Alma Mater Studiorum · Università di Bologna
Campus di Cesena

SCIENCES AND ENGINEERING
THREE-YEAR DEGREE COURSE IN COMPUTER SCIENCE AND ENGINEERING

Analysis and Implementation of Algorithms for Bicriteria Shortest Paths Problems

Elaborato in
RICERCA OPERATIVA

Supervisor:
Cha.mo Prof.
VIGO DANIELE

Co-supervisor:
NÉRON EMMANUEL

Student:
ROSSOLINI ANDREA

Academic Year: 2018/2019
III session

Abstract

In this paper a *pathfinding* problem is analyzed starting from a classical shortest path problem and then, after several optimization, going to resolve a graph that utilizes two static weights on its arcs, considering the most full satisfying set of solutions.

As well known, Dijkstra's algorithm is the most widely used to solve routing problems; in fact is very easy to create an implementation that attempts to find the best path in a classical weighted graph. So the paper will focus on the operations of optimization. The main part of the paper is the one that analyzes the paths of a graph with two weights for each arc, one value represents the distance (also present in the mono-criteria problem) and the other represents the danger of that arc. So the implementation will not only find the shortest and safest path, but also all the paths (not dominated) that take intermediate values.

Contents

Chapter 1

Introduction

Dijkstra's algorithm is widely used to find shortest path in routing problems that use graphs with not-negative and static values. But this algorithm takes into consideration only one "dimension" of costs; for example, to calculate a path from the source node to the destination, distance is the result of adding up the length between two nodes segment by segment. But the problem faced in this study considers two criteria for choosing the wanted path: the first cost defines the distance, while the second one represents the danger. Simply applying Dijkstra's algorithm it is possible to found a path very short, but it might be very dangerous, or vice versa. Multicriteria shortest path problem aims to determine a path that optimizes the costs from a source to the target. In general, there is not a single optimal solution; the goal of this project is to determinate a set of feasible and not-dominated solutions, finding different paths based on two criteria, so not only minimizing either distance or danger, but the relations between this two values.

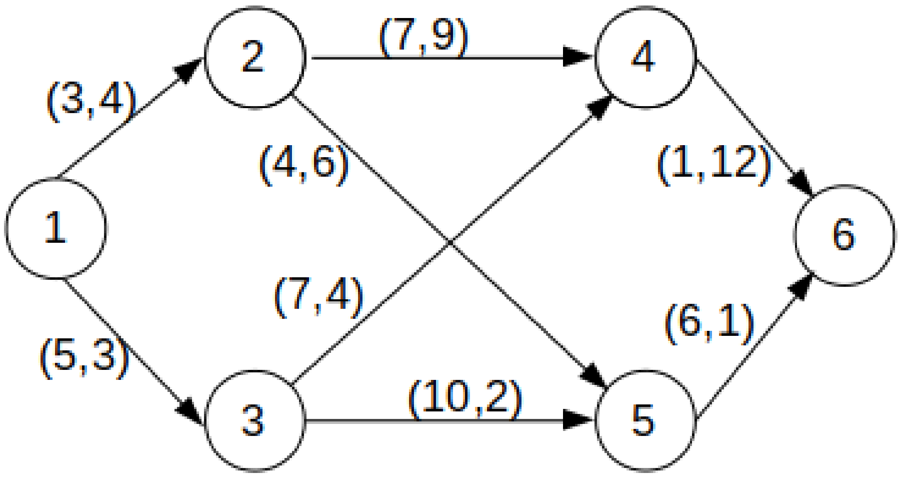


Figure 1.1: Graph example.

$p1$	$1 \to 2 \to 4 \to 6$	(11, 25)
$p2$	$1 \to 2 \to 5 \to 6$	(13, 11)
$p3$	$1 \to 3 \to 5 \to 6$	(21, 6)
$p4$	$1 \to 3 \to 4 \to 6$	(13, 18)

Table 1.1: Graph's solution

In fig.1.1 an example of a double criterion's routing is shown. The nodes are labeled with numbers $\{1, 2, ..., 6\}$ and the weights of the arcs are represented as a pair of value $(a,\ b)$, where a is distance and b is danger.

In table 1.1 all the possible graph's iterations are shown: $p1$ and $p3$, respectively the shortest and the safest paths, $p2$ is a not-dominated solution (it is longer than $p1$ but safer and more dangerous then $p3$, but shorter) and $p4$ is a dominated path (because it is longer and safer than $p1$, most dangerous and shorter than $p3$, but it has no advantage over $p2$ which has the same value of distance but is safer) so it is useless to us (for the formal definition see ch.3).

