+1$ of paths to d. The *calculatePref* function is used in algorithm (3), line 3: for each destination $d \in \mathcal{D}$, it initializes the value of *prefDict*. It is used in subsequent lines to compare the same-length prefixes of paths belonging to different destinations.

Algorithm 3 $calculatePref(d, \bar{\mathcal{P}})$

Input: d (destination in $\mathcal{D}$), $\bar{\mathcal{P}}$ (dictionary that maps each destination d to a set of path to d)
Output: *prefList* (array where in position i there is a list of prefixes of paths to d of length $i+1$)

1: $maxSize\_d = maxLength\{p \in \bar{\mathcal{P}}[d]\}$
2: **for** $i = 0$ to $maxSize\_d - 2$ **do**
3: $\quad prefList[i] = \bar{\mathcal{P}}[d][0 \text{ to } i+1]$
4: **end for**
5: **return** *prefList*

(2) *retPaths*(Alg. (2)): this function takes as input the set of optimal paths $\bar{\mathcal{P}}$, the prefix pr and the destination d of a specific path. It returns a list of paths (*paths*) belonging to $\bar{\mathcal{P}}(d)$ having pr as prefix. Given a destination $d \in \mathcal{D}$, the function searches through the paths in $\bar{\mathcal{P}}(d)$, those whose prefix of length k is equal to the prefix pr, so it adds them to the *paths* list returned as output.

Algorithm 4 $retPaths(\bar{\mathcal{P}}, pr, d)$

Input: $\bar{\mathcal{P}}$ (initial set of paths), pr (prefix of a generic path), d (destination in $\mathcal{D}$)
Output: list of paths in $\bar{\mathcal{P}}[d]$ having pr as prefix

1: $sizePref = pr.size$
2: $paths.clear$
3: **for** $i = 1$ to $(\bar{\mathcal{P}}(d).size)$ **do**
4: $\quad$ **if** $\bar{\mathcal{P}}(d)[i-1][0 \text{ to } sizePref] == pr$ **then**
5: $\quad\quad paths.add(\bar{\mathcal{P}}(d)[i-1])$
6: $\quad$ **end if**
7: **end for**
8: **return** *paths*

(3) *calculateIndexAndPaths* (Alg. (3)): it is the main function and takes as inputs the set of destinations, $\mathcal{D}$, the initial set of optimal path, $\bar{\mathcal{P}}$, the maximum value that the iteration index k can assume, k_max, and the total number of nodes, n. The function returns as output the maximum undisclosing index achievable with the fixed budget, $\mathcal{T}_{\bar{\mathcal{P}}}$. Moreover it provides the largest subset $\mathcal{P}_*^{r-opt} \subseteq \bar{\mathcal{P}}$, that is $(o - \mathcal{D})$-connected and that guarantees the obtained $\mathcal{T}_{\bar{\mathcal{P}}}$.

Note that in this pseudocode we use the temporary set of paths $\mathcal{Q}$, whose final value coincides with the largest optimal subset $\mathcal{P}_*^{r-opt}$ we are looking for. At the beginning, we impose the set $Q = \bar{\mathcal{P}}$. Its value is updated during each iteration k.

For each cycle k, the algorithm evaluate one by one the destination in $\mathcal{D}$. So for each destination $d \in D$, it compares the prefix pr of length k of each paths in $\mathcal{Q}(d)$ with the ones of the paths in $\mathcal{Q}(d')$, having the same length k. Notice that $d' \in D$ and $d' \neq d$ (See the for-loop at line 11). If there is no prefix in $\mathcal{Q}(d')$ equal to pr, it is necessary to insert the path whose prefix is pr in the so called *pathsRemoved*. When all the prefixes of length k of paths in $\mathcal{Q}(d)$ was analysed, the algorithm checks if $\mathcal{Q}(d)$ is empty. If so, i.e. if Q is not $(o - d)$-connected, it is necessary to re-insert in $\mathcal{Q}(d)$ the set of paths removed from it at the k^{th} step (stored in *pathsRemoved*). Note that, in this case, the algorithm returns the searched values $\bar{\mathcal{P}}_*$ and $\mathcal{T}_{\bar{\mathcal{P}}}$ (See pseudocode at line 37): the first one will be equal to the obtained Q set, while the second one will be simply equal to $k - 1$, as already discussed above (See case (i), at (4.20). At this point the algorithm stops its execution. On the contrary, if $\mathcal{Q}(d)$ is not empty, the algorithm deletes the content of the *pathsRemoved* list, proceeding with the analysis of the next destination in $\mathcal{D}$ (for-loop at line 7).

If the algorithm continues its execution up to the iteration $k = k_{max}$ and if, for each $d \in \mathcal{D}$, $\mathcal{Q}(d)$ is non-empty, the two value at line 45 are returned: in particular, $\bar{\mathcal{P}}_* = \mathcal{Q}$, as in the previous case, while $\mathcal{T}_{\bar{\mathcal{P}}} = k = k_{max}$. So the algorithm stops its execution and a practical modification of the initial model is obtained.

Algorithm 5 *calculateIndexAndPaths*$(\mathcal{D}, \bar{\mathcal{P}}, k_{max}, n)$

Input: $\mathcal{D}$ (set of destinations), $\bar{\mathcal{P}}$ (dictionary that maps each destination d to a set of path to d), k_{max} (maximum value for T_P), n (number of nodes in the graph)
Output: $T_{\bar{\mathcal{P}}}$ (max undisclosing index achievable), $\bar{\mathcal{P}}_*$ (modified version of $\bar{\mathcal{P}}$)

$\mathcal{Q} = \bar{\mathcal{P}}$
for all $d \in \mathcal{D}$ **do**
  $prefDict[d] = calculatePref(d, \bar{\mathcal{P}})$
end for
for $k = 1$ to $k_{max} = n - 3$ **do**
  $remove = True$
  for all $d \in \mathcal{D}$ **do**
    $pathsRemoved.clear$
    $L = prefDict[d][k]$
    for all pr in L **do**
      for all $d_i \in \mathcal{D}$ and $d_i \neq d$ **do**
        $L_i = prefDict[d_i][k]$
        if pr in L_i **then**
          $remove = False$
          break
        end if
      end for
      if $remove$ **then**
        $P_{pr} = retPaths(\bar{\mathcal{P}}, pr, d)$
        $Q.remove(P_{pr})$
        $pathsRemoved.add(P_{pr})$
        if not $\mathcal{Q}[d]$ **then**
          $\mathcal{Q}[d].add(pathsRemoved)$
          for all $d_j \in \mathcal{D}$ and $d_j < d$ **do**
            $L_j = prefDict[d][k-1]$
            for all pr_j in L_j **do**
              $paths_j = retPaths(\bar{\mathcal{P}}, pr_j, d_j)$
              for all p in $paths_j$ **do**
                if p doesn't exist in $\mathcal{Q}[d_j]$ **then**
                  $Q[d_j].add(p)$
                end if
              end for
            end for
          end for
          $\bar{\mathcal{P}}_* = \mathcal{Q}$
          $T_{\bar{P}} = k - 1$
          return $T_{\bar{P}}$, $\bar{\mathcal{P}}_*$
        end if
      end if
    end for
  end for
end for
$\bar{\mathcal{P}}_* = \mathcal{Q}$
$T_{\bar{P}} = k$
return $T_{\bar{P}}$, $\bar{\mathcal{P}}_*$

Correctness

We want to be sure that the algorithm works properly and correctly. The correctness of the *calculateIndexAndPaths* algorithm is shown below.

We start by proving the correctness of the *remove* operations in (3). This proof is used to guarantee that the algorithm calculates exactly both $\bar{\mathcal{P}}_*$ and $\mathcal{T}_{\bar{\mathcal{P}}}$, the main goals of this implementation.

Lemma 7. Given a starting set $\bar{\mathcal{P}}$, a destination $d \in \mathcal{D}$, a set of destination $d' \in \mathcal{D}, d' \neq d$, L_i of prefixes of length k of paths in $\bar{\mathcal{P}}(d')$, a prefix $pr \in \bar{\mathcal{P}}(d)$ of length k, at each iteration k ($k \in [1, k_{max}]$) the algorithm individuates only the paths γ whose undisclosing index is

$$t_{\gamma,\bar{\mathcal{P}}} \leq k - 1$$

Proof To prove this lemma, we first consider the base case: for $k < 1$, we should remove from $\bar{\mathcal{P}}$ the paths γ whose undisclosing index is

$$t_{\gamma,\bar{\mathcal{P}}} < 0$$

By definition, the undisclosing index is always a positive value. So, the algorithm individuates correctly the paths whose undisclosing index is less than 0, simply by doing nothing: in fact, we have no paths in $\bar{\mathcal{P}}$ whose undisclosing index is less than 0.

Otherwise, if $k \geq 1$, we have to sought the correctness of the k^{th} iteration at line 13 of the Algorithm (3): it checks if there exists a prefix of length k in L_i, that is equal to the considered prefix pr (belonging to set L, with length k). If there is no prefix in L_i equal to the considered one, pr, recalling the concept of undisclosing index (see section 4), we can state that the path γ, of which pr is prefix of length k, reveals its goal at a step that precedes the $(k)^{th}$ step. So

$$t_{\gamma,\bar{\mathcal{P}}} \leq k - 1$$

Lemma 8. Given a starting set of paths $\mathcal{Q}$, at each iteration k ($k \in [1, k_{max}]$), the algorithm chooses to remove the paths $\gamma \in \mathcal{Q}$, such that their undisclosing index is equal to $k - 1$.

Proof As in the previous case, the base case is $k < 1$ and the algorithm chooses correctly which paths to remove (the paths whose undisclosing index is equal to $k - 1$) by doing absolutely nothing: it does not remove any paths from set $\mathcal{Q}$. This is related to the previous lemma (7), in fact, our specific case

$$t_{\gamma,\mathcal{Q}} = k - 1$$

is contemplated by

$$t_{\gamma,\mathcal{Q}} \leq k - 1$$

So, the algorithm (3), for $k < 1$ will not choose any path $\gamma \in Q$ to be removed.

For the more general case $k \geq 1$, we can use the previous lemma again. According to lemma 7,

$$t_{\gamma,\mathcal{Q}} \leq k - 1$$

In other words,

$$t_{\gamma,\mathcal{Q}} = j - 1$$

with $j = 0, 1, ..., k$. Since the path removal operation we are discussing is inside a for-loop (See line 5) and considering the 4.3.1, we can state that, for each generic step, the algorithm chooses as paths to be removed the paths $\gamma \in \mathcal{Q}$ whose undisclosing index is

$$t_{\gamma,\mathcal{Q}} = k - 1$$

The step discussed is correct.

Theorem 9. Given a set of paths $\mathcal{Q} \subseteq \bar{\mathcal{P}}$, such that at the beginning $\mathcal{Q} \equiv \bar{\mathcal{P}}$, at the generic iteration k (k from 1 to k_{max}), removing from $\mathcal{Q}$ the paths whose undisclosing index is $k-1$ means that the resulting Q is the largest sub-set of $\bar{\mathcal{P}}$ whose undisclosing index is:

$$t_Q \geq k$$

Proof It is simple to affirm that the proposition is valid for the base case $k < 0$: in this case no paths are removed from $\mathcal{Q}$ (See base case of lemma 7) so that $\mathcal{Q}$ is the largest subset of $\bar{\mathcal{P}}$ (in particular the two sets coincide at the beginning). Moreover, because the undisclosing index is always a positive number,

$$t_{\mathcal{Q}} \geq 0$$

So, for $k < 1$, it holds that

$$t_{\mathcal{Q}} \geq k$$

The undisclosing index is always positive.