1.1 Mathematical formulation

Let $G(N, A)$ denotes direct network which is composed of a finite set $N = \{0, 1, \dots, n\}$ of nodes and a finite set $A \subseteq N \times N$, that represents the set of directed edges. Each arc can be denoted as an ordered pair (i, j), where $i \in N$, $j \in N$ and both are two different nodes in $G(N, A)$.

Let define $c^k_{i,j}$ where $(i,j) \in A$ and $1 \leq k \leq 2$ (because it is a double criterion problem) represent the cost which referring to. Let's define two nodes in the graph: s and t, where $s \in N$ and $t \in N$, these are respectively the *source* and *target* of which one or more paths are searched. A path $p_{s,t}$ can be qualified as a sequence of alternating nodes and arcs: $p_{s,t} = \{s, (s, i_1), i_1, \ldots, i_l, (i_l, t), t\}$.
So it is possible to say that each $c^k_{i,j}$ refers to one of the two costs of each arc (i,j), therefore the total cost of the entire path can be represented in this way:

$$(c^1(p_{s,t}), c^2(p_{s,t})) \tag{1.1}$$

$$c^1(p_{s,t}) = \sum_{(i,j)\in p} c^1_{i,j} \tag{1.2}$$

$$c^2(p_{s,t}) = \sum_{(i,j)\in p} c^2_{i,j} \tag{1.3}$$

Our purpose is to **minimize** the (1.2) to find the shortest path or the (1.3) to find the safest one.

Chapter 2

Mono-criteria algorithms

In this chapter the algorithms which concern the single criteria routing will be explained. The analysis focuses on the evolution and optimization of the Dijkstra's algorithm, explaining some implementation choices.

2.1 Dijkstra's algorithms

Dijkstra's algorithm is a very famous algorithm used to find the shortest paths between nodes in a graph connected by arcs with positive weights.
Different implementation of this algorithm are present in this paper. In the following a brief explanation of the implementations referred in this paper.

2.1.1 One to all

This implementation is useful to find **all the shortest paths** from a source node to each other graph's nodes. The starting node will save a dictionary where there is a key for each reached node and the respective value of distance. It implements a *priority queue* so the complexity of this implementation is $O((|N| + |A|)\log_2(|N|))$ where N is the number of vertices and A the number of edges. In the worst case, so where $A >> N$, the time complexity is $(|A|\log_2(|N|))$.

This implementation explores all the nodes reachable from the source; so it doesn't stop until the graph is totally explored.

2.1.2 One to one

This implementation focuses to find the shortest path between the node source and the target, using the classical implementation of Dijkstra's algorithm.
The difference with "One to all" is that this doesn't use a priority queue, but the algorithm will interrogate each not-visited node every loop, selecting

```
function DijkstraAlgorithm(source, target):
    dist[source] $\leftarrow 0$
    create priorityQueue Q
    q.put(source, dist[source])
    while Q is not empty do
        n $\leftarrow$ Q.extract_min()
        for each neighbor in v of n do
            alt $\leftarrow$ dist[n] + lenght(n, v)
            if alt < dist[v] do
                dist[v] $\leftarrow$ alt
                prev[v] $\leftarrow$ n
                Q.put(v, alt)
    return dist, prev
```

Figure 2.1: Pseudocodice dell'Algoritmo di Dijkstra (list of candidate)

the nearest node; this is very time consuming, in fact the complexity of this implementation is $O(|N^2|)$.

2.1.3 List of candidate

The last version of Dijkstra's algorithm is another implementation that uses a priority queue. The elements of this queue are inserted time to time by each new visited node, using the value of distance as priority value; so the queue's elements are the neighbors of the visited nodes, each time a node is visited it is removed from the queue. In this way the algorithm needs to interrogate only some nodes and not all the graph.
The list of candidate algorithm has the same *worst-case complexity* of the *One to all* algorithm: $O((|N| + |A|) \log_2(|N|))$.

2.2 "A Star" algorithm

The A Star algorithm (or 'A*') can be considered an extension of Dijkstra's algorithm, because it achieves better performance and accuracy by using heuristics[1]. A*, to determinate how to extend its paths to the target, needs to minimizes this equation:

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

Where:

- n is the next path's node

[1]https://en.wikipedia.org/wiki/Heuristic_(computer_science)

- $g(n)$ is path's cost from the beginning to n
- $h(n)$ is the heuristic function

Heuristic, in this case, is the shortest distance from n to the goal, so a *straight-line* or better the **euclidean distance** to the target. According with [2] the time complexity is related to h and the number of nodes explored is exponential in the depth of the shortest path solution. So the worst case is $O(|N|) \equiv O(b^d)$ where b is the average number of successors per node and d the depth of the solution.

Implementation's details

At each iteration:

1. The node with the lowest $f(n)$ is popped by the queue (implemented as a priority queue).
2. Update the values of the neighbors and then add them to the queue.
3. The algorithm repeat until the goal is reached.

The Euclidean distance between two points is:

$$\sqrt{(i_x - t_x)^2 + (i_y - t_y)^2}$$

where i is a node of the graph and t the target, x and y are latitude and longitude, converted to Cartesian coordinates.

To calculate the square root like this is very expensive, in term of time, so it is necessary to find a way for make this operation only one time per node (in this implementation the value of Euclidean distance is stored in an node's attribute).

2.3 Analysis of the results

This paragraph shows the principal characteristics and results of each implementation.

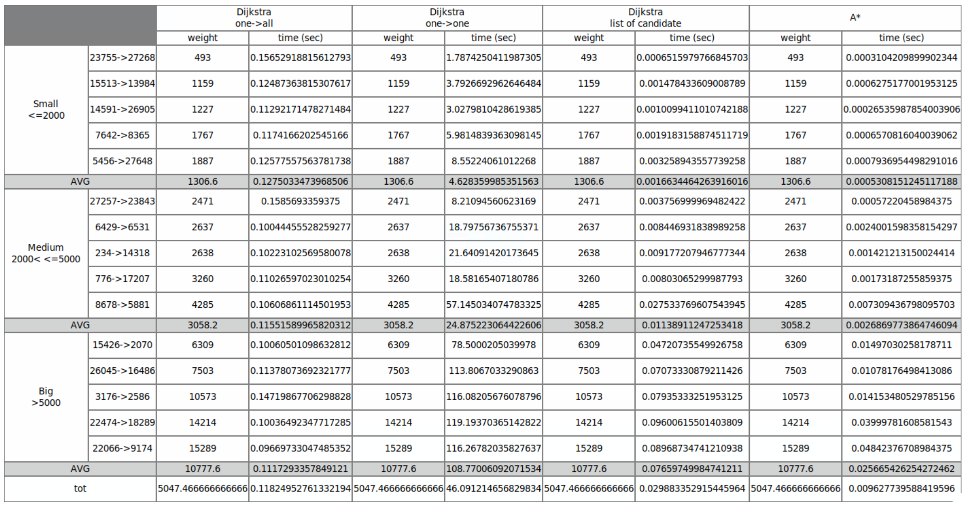
2.3.1 Performance comparison

In the figure 2.3 are shown some results, from 15 different iteration, classified in three "set" that groups different path's length.

[2] `https://en.wikipedia.org/wiki/A*_search_algorithm`

```
function DijkstraAlgorithm(source, target):
    score[source] ← 0
    create priorityQueue Q
    q.put(source, score[source])
    while Q is not empty do
        n ← Q.extract_min()
        for each neighbor v of n do
            if v not visited do
                tmpScore ← score[n] + lenght(v, n)
                if tmpScore < score[v] do
                    if euclidean[v] is None do
                        euclidean[v] ← calcEuclidean(v, target)
                    score[v] ← tmpScore
                    prev[v] ← n
                    Q.put(v, tmpScore)
    return score, prev
```