For the more generic case of $k \in [1, k_{max}]$, we recall the lemma 8. The algorithm correctly chooses which path to remove from $\mathcal{Q}$: so, if at the generic step k it removes from $\mathcal{Q}$ the set of paths whose undisclosing index is $t_{\mathcal{Q}} = k - 1$, then the remaining paths in $\mathcal{Q}$ will have:

$$t_{\mathcal{Q}} < k - 1 \qquad \text{and} \qquad t_{\mathcal{Q}} \geq k$$

Since the operation is at the inner of a for-loop (See line 5), the algorithm will remove at each step k^{th} the paths in Q whose undisclosing index is equal to $k-1$ (See lemma 8), so that the remaining paths, after the execution of the generic cycle k, have undisclosing index equal to or greater then k. The removal operation will return the largest sub-set $\mathcal{Q} \subseteq \bar{\mathcal{P}}$ s.t. $t_{\mathcal{Q}} \geq k$.

Chapter 5

Experimental results

In this chapter, we want to provide a practical application of the analysed problem related to Goal Recognition. In particular, the algorithm presented in the previous chapter has been tested on a series of data representing the real world. A graphical user interface has been created in order to guarantee a simple and direct extrapolation of data belonging to real maps. Moreover, the algorithm, implemented in java language, allows the conversion of map data into a connected graph. It represents the model of the environment we are analysing. So we can use this model to implement some Graph algorithms, like the well-known Dijkstra and Yen's algorithms. The algorithm implemented in chapter 4 will be also tested on the obtained graph representation.

As specified in the previous chapters, in this Goal Recognition problem we analyse the case in which a given target flees from an observer, trying to hide its final goal for as long as possible. Consider putting yourself on the side of the observer: it evaluates a series of possible actions the fleeing agent can perform in the given environment.

Imagine, for example, the scenario in which, at a given instant, the target disappears from the observer's visual radius. So, based on the last known position of the fleeing agent, it must devise a planning strategy aimed at recognized the target's final goal.

The observer, as a user of the web application implemented, selects on a real map an area of interest, so exports it. It must try to identify the target's possible goals within that area. So, the web-app converts the data and provides a viewable version of the corresponding graph, implementing on it some algorithms: this tool helps the observer reason about the possible paths of the target based on its budget and visualize them. In this way, it can devise its plan strategy.

5.1 Graphical user interface design

The application requires spatial information as input. For this reason, the graphical interface takes its data from optimal data source OpenStreetMap: it is a collaborative project to create and export maps and data related to them. In this case, the exported data are used for route planning.

At first glance, the interface appears as in figure 5.1.

The user taps the zoom labels to select the part of the map it is interested in. So it exports the submap in XML format to build the corresponding graph.

It may be useful to provide some additional concepts to better understand how the conversion from the XML file to the graph takes place.

The next sections will explain what the XML format is and how it can be analyzed. Moreover, we will provide some practical concepts related to the XML/graph conversion.

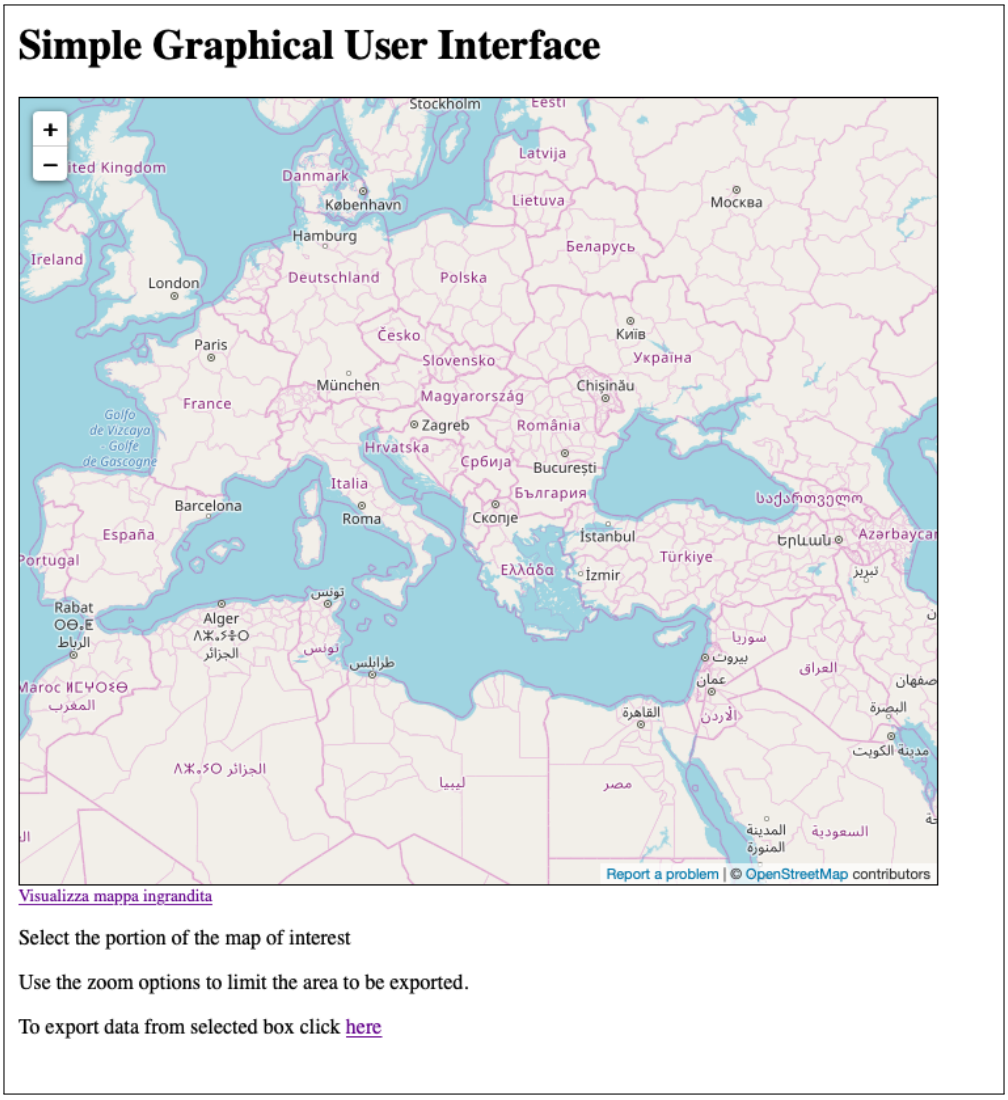


Figure 5.1. A view of the graphical user interface.

5.1.1 XML file

As written in the previous section, an XML file, representing the area selected by the user, is exported by using the web application.

The XML format stores data according to a structure that is machine-readable and human-readable. It is formatted much like an HTML document but uses custom tags to define objects and data within each object.

An OSM XML file is a list of instances of data primitives such as *nodes*, *ways* and *relations*.

A *node* consists of a single point in space, defined by its latitude, longitude and node ID. Other optional characteristics can also be included in the node description (e.g. altitude, layer, level, etc.). These optional features will not be considered in this thesis, as only standard cases will be analysed.

Nodes are used to define the shape of *ways*. A *way* is an ordered list of nodes. It also has tags and can be included within a *relation*. A *way* can have more than a thousand nodes, although faulty ways with zero or a single node may exist. A *way* can be open or closed. In a closed way, the last node is also the first one. There exist different types of way tags that indicate, for example, the type of highway (residential, primary and others), the name of the way, etc. A relevant tag for our application is the one indicating the direction of the way: in particular, if the *oneway* tag is set as *yes*, the way is a 'one-way road'. If *oneway* = *no*, the situation is opposite to the previous one.

Finally, there are *relations*. A *relation* is another of the core data elements of an OSM XML file. It defines logical or geographic relationships between elements of a map. There exist different types of relations (*multi – polygon* type, to represent areas, *bus route*, in which each variation of a bus route itinerary is represented by a relation containing customized tags, and others).

In an OSM XML file we have three types of block:

- block of nodes: each block presents an identification number *id*, latitude *lat* and longitude *lon* tags.

```
5  <node id="619514736" visible="true" version="2" changeset="8049803"
       timestamp="2011-05-04T15:13:02Z" user="David Paleino" uid="71261"
       lat="37.7206181" lon="12.8893198"/>
6  <node id="619514737" visible="true" version="2" changeset="7424168"
       timestamp="2011-02-28T20:36:33Z" user="David Paleino" uid="71261"
       lat="37.7208540" lon="12.8894220"/>
7  <node id="619514751" visible="true" version="2" changeset="8049803"
       timestamp="2011-05-04T15:12:55Z" user="David Paleino" uid="71261"
       lat="37.7228110" lon="12.8894169"/>
```

Figure 5.2. Example of *Node blocks* in a OSM XML file

- block of ways: an *id* tag identifies each way we are considering. Moreover, for each block there is a list of nodes, belonging to a specific way, with a reference number (*ref* tag). These values correspond with their *id*. We see an example in figure 5.3.

```
621  <way id="218287351" visible="true" version="1" changeset="15812766"
       timestamp="2013-04-21T16:08:53Z" user="adirricor" uid="245439">
622   <nd ref="2275519753"/>
623   <nd ref="2275519068"/>
624   <nd ref="2275519535"/>
625   <nd ref="2275519472"/>
626   <nd ref="2275519753"/>
627   <tag k="building" v="yes"/>
628  </way>
```

Figure 5.3. Example of *Way block* in a OSM XML file

- block of relation: each block is identified by an *id* tag, and contains the references to its members (*ref* tag). An example of a relation block is shown in figure 5.4.

```
320  <relation id="2565343" visible="true" version="1" changeset="13815177"
          timestamp="2012-11-09T22:25:56Z" user="David Paleino" uid="71261">
321   <member type="way" ref="111711567" role="street"/>
322   <member type="node" ref="2005418136" role="house"/>
323   <tag k="name" v="Via Giuseppe Garibaldi"/>
324   <tag k="type" v="street"/>
325  </relation>
```

Figure 5.4. Example of *Relation block* in a OSM XML file

5.1.2 Preprocessing

Data obtained from OSM XML file are analysed by the implemented algorithm to obtain the corresponding graph.

The analysis and parsing procedures of the OSM XML file takes into account of different aspects, such as the *node reduction*, the computation of the edge cost (distance between two adjacent nodes), etc.

So we present a list of subsections explaining this parsing preliminary steps.

Node Reduction

Spacial data provided by the OSM XML file describe the Earth: in particular, it is a set of nodes joined by lines.

Imagine the particular case of a bending road, graphically represented by a bending line. To describe this particular shape, it is necessary to use a large number of linked nodes to depict the curvature of the way (See Fig. 5.5 and 5.6)

The two figure represent the map representation (on the left) and its data representation (on the right).

Figure 5.5. Map representation of a bending line

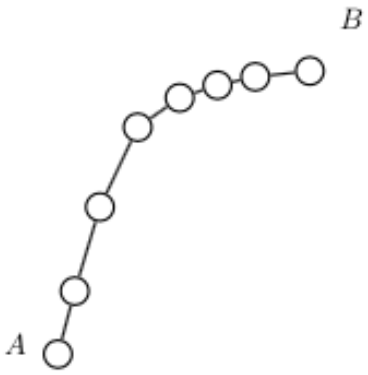


Figure 5.6. Data representation of a bending line

In our discussion, it is not necessary to build a graph whose representation is too accurate: in particular, the shape of its arcs allows less accuracy, in favour of a faster and less heavy conversion and representation of data.

The strategy used to build the graph is based on the following association:

a node *for each* **crossroad**

Notice that a crossroad is, trivially, an intersection between two or more *ways*. In this way, we avoid representing a graph with superfluous nodes and edges (See Fig. 5.7 and 5.8).

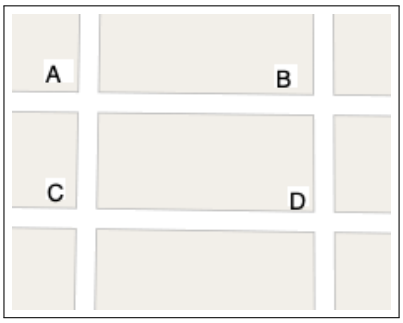


Figure 5.7. Map with four crossroads

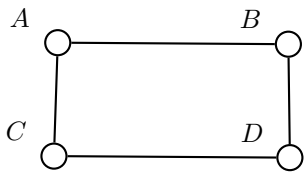


Figure 5.8. Data representation of the map in figure 5.7

The building of a weighted graph

After removing the superfluous nodes, proceed with the assignment of the weight to each edge joining two adjacent nodes.

Data in the OSM XML file are provided, as previously written, as lists of nodes, ways and relations. It does not contain any direct information about nodes distance, way's length, etc. This kind of information is relevant in this dissertation, since we are interested to know the path's length, that is its cost.

Each node has two coordinates, latitude and longitude. Trivially, the latitude is a value from $-90°$ to $90°$, while longitude is a value between $-180°$ and $180°$.

This set of coordinates represents a unique label for every node in the map.

Imagine to have two adjacent nodes, A and B s.t.:

$$A: \qquad lat = LAT_A, \qquad lon = LON_A$$

and

$$B: \qquad lat = LAT_B, \qquad lon = LON_B$$

Note that the two pairs cannot be equal, i.e.

$$LAT_A \neq LAT_B$$

or

$$LON_A \neq LON_B$$

Otherwise, $A \equiv B$.

In the case in which $A \neq B$, i.e. at least one of the coordinates differs between nodes A and B, then their distance is greater than zero. Trivially, by using Pythagorean theorem, we can compute the distance between the two considered points. So:

$$d = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2} = \sqrt{(LAT_A - LAT_B)^2 + (LON_A - LON_B)^2} \tag{5.1}$$

assuming to map latitude on x-coordinate and longitude on y-coordinate.

In order to compute the distance on Earth, it is needed to use the great circle distances. It considers the curvature of the sphere. This allows us to obtain a good approximation of the represented real world.

For this purpose, the $Haversine$ formula is used:

$$a = sin^2(\Delta\Psi/2) + cos(\Psi_A)cos(\Psi_B)sin^2(\Delta\lambda/2) \tag{5.2}$$

where $\Delta\Psi$ is the difference between the two latitudes, Ψ_A is the latitude of A and Ψ_B is the latitude of B, $\Delta\lambda$ is the difference of the two longitudes.

By using the 5.2

$$c = 2atan2(\sqrt{a}, \sqrt{1-a}) \tag{5.3}$$

So the resulting distance is:

$$d = Rc \tag{5.4}$$

where R is the radius of Earth.

Export from a bounding box on map

Once we have defined the latitude and longitude coordinates, we can present the concept of "bounding box". In particular, when the user taps the zoom labels, it selects the part of the map it is interested in. So it exports the submap in XML format, in order to build the corresponding graph. The selected area is a *bounding box* defined by four coordinates, two latitude coordinates and two longitude coordinates.

The export of the area bounded by the box operates according to command similar to the one shown below:

bounding box = min Lon, min Lat, max Lon, max Lat

or, in a more simple way, considering the edges of the "box":

bounding box = left, bottom, right, top

GraphViz Tool and DOT language

Before to explain how the processing module works, it is necessary to provide some information about a specific tool used by our web-app to display the graph. The used tool is $GraphViz$: it is a simple Graph Visualization Tool that reads attributed graph from a text file and returns images or SVG for web pages, PDF or Postscript for inclusion in other documents, etc. In our implementation, the tool receives as input a text file that describes the considered graph by using a specific language, called DOT. It can represent both directed and undirected graphs.

We can see two simple examples of DOT file (Fig. 5.9 and 5.11) converted into the corresponding graphs (Fig. 5.10 and 5.12) by using $GraphViz$ tool.

```
graph {
   a -- b;
   b -- c;
   c -- d;
   a -- f;
   a -- c;
   a -- d;
   b -- d;
   b -- f;
   c -- f;
   d -- f;
}
```

Figure 5.9. Example of simple undirected graph represented in DOT format

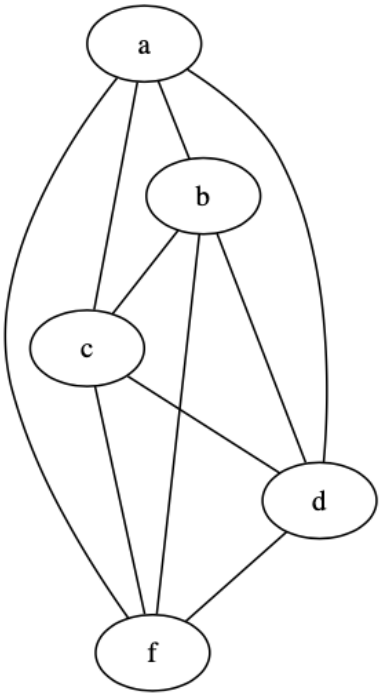


Figure 5.10. Example of undirect graph built by using GraphViz, starting from DOT format (See fig. 5.9)

It is often useful to adjust the graph representation or position of its nodes and edges. For this purpose, it is possible to use and set attributes of nodes and edges in the input DOT file.

In order to build a directed graph, we can use the *digraph* functionality of the tool. The example in figure 5.11 is a DOT representation of a directed graph, in which we also introduce *label* attributes. It is used in our implementation in order to show the weight of each edge.

```
digraph {
   a -> b[label="0.2"];
   a -> d[label="0.4"];
   a -> c[label="0.4"];
   c -> b[label="0.6"];
   c -> e[label="0.6"];
   d -> e[label="0.5"];
   e -> b[label="0.7"];
}
```

Figure 5.11. Example of directed weighted graph represented in DOT format

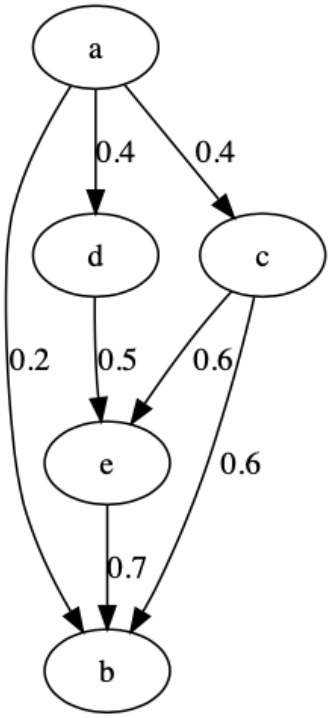


Figure 5.12. Example of direct graph built by using GraphViz, starting from DOT format (See fig. 5.11)

5.1.3 Processing

After receiving the entry directive from the user, the application, as previously written, starts its computation by exporting the XML file. It is adequately analysed through specific functions, that export the data and store it in a different type of map. In particular, we use the specif *Map* class of Java programming language, so:

$$Map < K, V >$$

where

- K is the type of keys maintained by this map
- V is the type of mapped values

So, by using this class, we create a Map object in which the keys are the ID of the elements, while the mapped values are the OSM elements objects. The OSM objects are, trivially, *nodes*, *ways* and *relations*, whose description has been provided in the previous section. Each of them is described by a dedicated class and therefore stored among the elements of the Map. Note also that we provide a *tag* storage system for each of the OSM objects.

Once the map is obtained, it is possible to build a graph whose nodes and edges are the elements of the Map class. The algorithm searches, among the nodes in Map, for those that could constitute nodes of our graph. Recalling what previously written about the choice of the graph nodes, we look for those that identify a crossroad. From a practical point of view, if a node on a specific *way* is also present in another *way*, then it represents a crossroad, therefore it will be added to the graph.

In order to identify the edges between adjacent nodes, it is necessary to use the set of nodes of the newly graph. So the algorithm combines the set of nodes just mentioned in groups of two: for each combination, it checks that the elements of the pair are both in one of the *ways* contained in Map: if so, the pair is made up of adjacent nodes, so we can add the corresponding edge to the graph. Note that, if no information are specified about the *oneway* tag of the *way* in question, a bidirectional edge will be added to the graph. Moreover, in this phase, it is necessary to use a function useful for calculating the distance between the two considered nodes, thus assigning a weight to the considered edge. In particular, using the coordinates of the two adjacent nodes, the distance is calculated, using the formulas written in the previous section (See eq. 5.4).

At this point, having the graph, it is necessary to display on it some planning algorithms. Remember, in fact, that the user (the observer), after having selected a region on the map, runs Goal Recognition algorithms that aim to discover where the target is going: in particular, it needs to reason about the target's strategy to obscure its goals. So, by using the planning algorithm provided in chapter 4, in addition to the well-known Dijkstra and Yen's algorithms, the web-app displays the possible paths the target can select, based on its budget. To do this, the web-app uses the $GraphViz$ tool.

Before to show how the application displays the algorithms on the graph, we test that it returns the correct graphical representation of the graph described by the OSM XML file. To do this, we compare the map received from the OSM web page with the graph created by GraphViz tool.

Given the simple map in figure 5.13, the corresponding graph in figure 5.14 shows the corresponding graphical representation: there is, as expected, a node for each crossroad.

Obviously, the *ways* that are outside the box the user create are not tacked into account during the graph building.

As previously explained, it was computed the distance between nodes: it is represented on the edges of the graph in figure 5.14.

The map in figure 5.13 presents a main *way* in yellow with different nodes $N0, N1, N2, N3$ on it. The other *ways* are not represented on the corresponding graph (Fig. 5.14) due to the fact that we have no information about their end. It could be, for example, a *dead end*, so it would not lead to any node or target goal. Note also that, if we have more than one *node* on the same *way*, then the algorithm creates an edge for every possible combination of two nodes.

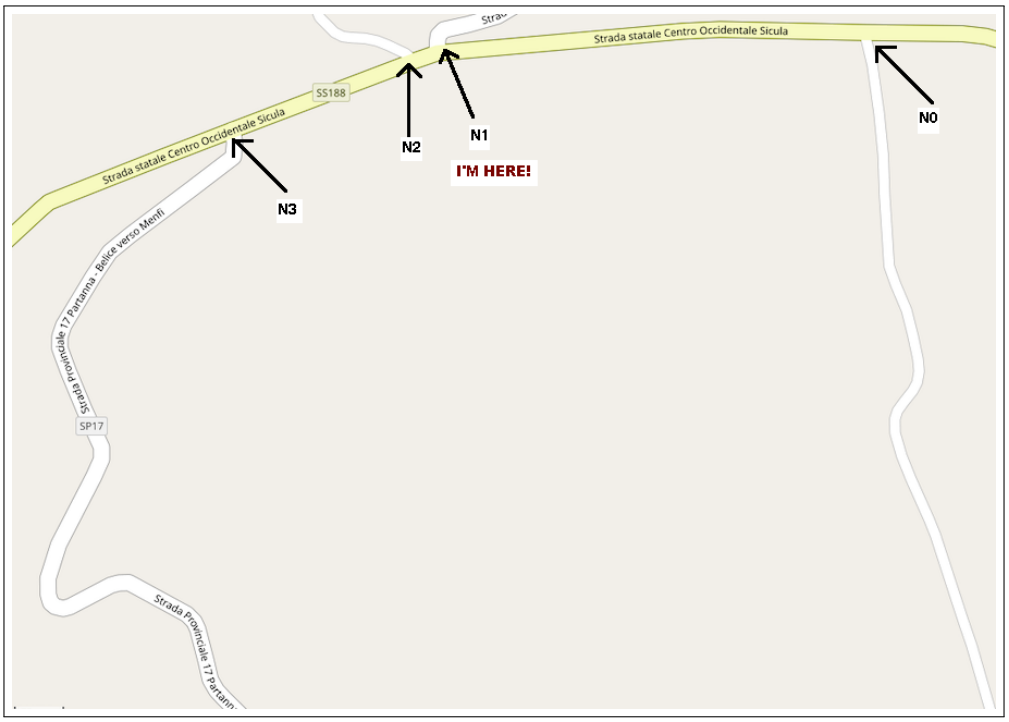


Figure 5.13. Portion of map that the user selects on OSM

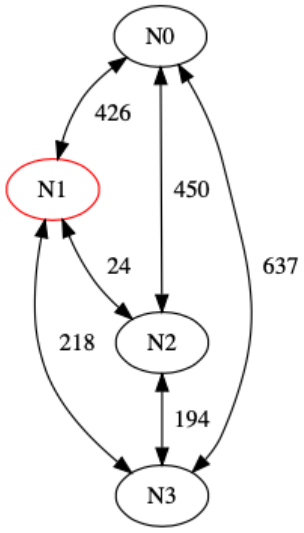


Figure 5.14. Graph corresponding to the Map in figure 5.15

Before presenting the tests performed, some details are provided regarding the choice of the algorithms implemented on the graph.