Figure 2.2: Pseudocodice dell'Algoritmo A*



		Dijkstra one->all		Dijkstra one->one		Dijkstra list of candidate		A*	
		weight	time (sec)	weight	time (sec)	weight	time (sec)	weight	time (sec)
Small <=2000	23755->27268	493	0.15652918815612793	493	1.7874250411987305	493	0.000651597976684570З	493	0.0003104209899902344
	15513->13984	1159	0.12487363815307617	1159	3.7926692962646484	1159	0.001478433609008789	1159	0.0006275177001953125
	14591->26905	1227	0.11292171478271484	1227	3.0279810428619385	1227	0.0010099411010742188	1227	0.00026535987854003906
	7642->8365	1767	0.1174166202545166	1767	5.9814839363098145	1767	0.0019183158874511719	1767	0.0006570816040039062
	5456->27648	1887	0.12577557563781738	1887	8.55224061012268	1887	0.003258943557739258	1887	0.0007936954498291016
AVG		1306.6	0.1275033473968506	1306.6	4.628359985351563	1306.6	0.0016634464263916016	1306.6	0.0005308151245117188
Medium 2000< <=5000	27257->23843	2471	0.1585693359375	2471	8.21094560623169	2471	0.003756999969482422	2471	0.00057220458984375
	6429->6531	2637	0.10044455528259277	2637	18.79756736755371	2637	0.008446931838989258	2637	0.0024001598358154297
	234->14318	2638	0.10223102569580078	2638	21.64091420173645	2638	0.009177207946777344	2638	0.001421213150024414
	776->17207	3260	0.11026597023010254	3260	18.58165407180786	3260	0.00803065299987793	3260	0.00173187255859375
	8678->5881	4285	0.10606861114501953	4285	57.145034074783325	4285	0.027533769607543945	4285	0.007309436798095703
AVG		3058.2	0.11551589965820312	3058.2	24.875223064422606	3058.2	0.01138911247253418	3058.2	0.0026869773864746094
Big >5000	15426->2070	6309	0.10060501098632812	6309	78.5000205039978	6309	0.04720735549926758	6309	0.014970302581787 11
	26045->16486	7503	0.11378073692321777	7503	113.8067033290863	7503	0.07073330879211426	7503	0.01078176498413086
	3176->2586	10573	0.14719867706298828	10573	116.08205676078796	10573	0.07935333251953125	10573	0.014153480529785156
	22474->18289	14214	0.10036492347717285	14214	119.19370365142822	14214	0.09600615501403809	14214	0.03999781608581543
	22066->9174	15289	0.09669733047485352	15289	116.26782035827637	15289	0.08968734741210938	15289	0.04842376708984375
AVG		10777.6	0.1117293357849121	10777.6	108.77006092071534	10777.6	0.07659749984741211	10777.6	0.025665426254272462
tot		5047.466666666666	0.11824952761332194	5047.466666666666	46.091214656829834	5047.466666666666	0.029883352915445964	5047.466666666666	0.009627739588419596

Figure 2.3: Performance of the mono-criteria algorithms (generate with tk-inter library)

From the table it's possible to recognize that the first column, '*one to all*', has a pretty much constant elaboration time (it comprehends the algorithm's work completion and the research in target node's attribute the distance from source). From the data of *one to one* algorithm is easily to recognize that the implementation without a priority queue is too slow, especially in the longest paths. Comparing the total average times of the latest two algorithm is possible to see that the *A Star*'s implementation is 3 times more faster than the *List of candidate*'s implementation. *Dijkstra's List of candidate*, in average, is near to be five times more performing than the *one to all* implementation.