We have chosen to implement on the obtained graph three planning algorithm, the Dijkstra algorithm, the Yen algorithm and the main algorithm (presented and explained in chapter 4). Remember that the first one returns the optimal path from source to destination, while the second one has been implemented such that it returns the set of suboptimal paths whose cost is less than or equal to a specified travel budget. Finally, the main algorithm returns the largest subset of suboptimal paths s.t. its undisclosing index is maximum. So the web-app shows on the display different versions, i.e. different graphical windows, of the same graph.

The first set of windows shows the optimal paths obtained by the implementation of Dijkstra's algorithm on the given graph: we have a window for each destination.

The implementation of Yen's algorithm on the graph produces a new set of windows: it uses the budget index r, mentioned in chapter 4, and provided by the user, to compute a set of suboptimal paths γ s.t.

$$|\gamma| \leq r\lambda(o, d)$$

where $d \in \mathcal{D}$ is a generic destination in the given graph, $|\gamma|$ is the cost of path γ and λ is the weight of the shortest path connecting the source o to the destination d (See 4).

The last set of windows shows the set of remaining paths after the algorithm presented in chapter 4 has been implemented on the given graph. The algorithm also provides the value of maximum undisclosing index achieved within a certain budget value.

5.1.4 First Test

Now we can present a simple test: the goal is to show that the application actually converts a map from XML format to graph representation, and correctly applies on it the planning algorithm just discussed. To do this, consider the map in figure 5.15.

The bounding box returns the following four coordinates:

$$\begin{array}{rcc} \text{min Lon} & = & 7.67635 \\ \text{min Lat} & = & 45.08373 \\ \text{max Lon} & = & 7.67927 \\ \text{max Lat} & = & 45.08521 \end{array}$$

Then the user selects the source node and the set of possible destinations by using the command line. In this case

$$\begin{array}{c} \textit{source node} : N0 \\ \textit{destinations node}: N3,\ N5 \end{array}$$

Given the starting data, the application shows the resulting graph (See Fig. 5.16) and implements on it the algorithm previously mentioned.

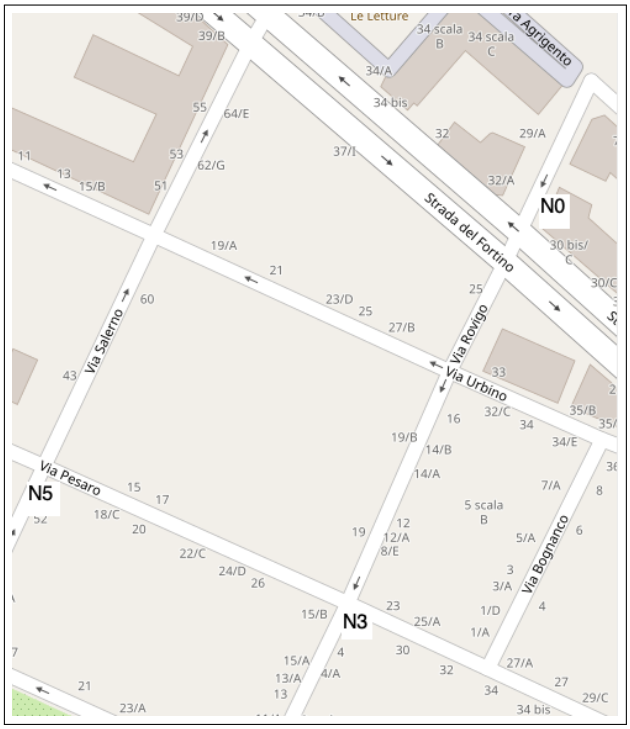


Figure 5.15. Map that the user selects on OSM

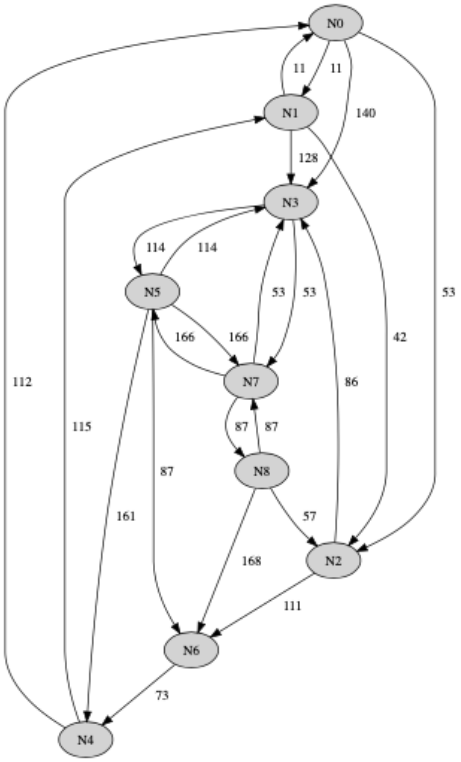


Figure 5.16. Graph corresponding to the Map 5.15

The first implemented algorithm is the Dijkstra algorithm. The system returns the optimal paths (See figure 5.17 and 5.18) with the corresponding optimal costs:

$$\text{Optimal path wrt destination } N3 : N0 - N1 - N3$$
$$\text{Optimal path wrt destination } N5 : N0 - N1 - N3 - N5$$

The corresponding optimal costs are:

$$cost_{N3}* = 139$$
$$cost_{N5}* = 253$$

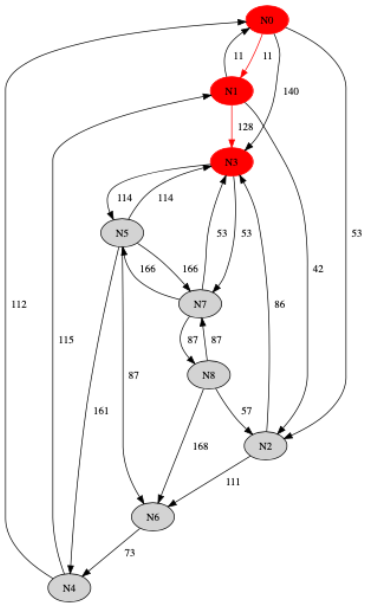


Figure 5.17. Implementation of Dijkstra Algorithm wrt destination $N3$ on the graph corresponding to the Map in figure 5.15

source node: N0
destination node: N3
shortest path: N0 - N1 - N3
cost: 139

source node: N0
destination node: N5
shortest path: N0 - N1 - N3- N5
cost: 253

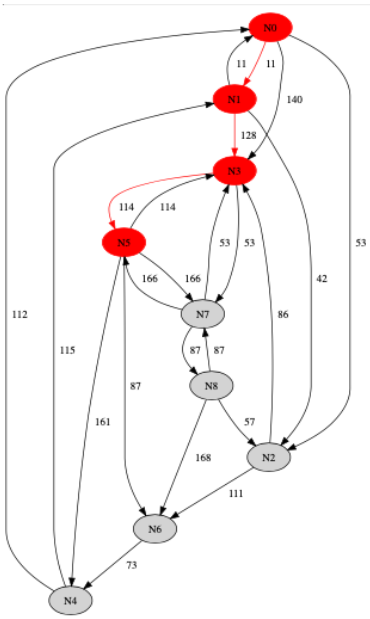


Figure 5.18. Implementation of Dijkstra Algorithm wrt destination $N5$ on the graph corresponding to the Map in figure 5.15

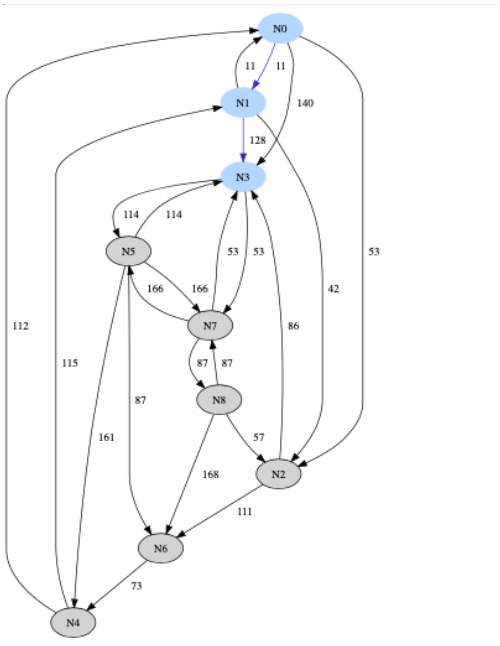
Starting from the optimal cost values, the Yen algorithm is implemented: it returns the set of suboptimal paths. Their cost has to be less than an upper limit depending on the optimal corresponding cost. In particular, by selecting, for example, a budget value $r = 1.2$, the Yen algorithm returns the paths in table 5.1.

Suboptimal paths wrt destination $N3$	$N0 - N1 - N3$
	$N0 - N2 - N3$
	$N0 - N3$
Suboptimal paths wrt destination $N5$	$N0 - N1 - N3 - N5$
	$N0 - N2 - N3 - N5$

Table 5.1. Suboptimal paths obtained by using Yen's algorithm, budget $r = 1.2$, source node $N0$, destination nodes $N3, N5$

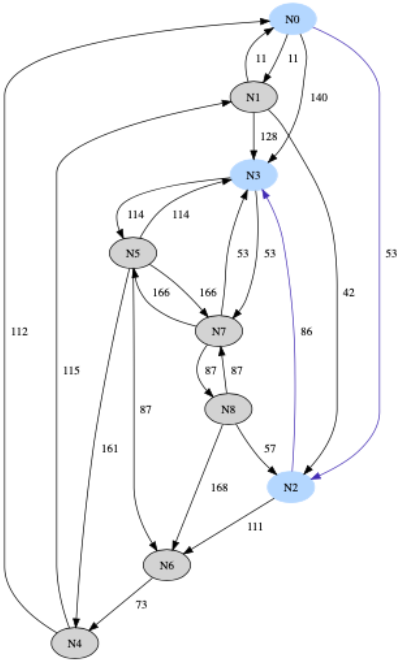
The web app return a graph for each of this suboptimal paths: they are shown in figures 5.19, 5.20, 5.21, 5.22 and 5.23.

For the destination $N3$ the suboptimal paths are shown in blue: the costs of that paths must be less than or equal to $r \cdot cost_{N3}*$ ($r = 1.2$, $cost_{N3}* = 139$).



source node: N0
destination node: N3
suboptimal path: N0 - N1 - N3
cost: 139

Figure 5.19. A suboptimal path from source N0 to destination N3



source node: N0
destination node: N3
suboptimal path: N0 - N2 - N3
cost: 139

Figure 5.20. A subopt. path from source N0 to dest. N3

source node: N0
destination node: N3
suboptimal path: N0 - N3
cost: 140

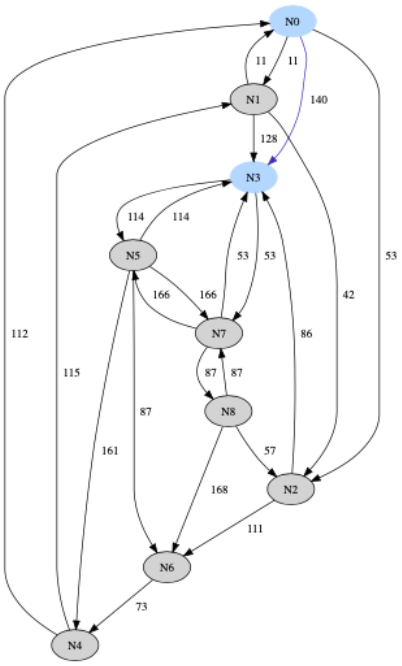
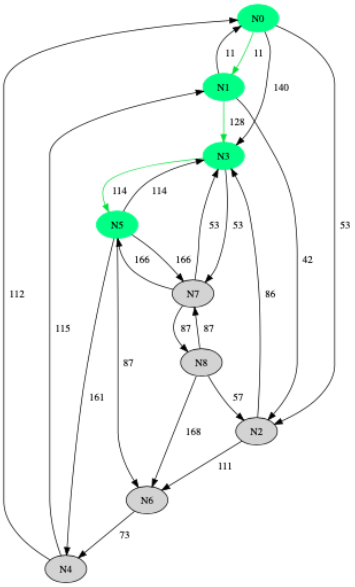


Figure 5.21. A subopt. path from source N0 to dest. N3

For the destination $N5$ the suboptimal paths are shown in green: the costs of that paths will be less than or equal to $r \cdot cost_{N5}*$ (r = 1.2, $cost_{N5}*$ = 253).



source node: N0
destination node: N5
suboptimal path: N0 - N1 - N3 - N5
cost: 253

Figure 5.22. A subopt. path from source N0 to dest. N5

source node: N0
destination node: N5
suboptimal path: N0 - N2 - N3 - N5
cost: 253

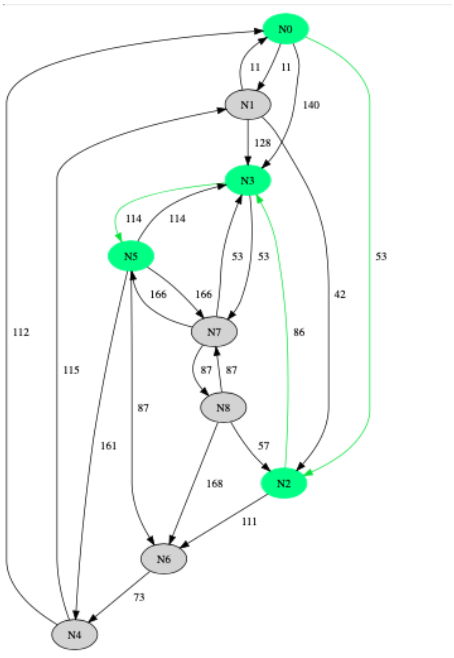


Figure 5.23. Subopt. path from source N0 to dest. N5

The last algorithm that we implement on the considered graph (See Fig. 5.16) is the main algorithm described in chapter 4: practically, it selects the largest subset of paths reported in table 5.1 s.t. its undisclosing index is the maximum achievable. The algorithm returns the following set of paths and its maximum undisclosing index:

Subset of subopt. paths	$N0 - N1 - N3$ $N0 - N2 - N3$ $N0 - N1 - N3 - N5$ $N0 - N2 - N3 - N5$
Max. undisclosing index achievable (r=1.2)	2

Table 5.2. Subset of paths $\bar{\mathcal{P}}^{\star}$ whose undisclosing index $T_{\bar{\mathcal{P}}}$

As expected, the graph is still (o, d) – *connected*, where $d \in \mathcal{D}$ and $\mathcal{D}$ is the set of destinations of the given graph. Moreover, since the graph considered is very small, it is possible to verify that the maximum undisclosing index achievable within the new subset of paths is actually equal to 2.

5.2 Results, precision and speed

The tests were performed on three data sets obtained from OSM web page. The data are derived from the export of concentric bounded boxes with different sizes. In our case, the central point could be, for example, the last known position of the fleeing agent: the user, in this case the observer, must plan possible trajectories from that point (it is considered, therefore, the source of the graph) to the various destinations available in the map, in order to find the target.

The first set is the one used in the previous section (sec. 5.1.4) in order to show how the app graphically displays the results. The second set is larger than the first one, as already written: its coordinates and graphic representation are in the appendix (sec. .1.1). The same applies to the third set (sec. .1.2).

Then, for each set, different tests were performed, one for each value of r chosen. The appendix shows the algorithm results for the different data sets and budget values chosen (see appendix, Sec. .2.1, .2.2 and .2.3).

Tests related to the speed and accuracy of the implemented web application have also been performed. Note that only the execution time of the main computation part was measured, that is the part of code related to the acquisition of the graph described by adjacency lists and the implementation of the planning algorithms on it.

First of all, we present and highlight how the computational time strongly increases, as the chosen area increases. For the observer, choosing a large area can be advantageous: it can consider a wider set of target's escape routes. At the same time, the greater the range of possible paths, the greater the possibility of the target to keep its final destination hidden. The considered tests were performed for different values of r (i.e. for different travel budgets). The results are shown in the table 5.24.

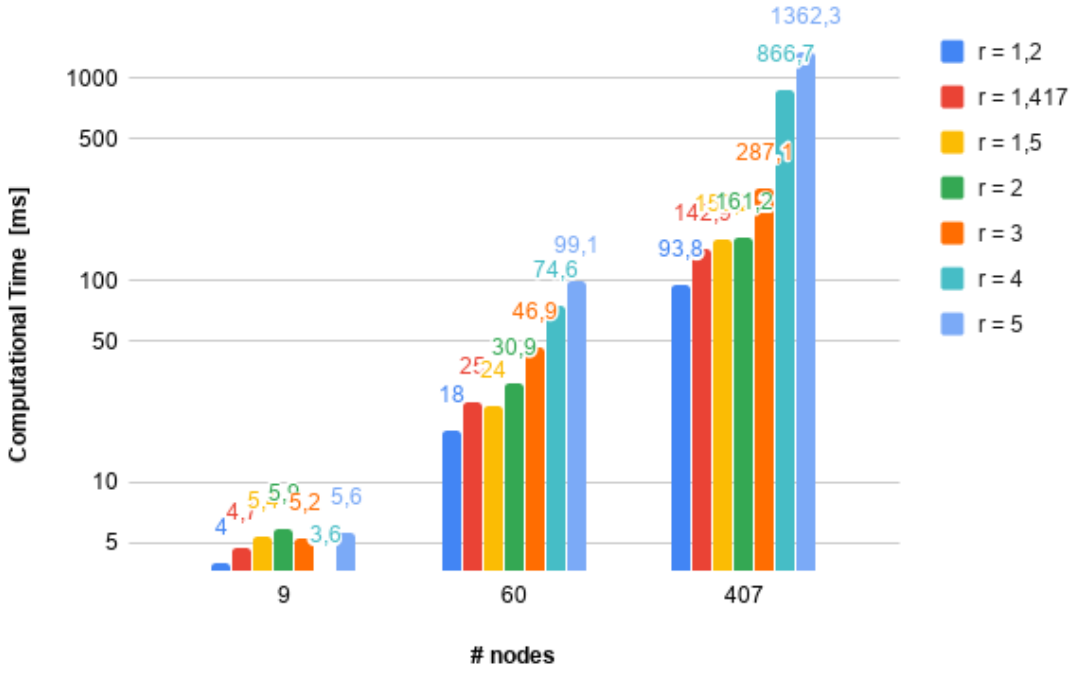


Figure 5.24. Computation time trend with respect to the number of nodes. Test performed for different values of r ($r = 1.2, 1.417, 2, 3, 4, 5$)

The values of r chosen derive from the particular topology of the environment. In particular, "critical" values were chosen. For each of these critical values, by applying the Yen algorithm, suboptimal sets of different sizes are obtained. For example,

- for $r = 1.200$ the set of paths connecting the source to the first destination $N3$ consists of 3 paths, while the set of paths connecting the source and the destination $N5$ consists of 2 paths,
- for $r = 1.417$ the set of paths connecting the source to the first destination $N3$ consists of 3 paths, as well as the set of paths connecting the source and the destination $N5$,
- for $r = 1.5$ the set of paths connecting the source to the first destination $N3$ consists of 3 paths, while the set of paths connecting the source and the destination $N5$ consists of 4 paths,

and so on.

Note that the computation time trend has been represented with respect to the number of nodes using a logarithmic scale, in order to better visualize the results. The computation time, in fact, grows exponentially, as the number of nodes in the graph increases, as expected (See Fig. 5.24). Then, the size of data set strongly influences the speed of the web-app execution. The trend depends on the topology of the map. In some particular case, we can have the same number of nodes in two or more concentric areas with different size: think, for example, of an area delimited by green areas, without viable roads.

It may be useful to isolate the results for each map, to analyze the evolution of the computation time with respect to the budget increase (See figures 5.25, 5.26 and 5.27).

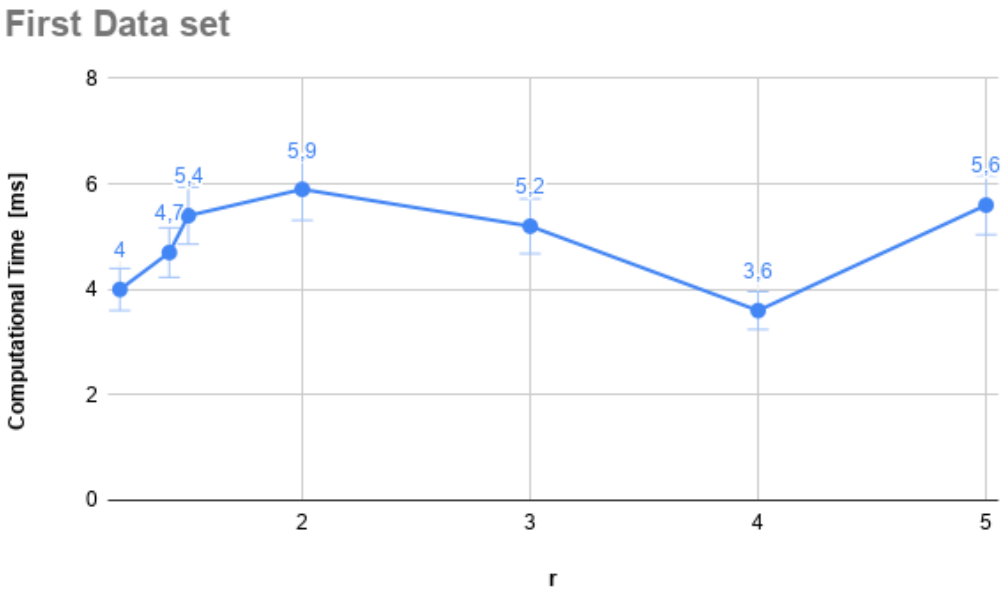


Figure 5.25. First Data Set: trend of computational time wrt r (index related to the target's budget)

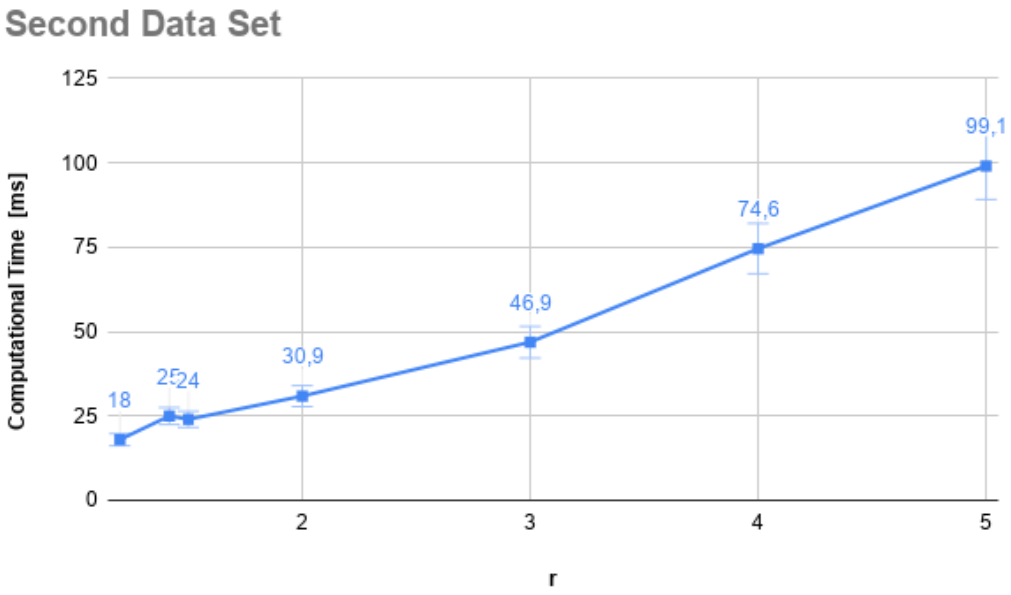


Figure 5.26. Second Data Set: trend of computational time wrt r (index related to the target's budget)