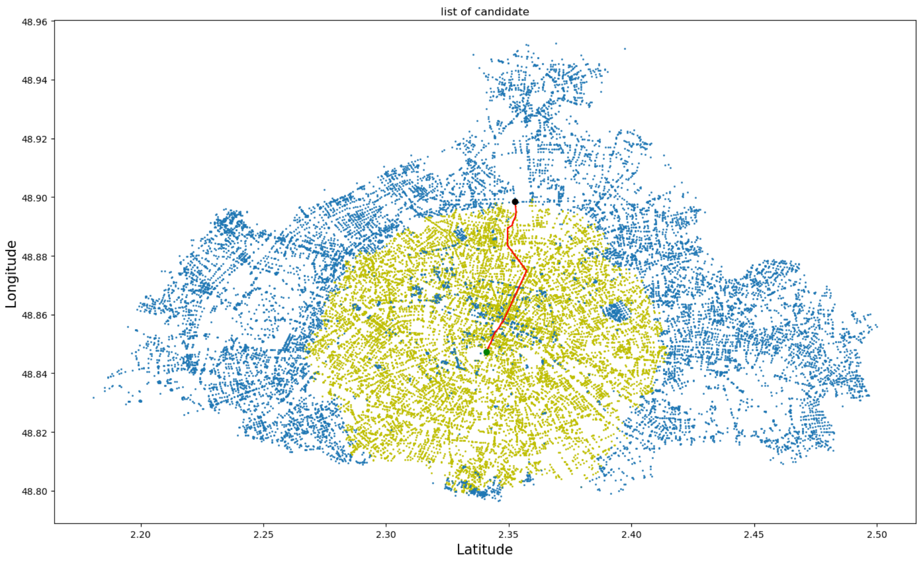
2.3.2 Node visited

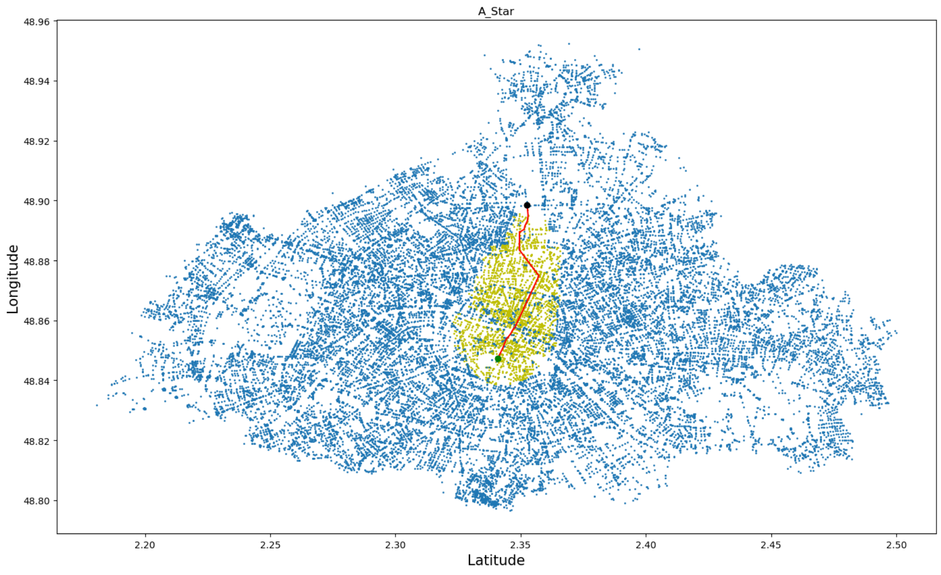
All the implementation are going to find the same path between nodes; in particular all the implementation (except for the *A Star*'s algorithm) explore the same nodes, instead, the *A**, explores less number of nodes, this means that it makes less loops during the elaboration and research of the shortest path.

In the following pictures it is shown how, researching the shortest path, the algorithms visit nodes in a different way:
Dijkstra's algorithm (*list of candidate*) makes a research more "circling" around the source (big green point), instead the 'A Star', for its nature, is more direct and it reaches the target exploring an "oval" between the starting nodes and the target. So it's easy to see and understand how A* is more efficient than the others implementation.

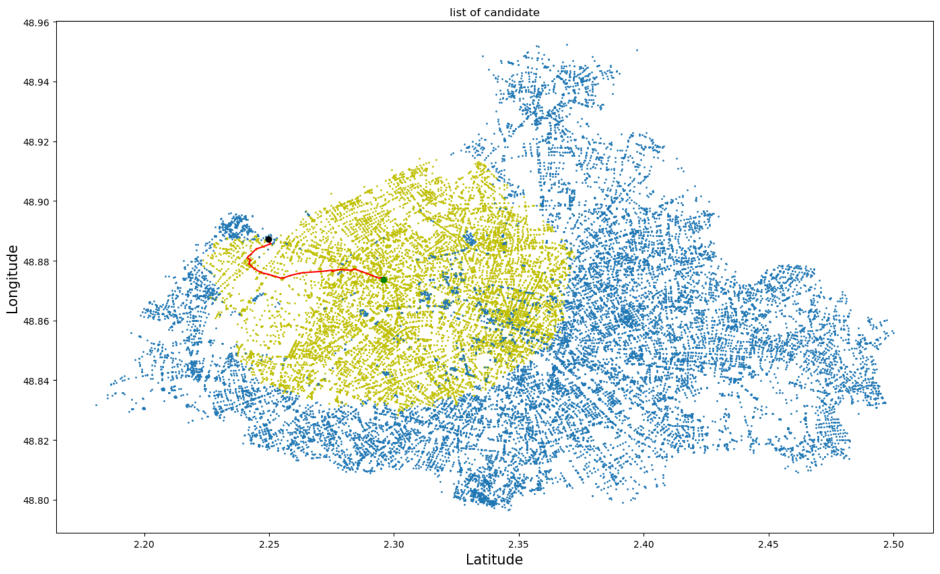
***Legend*:**

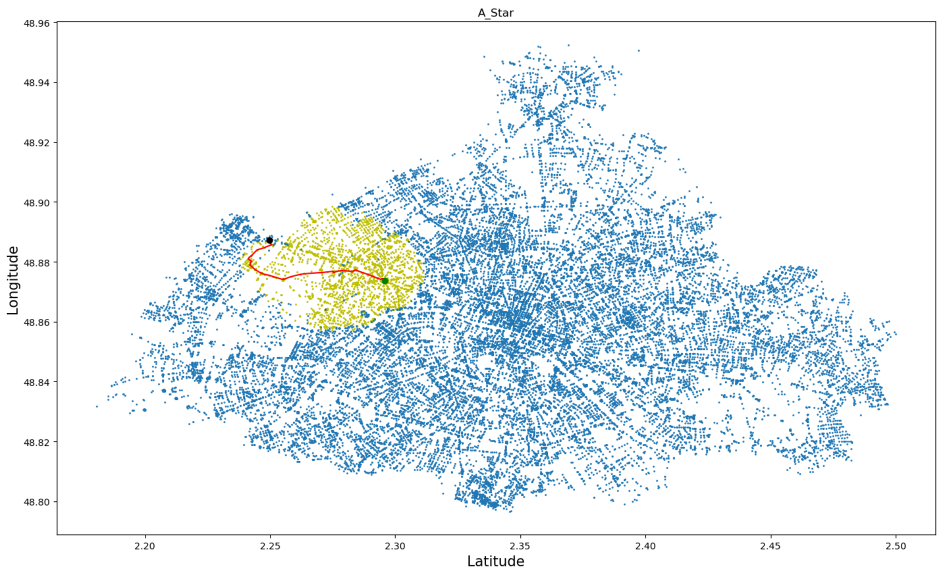
- **Blue**: Nodes of graphs.
- **Yellow**: Nodes visited.
- **Green**: Source node.
- **Black**: Target node.
- **Red line**: Path.