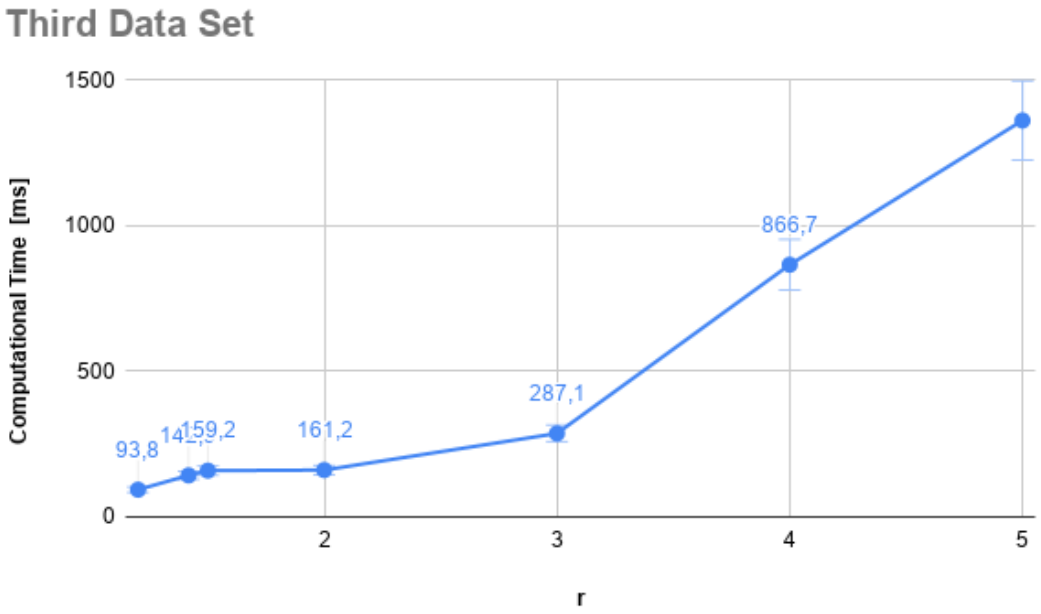


Figure 5.27. Third Data Set: trend of computational time wrt r (index related to the target's budget)

The first data set does not present evident variations regarding the running time: with an increase of r, the speed does not differ too much from the average value of 4.91 ms.

The second data set, on the other hand, presents a more evident growth of the computation time, compared to the increase in the budget (r). Quite simply, the greater the value of r, the greater the number of paths that can be traversed by the target within its budget, the greater the computational effort of the web application in calculating all the possible paths.

Finally, the running time with respect to the third set of data presents an even more accentuated growth than the second case. The growth of the computation time is evident for $r \geq 3$. Note, however, that tripling, quadrupling, etc. the value of r means greatly increasing the resources that the target has at its disposal. Such cases are, generally, not taken into consideration.

We want to show how the undisclosing index varies by executing algorithm (3) on the set of suboptimal paths. In particular, we aim to show that the implemented algorithm helps increases the undisclosing index value of a given set of paths, properly removing some of its elements. So, for each data set and for each corresponding suboptimal set obtained from the implementation of the Yen's algorithm, the value of the initial undisclosing index was calculated and compared with the final one, taken by executing the algorithm (3). Recalling the mathematical notation used, we denote by $\bar{\mathcal{P}}$, the suboptimal set of paths: the undisclosing index of the considered set is $t_{\bar{\mathcal{P}}}$. The maximum undisclosing index achievable within the subset of paths obtained by executing the algorithm (3) was indicated, instead, with $T_{\bar{\mathcal{P}}}$. The tables 5.3, 5.4 and 5.5 show a comparison between $t_{\bar{\mathcal{P}}}$ and $T_{\bar{\mathcal{P}}}$ for each data sets and travel budgets.

	$t_{\bar{\mathcal{P}}}$	$T_{\bar{\mathcal{P}}}$
r = 1.2, 1.417, 1.5, 2, 3, 4, 5	0	2

Table 5.3. The undisclosing index of the set $\bar{\mathcal{P}}$ obtained starting from the implementation of the Yen's algorithm and the undisclosing index after the execution of the algorithm (3)

	$t_{\bar{\mathcal{P}}}$	$T_{\bar{\mathcal{P}}}$
r = 1.200, 1,417, 1.5, 2, 3,	0	2
r = 4, 5	0	4

Table 5.4. The undisclosing index of the set $\bar{\mathcal{P}}$ obtained starting from the implementation of the Yen's algorithm and the undisclosing index after the execution of the algorithm (3)

	$t_{\bar{\mathcal{P}}}$	$T_{\bar{\mathcal{P}}}$
r = 1.200, 1,417, 1.5, 2, 3,	0	2
r = 4, 5	0	4

Table 5.5. The undisclosing index of the set $\bar{\mathcal{P}}$ obtained starting from the implementation of the Yen's algorithm and the undisclosing index after the execution of the algorithm (3)

The tables show how the execution of the algorithm (3), given the suboptimal set of paths $\bar{\mathcal{P}}$, leads to an increase in its undisclosing index. Remember that the strategy used, involves removing from the starting set the paths whose undisclosing index is less than the maximum one achievable. So, through this approach, as is evident in the tables 5.3, 5.4 and 5.5, we were able to maximize the undisclosing index of a set of paths.

We now want to show how the trend of the undisclosing index varies with changes in the travel budget of the target. By increasing the budget, the number of paths that make up the suboptimal set increases: the larger the set of suboptimal path, the greater can be the maximum undisclosing index achievable (see proposition 6). Remember that, given a source o and a destination d, the suboptimal set consists of all the paths that connect o and d and whose cost is less than or equal to $r \cdot cost*$, where $cost*$ is the optimal cost (the shortest path length connecting o and d). The reasoning in our case is extended to a number of destinations greater than one. The trend of the maximum undisclosing index achievable is shown in figure 5.28.

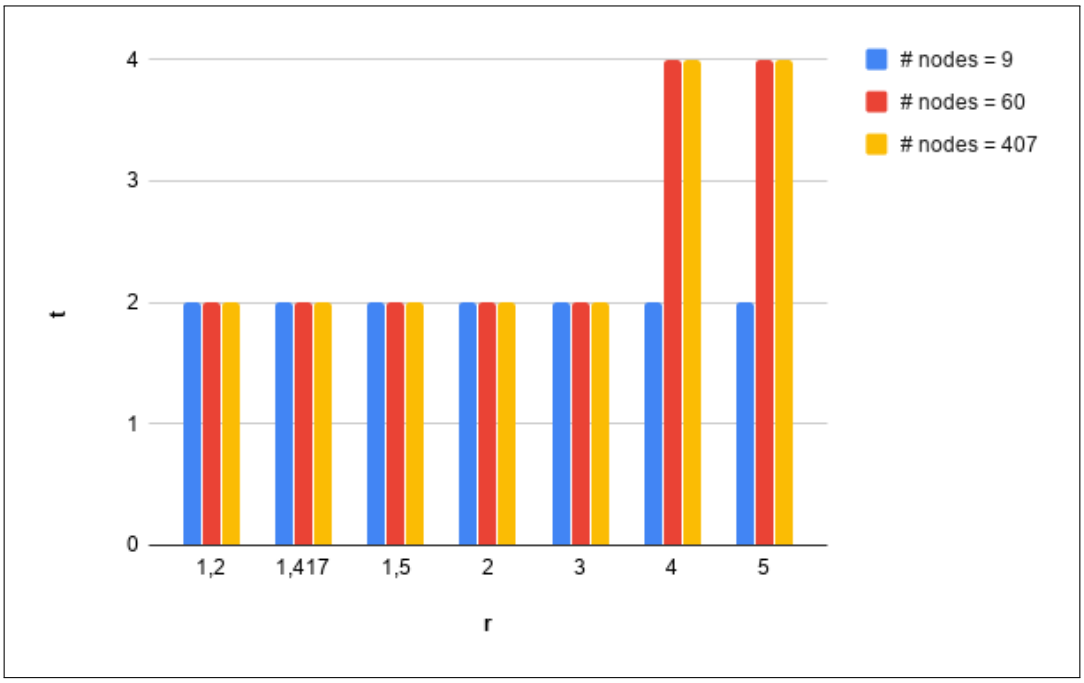


Figure 5.28. Trend of the Maximum Undisclosing Index achievable t wrt budget value r

In general, the undisclosing index value increases as the available budget increases. In reality, the relationship between the undisclosing index and the budget is strongly dependent on the size of the area considered: for small areas where the number of nodes is not too large, the number of possible paths is very limited. Therefore, even if the budget increases, the maximum undisclosing index remains unchanged, since the number of paths that constitutes the set of suboptimal paths remains unchanged (See Max.t achievable for a number of nodes equal to 9, blue set, Fig. 5.28).

Differently, considering areas with a higher number of nodes, then the maximum undisclosing index varies more consistently, depending on the travel budget available (See Fig. 5.28, orange and yellow sets).

Practically, with the same travel budget, as the area considered increases, the number of nodes increases, so the number of paths belonging to the suboptimal set could increase. Therefore, for the first set of data, whose number of suboptimal paths is strongly less than the ones in sets 2 and 3, we will have a maximum undisclosing index achievable lower than the maximum undisclosing indexes achievable in map 2 and 3, considering the same budget conditions (See figure 5.28, values of t in Map 1, 2, 3 for $r = 4$ and for $r = 5$).

Chapter 6

Conclusion

6.1 Conclusion and Future Work

Through this thesis we have provided an instance of Goal Recognition: the considered scenario was a typical surveillance problem in which an evasive target, for example, a criminal, moving in the environment, aims to reach its destination (a hideout) by car as soon as possible, while an observer, for example, a drone, flies over the geographical area to discover the fleeing target.

The problem was analyzed using planning techniques and graph algorithms, such as Dijkstra and Yen's algorithms. Furthermore, the introduction of a new measure, the the *undisclosing index*, contributing to the modification of the environment model, was useful for the observer to reason about the possible target's strategies to obscure its goals. The undisclosing index represents the maximal length of the prefix of a path that a fleeing agent may take before its goal becomes apparent to the observer: therefore, maximizing the value of the undisclosing index means modifying the model of the environment in order to increase the probability that the target has to escape. In fact, the greater the maximum value of the undisclosing index within a given travel budget, the greater the number of moves the target can perform before its final goal is evident to the observer.

Experimental evaluations have confirmed that the undisclosing index strongly depends on the imposed travel budget: in particular, as the available travel budget grows, the maximum achievable undisclosing index increases. In particular, the number of paths whose cost is within the imposed travel budget increases as the latter increases. In a larger set, the probability of finding paths whose prefixes coincide also increases, therefore, according to the theoretical concepts already expressed in the previous chapters, the maximum undisclosing index achievable within the given set increases. The results also showed that the maximum undisclosing index that can be reached depends on the area chosen by the user: the greater the extent of the selected region, the greater the number of suboptimal paths found on it. A larger set of suboptimal paths implies a greater probability of obtaining a higher maximum undisclosing index value. Trivially, the higher the achievable undisclosing index, the greater the probability of the target reaching its final goal, before the observer finds it.