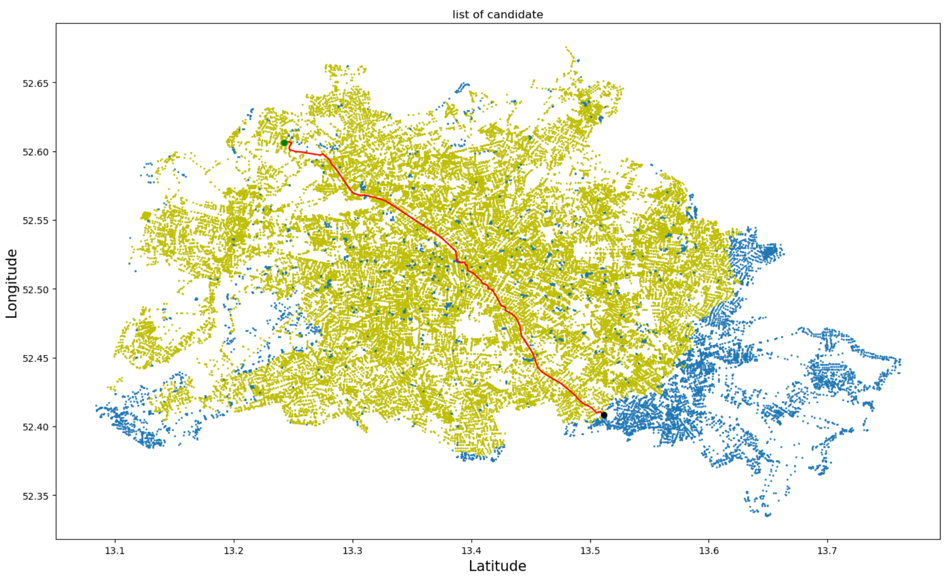


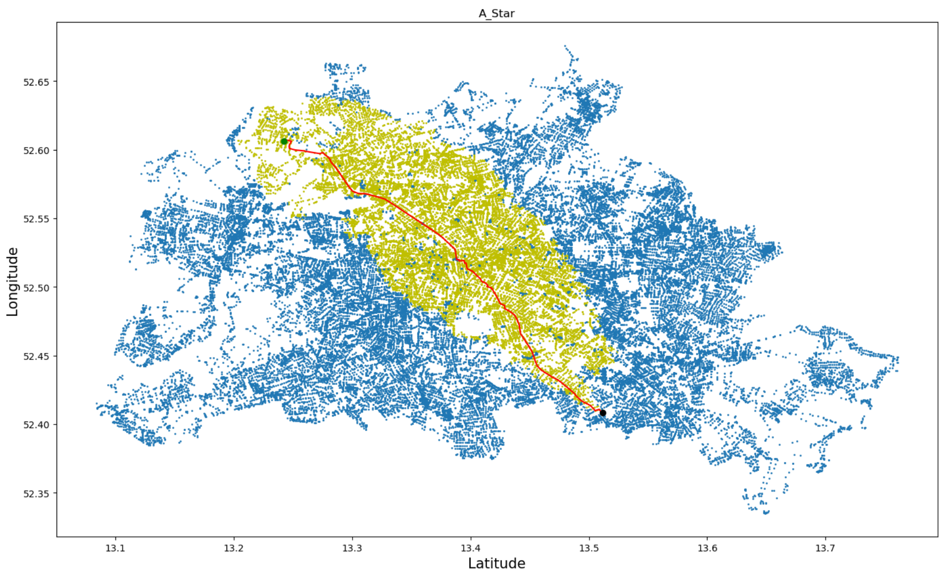
Source: 2070 → Target: 15426
Map: Paris





Source: 2000 → Target: 2689
Map: Paris





Source: 1000 $\rightarrow$ Target: 1510
Map: Berlin

Chapter 3

Bicriteria algorithms

In this chapter the algorithms implemented to studying the case of a graph with arches having two different weights are explained. Keeping in mind the mathematical explanation made in the introduction, it is possible to enunciate some definitions:

Definition 1. Feasible solution
Let x, y be two distinct feasible path from a source s to a target t. It is said that x *dominates* y if and only if $c^k(x) \leq c^k(y)\ \forall\ k \in \{1, 2\}$.

Definition 2. Convex hull
The **convex hull** of a set X of feasible solutions is the smallest **convex set** that contains X. In this case is defined by the equation

$$\alpha * c^1 + (1 - \alpha)c^2 = k \tag{3.1}$$

where k is constant and $\alpha \in \{0, 1\}$.
and c^1 is the *distance* and c^2 the *danger*.

Definition 3. Pareto front
The **Pareto front** is a set of optimal solutions, so consisting of a set of not-dominated points (there's no other point that has better values at the same time for each point's criteria).
A point can be part of the Pareto front without dominate any other points.

The figures 3.3 and 3.4 give a graphic representation of the definitions 2 and 3.

3.1 Dijkstra applied to bicriteria

According with the function 3.1, it is possible to say that "bicriteria graph" $G = (N, V, dist, dang)$, where *dist* is c^1 and *dang* is c^2, is reducible to $G = (N, V, \alpha * dist + (1 - \alpha)dang)$ the problem can be addressed as the previous case, so using Dijkstra's algorithm (list of candidates) giving a value to α.

It is easy to deduce that the more value of α approaches 0 the more it is possible to find the safest paths. On the contrary the more α is approaching 1 the more shortest paths will be found.

The algorithm's implementation is similar to Dijkstra's list of candidate, but with the difference that this version needs in input, together with the source and the target, the value of α; then, using (3.1), the algorithm will choose the optimal path.

```
function DijkstraAlgorithm(source, target, alpha):
 score[source] ← 0
 create priorityQueue Q
 Q.put(source, score[source])
 while Q is not empty do
     n ← Q.extract_min()
     for each neighbor v of n do
         alt ← alpha*firstW(n, v) + (1 - alpha)*secondW(n, v)
         alt ← alt + score[u]
         if alt < score[v] do
             score[v] ← alt
             prev[v] ← n
             Q.put(v, tmpScore)
 return score, prev
```

Figure 3.1: Pseudocodice dell'Algoritmo di Dijkstra con due criteri

3.1.1 Application of Bicriteria Dijkstra

To use the bicriteria Dijkstra's algorithm, it is needed to find a way to call the algorithm several times, with different values for α

Bicriteria Dijkstra iteration

This is a very raw version, its operation is based on recalling the bicriteria Dijkstra's function, several time with different values for α, increasing its value by a constant factor called "**precision**".

The required elaboration time is related to the chosen precision, because it has to call the same function a lot of time and probably with the same result. So more precision means more results (*feasible paths*), but with longer elaboration times.

Bicriteria Dijkstra with binary research

This version is an “evolution” of the Bicriteria Dijkstra iteration, because it uses a *binary research* algorithm. It is based on calling the function at least two time: with $\alpha = 0$ and $\alpha = 1$, then, if the results are different, with $\alpha = 0.5$, again, if the result is different with $\alpha = \alpha/2$, and so on and so on; until there’s no more solution.