The creation of a graphic interface has simplified the use of the implemented algorithms and the visualization of these on graphs, built starting from real data and exported from OpenStreetMap web page.

A natural extension of the problem in question involves considering real changes and modification in the environment, therefore developing strategies aimed at favouring or hindering Goal Recognition, depending on the application of interest. The implemented algorithms and the experimental evaluations presented in this thesis could be useful tools for solving these more complex problems that fall under the category of Goal Recognition Design problems.

Appendices

.1 Tested Areas

.1.1 Map and data: second tested area

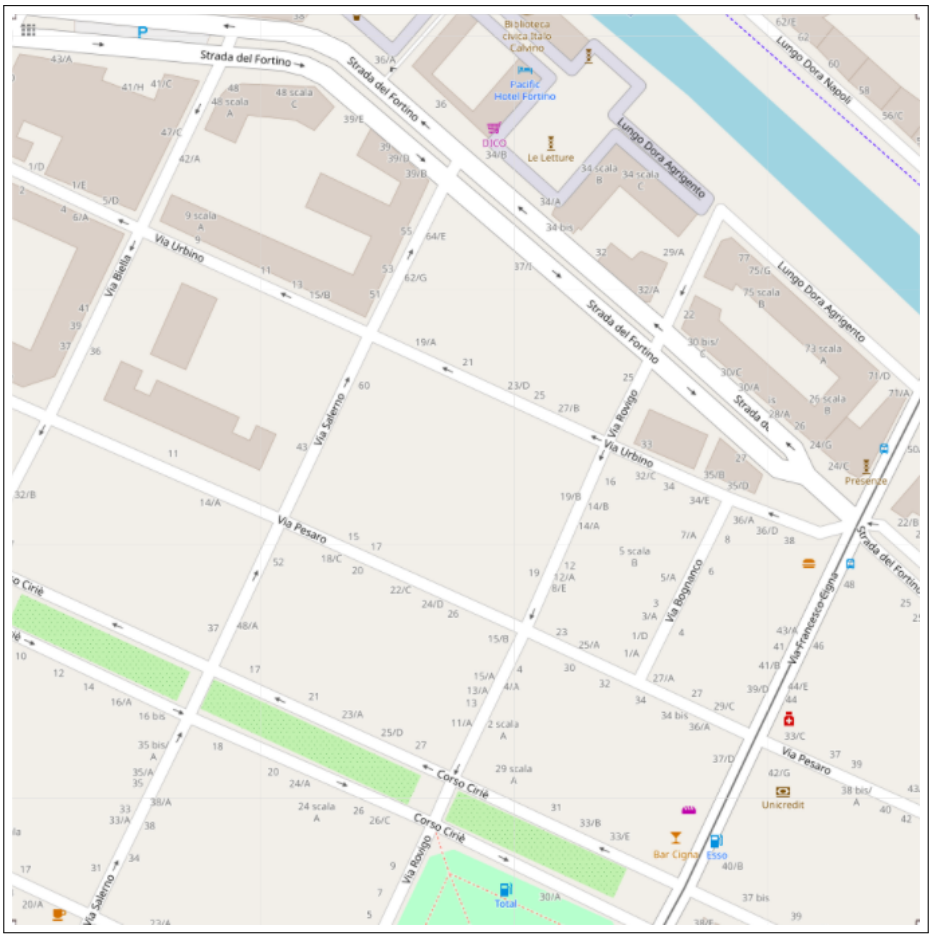


Figure 1. Map of the second selected area

minLat	45.08273
minLon	7.67535
maxLat	45.08621
maxLon	7.68027
nodes found	2330
nodes added to graph	60
ways	55
relations	81

Table 1. Table representing the area selected and the corresponding graph's elements statistic

.1.2 Map and data: third tested area

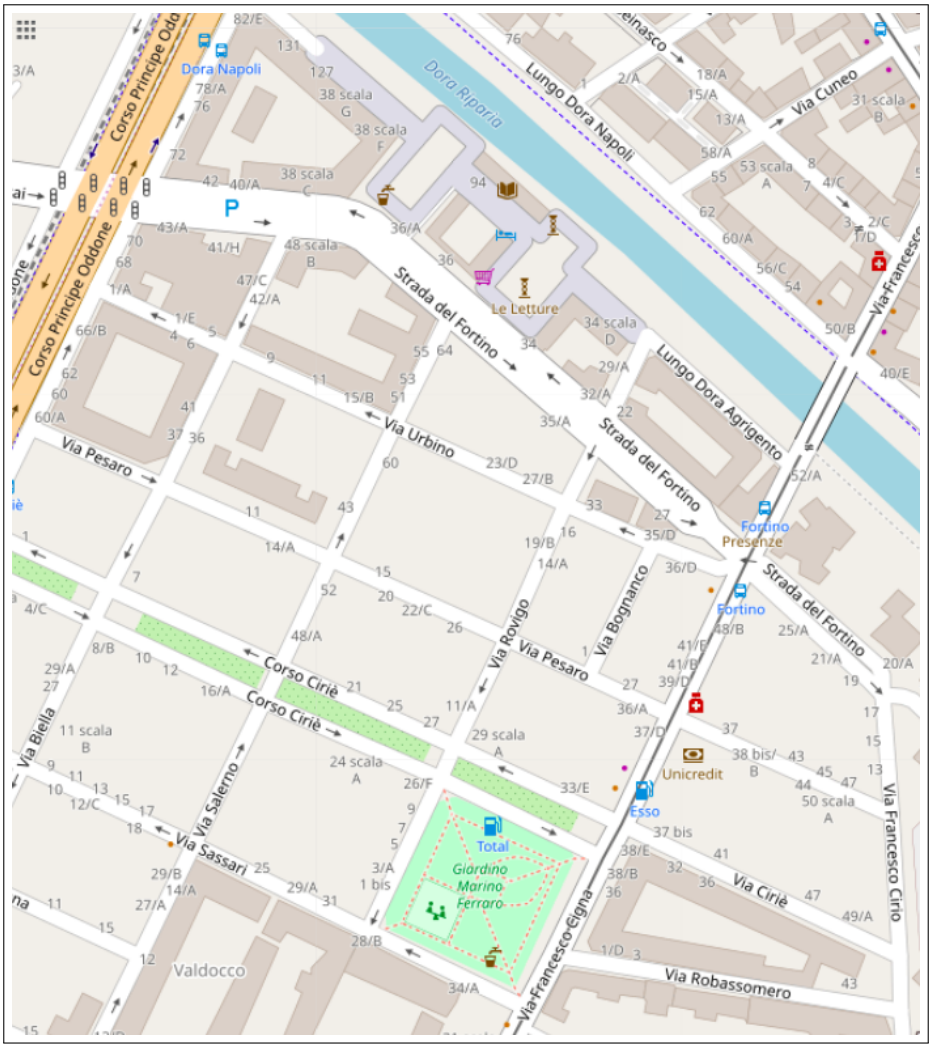


Figure 2. Map of the third selected area

minLat	45.08173
minLon	7.67435
maxLat	45.08721
maxLon	7.68127
nodes found	4479
nodes added to graph	407
ways	328
relations	110

Table 2. Table representing the area selected and the corresponding graph's elements statistic

.2 Tables of Tests

.2.1 Results: first data set

Source	**Dest**	**r=1.2**	**r=1.417**	**r=1.5**	**r=2**	**r=3**	**r=4**	**r=5**	**cost*[m]**
$N0$	$N3$	$(N0, N1, N3)$	/	/	/	/	/	/	139
$N0$	$N5$	$(N0, N1, N3)$	/	/	/	/	/	/	253

Table 3. Optimal paths from source $N0$ to destinations $N3$ and $N4$

r = 1.2

Source	**Dest**	**Suboptimal paths**
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$

Table 4. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 1.2$

Source	**Dest**	**Suboptimal paths**
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$

Table 5. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 1.2$)

r = 1.417

Source	Dest	Suboptimal paths
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$
$N0$	$N3$	$(N0, N3, N5)$

Table 6. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where r = 1.417

Source	Dest	Suboptimal paths
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$

Table 7. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is t = 2 (r = 1.417)

r = 1.5

Source	Dest	Suboptimal paths
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$
$N0$	$N3$	$(N0, N3, N5)$
$N0$	$N3$	$(N0, N3, N7, N5)$

Table 8. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where r = 1.5

Source	Dest	Suboptimal paths
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$

Table 9. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is t = 2 (r = 1.5)

r = 2,3,4,5

Source	Dest	Suboptimal paths
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$
$N0$	$N3$	$(N0, N3, N5)$
$N0$	$N3$	$(N0, N3, N7, N5)$

Table 10. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where r = 2,3,4,5

Source	Dest	Suboptimal paths
$N0$	$N3$	$(N0, N1, N3)$
$N0$	$N3$	$(N0, N2, N3)$
$N0$	$N3$	$(N0, N1, N3, N5)$
$N0$	$N3$	$(N0, N2, N3, N5)$

Table 11. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is t = 2 (r = 2,3,4,5)

.2.2 Results: second data set

Note that, in the following tables, the source node is called $N35$ (it is equivalent to the old $N0$), while the destinations are $N37$ and $N53$ (they are equivalent to the old $N3$ and $N5$ respectively).

Source	**Dest**	**r=1.2**	**r=1.417**	**r=1.5**	**r=2**	**r=3**	**r=4**	**r=5**	**cost*[m]**
$N35$	$N37$	$(N35, N36, N37)$	/	/	/	/	/	/	139
$N35$	$N53$	$(N35, N36, N37, N53)$	/	/	/	/	/	/	253

Table 12. Optimal paths from source $N35$ to destinations $N37$ and $N53$

r = 1.2

Source	**Dest**	**Suboptimal paths**
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$

Table 13. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 1.2$

Source	**Dest**	**Suboptimal paths**
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$

Table 14. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 1.2$)

r = 1.417

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$
$N35$	$N53$	$(N35, N37, N53)$

Table 15. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where r = 1.417

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$

Table 16. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is t = 2 (r = 1.417)

r = 1.5

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$
$N35$	$N53$	$(N35, N37, N53)$
$N35$	$N53$	$(N35, N37, N54, N53)$

Table 17. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where r = 1.5

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$

Table 18. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 1.5$)

r = 2

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$
$N35$	$N53$	$(N35, N37, N53)$
$N35$	$N53$	$(N35, N37, N54, N53)$
$N35$	$N53$	$(N35, N38, N52, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N37, N53)$

Table 19. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 2$

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$

Table 20. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 2$)

r = 3

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$
$N35$	$N53$	$(N35, N37, N53)$
$N35$	$N53$	$(N35, N37, N54, N53)$
$N35$	$N53$	$(N35, N38, N52, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N37, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N23, N37, N53)$
$N35$	$N53$	$(N35, N30, N20, N53)$
$N35$	$N53$	$(N35, N30, N31, N20, N53)$
$N35$	$N53$	$(N35, N30, N25, N20, N53)$
$N35$	$N53$	$(N35, N27, N28, N51, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N51, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N55, N54, N53)$

Table 21. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 3$

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$

Table 22. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 3$)

r = 4

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N37$	$(N35, N4, N5, N36, N37)$
$N35$	$N37$	$(N35, N4, N5{,}36, N23, N37)$
$N35$	$N37$	$(N35, N38, N52, N53, N53, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$
$N35$	$N53$	$(N35, N37, N53)$
$N35$	$N53$	$(N35, N37, N54, N53)$
$N35$	$N53$	$(N35, N38, N52, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N37, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N23, N37, N53)$
$N35$	$N53$	$(N35, N30, N20, N53)$
$N35$	$N53$	$(N35, N30, N31, N20, N53)$
$N35$	$N53$	$(N35, N30, N25, N20, N53)$
$N35$	$N53$	$(N35, N27, N28, N51, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N51, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N55, N54, N53)$
$N35$	$N53$	$(N35, N39, N56, N57, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N57, N38, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N22, N54, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N22, N54, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N23, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N23, N37, N38, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N23, N38, N52, N53)$

Table 23. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 4$

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N38, N52, N53, N37)$
$N35$	$N53$	$(N35, N38, N52, N53)$

Table 24. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 4$ ($r = 4$)

r = 5

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N36, N37)$
$N35$	$N37$	$(N35, N23, N37)$
$N35$	$N37$	$(N35, N37)$
$N35$	$N37$	$(N35, N4, N5, N36, N37)$
$N35$	$N37$	$(N35, N4, N5{,}36, N23, N37)$
$N35$	$N37$	$(N35, N38, N52, N53, N53, N37)$
$N35$	$N53$	$(N35, N36, N37, N53)$
$N35$	$N53$	$(N35, N23, N37, N53)$
$N35$	$N53$	$(N35, N37, N53)$
$N35$	$N53$	$(N35, N37, N54, N53)$
$N35$	$N53$	$(N35, N38, N52, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N37, N53)$
$N35$	$N53$	$(N35, N4, N5, N36, N23, N37, N53)$
$N35$	$N53$	$(N35, N30, N20, N53)$
$N35$	$N53$	$(N35, N30, N31, N20, N53)$
$N35$	$N53$	$(N35, N30, N25, N20, N53)$
$N35$	$N53$	$(N35, N27, N28, N51, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N51, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N53)$
$N35$	$N53$	$(N35, N39, N27, N28, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N55, N54, N53)$
$N35$	$N53$	$(N35, N39, N56, N57, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N57, N38, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N22, N54, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N22, N54, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N23, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N23, N37, N38, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N21, N23, N38, N52, N53)$
$N35$	$N53$	$(N35, N39, N56, N50, N34, N36, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N50, N34, N23, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N50, N34, N37, N53)$
$N35$	$N53$	$(N35, N39, N56, N50, N34, N38, N52, N53)$

Table 25. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 5$

Source	Dest	Suboptimal paths
$N35$	$N37$	$(N35, N38, N52, N53, N37)$
$N35$	$N53$	$(N35, N38, N52, N53)$

Table 26. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 4$ ($r = 5$)

.2.3 Results: third data set

Note that, in the following tables the source node is called $N180$ (it is equivalent to the old $N0$), while the destinations are $N182$ and $N297$ (they are equivalent to the old $N3$ and $N5$ respectively).

Source	Dest	r=1.2	r=1.417	r=1.5	r=2	r=3	r=4	r=5	cost*[m]
$N180$	$N182$	$(N180, N181, N182)$	/	/	/	/	/	/	139
$N180$	$N297$	$(N180, N181, N182, N297)$	/	/	/	/	/	/	253

Table 27. Optimal paths from source $N180$ to destinations $N182$ and $N297$

r = 1.2

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$

Table 28. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 1.2$

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N297$	$(N180, N181, N182, N297)$
$N180$	$N297$	$(N180, N166, N182, N297)$

Table 29. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 1.2$)

r = 1.417

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$
$N180$	$N297$	$(N180, N182, N297)$

Table 30. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 1.417$

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N297$	$(N180, N181, N182, N297)$
$N180$	$N297$	$(N180, N166, N182, N297)$

Table 31. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 1.417$)

r = 1.5

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$
$N180$	$N297$	$(N180, N182, N297)$
$N180$	$N297$	$(N180, N182, N298, N297)$

Table 32. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 1.5$

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N297$	$(N180, N181, N182, N297)$
$N180$	$N297$	$(N180, N166, N182, N297)$

Table 33. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 1.5$)

r = 2

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$
$N180$	$N297$	$(N180, N182, N297)$
$N180$	$N297$	$(N180, N182, N298, N297)$
$N180$	$N297$	$(N180, N116{,}117, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N182, N297)$

Table 34. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 2$

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N297$	$(N180, N181, N182, N297)$
$N180$	$N297$	$(N180, N166, N182, N297)$

Table 35. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 2$)

r = 3

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$
$N180$	$N297$	$(N180, N182, N297)$
$N180$	$N297$	$(N180, N182, N298, N297)$
$N180$	$N297$	$(N180, N116{,}117, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N182, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N166, N182, N297)$
$N180$	$N297$	$(N180, N177, N164, N297)$
$N180$	$N297$	$(N180, N177, N178, N164, N297)$
$N180$	$N297$	$(N180, N177, N168, N164, N297)$
$N180$	$N297$	$(N180, N124, N177, N164, N297)$
$N180$	$N297$	$(N180, N170, N171, N296, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N296, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N117, N297)$
$N180$	$N297$	$(N180, N183, N303, N299, N298, N297)$

Table 36. Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 3$

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N297$	$(N180, N181, N182, N297)$
$N180$	$N297$	$(N180, N166, N182, N297)$

Table 37. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 2$ ($r = 3$)

r = 4

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N182$	$(N180, N159, N160, N181, N182)$
$N180$	$N182$	$(N180, N159, N160, N181, N166, N182)$
$N180$	$N182$	$(N180, N116, N117, N297, N182)$

$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$
$N180$	$N297$	$(N180, N182, N297)$
$N180$	$N297$	$(N180, N182, N298, N297)$
$N180$	$N297$	$(N180, N116{,}117, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N182, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N166, N182, N297)$
$N180$	$N297$	$(N180, N177, N164, N297)$
$N180$	$N297$	$(N180, N177, N178, N164, N297)$
$N180$	$N297$	$(N180, N177, N168, N164, N297)$
$N180$	$N297$	$(N180, N124, N177, N164, N297)$
$N180$	$N297$	$(N180, N170, N171, N296, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N296, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N117, N297)$
$N180$	$N297$	$(N180, N183, N303, N299, N298, N297)$
$N180$	$N297$	$(N180, N183, N303, N115, N299, N298, N297)$
$N180$	$N297$	$(N180, N183, N303, N115, N117, N297)$
$N180$	$N297$	$(N180, N183, N303, N115, N116, N117, N297)$
$N180$	$N297$	$(N180, N52, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N290, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N289, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N75, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N75, N51, N178, N168, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N289, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N75, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N68, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301{,}289{,}51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N51, N177, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N76, N290, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N76, N290, N67, N68, N51, N178, N164, N297)$

Table 38: Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost*$, where $r = 4$

Source	**Dest**	**Suboptimal paths**
$N180$	$N182$	$(N180, N116, N117, N297, N182)$
$N180$	$N297$	$(N180, N116, N117, N297)$

Table 39. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 4$ ($r = 4$)

r = 5

Source	Dest	Suboptimal paths
$N180$	$N182$	$(N180, N181, N182)$
$N180$	$N182$	$(N180, N166, N182)$
$N180$	$N182$	$(N180, N182)$
$N180$	$N182$	$(N180, N159, N160, N181, N182)$
$N180$	$N182$	$(N180, N159, N160, N181, N166, N182)$
$N180$	$N182$	$(N180, N116, N117, N297, N182)$
$N180$	$N297$	$(N180, N181, N182)$
$N180$	$N297$	$(N180, N166, N182, N297)$
$N180$	$N297$	$(N180, N182, N297)$
$N180$	$N297$	$(N180, N182, N298, N297)$
$N180$	$N297$	$(N180, N116,117, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N182, N297)$
$N180$	$N297$	$(N180, N159, N160, N181, N166, N182, N297)$
$N180$	$N297$	$(N180, N177, N164, N297)$
$N180$	$N297$	$(N180, N177, N178, N164, N297)$
$N180$	$N297$	$(N180, N177, N168, N164, N297)$
$N180$	$N297$	$(N180, N124, N177, N164, N297)$
$N180$	$N297$	$(N180, N170, N171, N296, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N296, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N297)$
$N180$	$N297$	$(N180, N183, N170, N171, N117, N297)$
$N180$	$N297$	$(N180, N183, N303, N299, N298, N297)$
$N180$	$N297$	$(N180, N183, N303, N115, N299, N298, N297)$
$N180$	$N297$	$(N180, N183, N303, N115, N117, N297)$
$N180$	$N297$	$(N180, N183, N303, N115, N116, N117, N297)$
$N180$	$N297$	$(N180, N52, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N290, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N289, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N75, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N290, N309, N310, N75, N51, N178, N168, N164, N297)$
$N180$	$N297$	$(N180, N52, N76, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N289, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N67, N68, N75, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N68, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301,289,51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N51, N177, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N76, N290, N309, N310, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N76, N290, N67, N68, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N76, N67, N68, N51, N178, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N301, N75, N76, N67, N68, N51, N178, N177, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N47, N49, N164, N297)$

$N180$	$N297$	$(N180, N52, N300, N137, N42, N70, N47, N49, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N56, N47, N49, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N210, N56, N47, N49, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N73, N47, N49, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N73, N47, N48, N49, N164, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N73, N47, N296, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N73, N47, N179, N296, N297)$
$N180$	$N297$	$(N180, N52, N300, N137, N42, N73, N47, N179, N296, N117, N297)$
$N180$	$N297$	$(N180, N52, N300, N139, N140, N51, N178, N164, N297)$

Table 40: Yen algorithm results: suboptimal paths whose cost is less than $r \cdot cost\star$, where $r = 5$

Source	**Dest**	**Suboptimal paths**
$N180$	$N182$	$(N180, N116, N117, N297, N182)$
$N180$	$N297$	$(N180, N116, N117, N297)$

Table 41. Main algorithm results: subset of suboptimal paths whose maximum undisclosing index is $t = 4$ ($r = 5$)

.3 Computational Time

	r = 1.2	**r = 1.417**	**r = 1.5**	**r = 2**	**r = 3**	**r = 4**	**r = 5**
Comp. Time of the 1^{st} Data Set [ms]	4.12	4.8417	5.55	6.1	5.3	4	5.6
Comp. Time of the 2^{nd} Data Set [ms]	18	25	24	30.9	46.9	74.6	99.1
Comp. Time of the 3^{rd} Data Set [ms]	93.8	142.9	159.2	161.2	287.1	866.7	1362.3

Table 42. Mean values of the computational times wrt to the value of r (budget)

Bibliography

[1] Balakrishnan, R. and Ranganathan, K., 2012, *A Textbook of Graph Theory*, Springer, New York, USA

[2] Diestel, R., 2000, *Graph Theory*, Springer, New York, USA

[3] Zhan, F. B., Noon, C. E.,1998, Shortest Path Algorithms: An Evaluation using Real Road Networks. In *Transportation Science*, Vol. 32, No. 1

[4] Dijkstra, E. W., 1959, A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik 1*, 269-271.

[5] Yen, J. Y., 1971, Finding the K shortest Loopless Paths in a Network. *Management Science*, Vol. 17, No. 11

[6] Cohen, P. R., Perrault, C.R., and Allen, J.F., 1981, Beyond question answering. In Lehnert W. and Ringle M., eds., *Strategies for Natural Language Processing*. LEA.

[7] Pentney, W., Popescu, A., Wang, S., Kautz, H., and Philipose M., 2006, Sensor-Based Understanding of Daily Life via Large-Scale Use of Common Sense. In *Proc. AAAI-06*.

[8] Yang, Q., 2009, Activity Recognition: Linking low-level sensors to high-level intelligence. In *Proc. IJCAI-09*, 20-26.

[9] Ramirez M., Geffner H., 2009, Plan Recognition as Planning. In *Proc. 21st Intl. Joint Conf. on Artificial Intelligence*, 1778–1783. AAAI Press.

[10] Ramirez M. and H. Geffner, 2010, Probabilistic plan recognition using off-the-shelf classical planners. In *Proc. AAAI-10*. AAAI Press.

[11] Keren, S., Gal, A., Karpas, E., 2014, Goal recognition design. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling*. pp. 154–162.