This version generally is more efficient and precise than the *iteration* algorithm. The output on figure 3.2 shown that the *Bicriteria Dijkstra iteration* with a precision of (e.g.) 0.2 is pretty faster than the binary search; while, with a smallest value for precision (little value means more paths), the *Bicriteria iteration* becomes slower but finds more results, but anyway it finds less results than binary search.

`img/biCriteriaOutput.png`

Figure 3.2: Output of the two version of Dijkstra’s bicriteria algorithm.

3.2 Label-setting algorithm

In the paper was told that the algorithm using Dijkstra is not able to find all feasible solutions, but only those that make the *Convex hull*; so this algorithm is useful for finding all the set of non-dominated solutions, hence the *Pareto front*. Below a comparison of bicriteria Dijkstra and label-setting algorithm using two graphs for clarifying the concept.

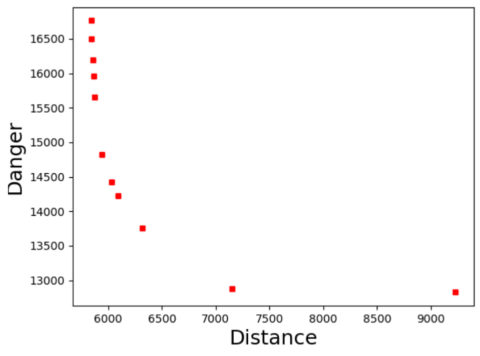


Figure 3.3: Graph generated by *Bicriteria Dijkstra with binary search*

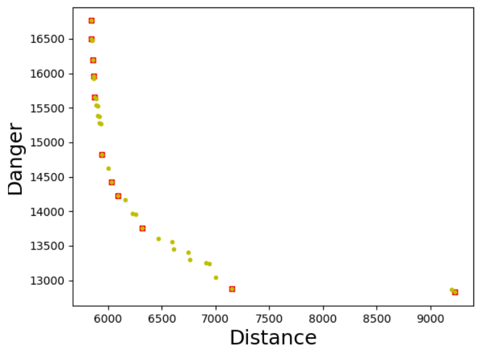


Figure 3.4: Graph generated by *Label-setting algorithm*

This example has been calculated using the map of Paris, starting from node 2000 to node 2689.

In the graph 3.3 are present only some point of the *Convex hull* (red squares) found using bicriteria Dijkstra, while the second graph (3.4) has much more points (yellow points) and, as it is possible to see, the label-setting algorithm has found also the points found by Bicriteria Dijkstra too (*Convex hull*). The solutions found by label-setting algorithm are all feasible, hence it is more complete.

3.2.1 Implementation

Each node of the graph store a set of label with all the useful information "inside" its; the structure of label is as follow:

$$(distance, danger, owner, predecessor, ownerIndex, predIndex)$$

The first two parameter are the costs of the path from the source, the third and four elements indicate the node that owns the label and from which node it comes, the fifth element is the label's position inside the label set of the owner node, the sixth indicate the position of parent's label inside its label set (used for backtracking). As always happens the algorithm uses a priority queue where labels are stored and then chosen in function of the distance's value. The algorithm works in this way:

i. Create first label $(0,\ 0,\ s,\ null,\ 0,\ null)$ and put in the priority queue (s represent the starting point).

ii. If the queue is empty perform step (6) otherwise get the label with smallest distance's value from the queue and calculate the label for all its owner's neighbors.

iii. Check if the calculated label is or isn't a non-dominated label. There's can be three different solution:

   (a) The calculated label is dominated, so the algorithm will discards it.

   (b) The calculated label dominate other labels, so those labels can be removed.

   (c) The calculated label doesn't dominate any other label, but it isn't dominated at all, so the algorithm will keep it.

iv. If the calculated label is feasible is put in the queue, using the distance as a criteria for its priority.

v. Return to step (2).

vi. End.

```
function DijkstraAlgorithm(source, target):
    originLabel ← (0, 0, source, Null, 0, Null)
    source.add(originLabel)
    create priorityQueue Q
    Q.put(orginLabel, priority())
    while Q is not empty do
        actualLabel ← Q.extract_min()
        owner ← actualLabel[2]
        for each neighbor v of owner do
            firstW ← weightOne(owner, v) + actualLabl[0]
            secondW ← weightTwo(owner, v) + actualLabel[1]
            vIndex ← v.labelIndex()
            predIndex ← owner.labelIndex()
            label ← (firstW, secondW, v, owner, vIndex, predIndex)
            if label is not dominated by v.labels() do
                v.labelsAdd(label)    //rimuove i label dominati
                if v is not target do
                    Q.put(label, priority())
```

Figure 3.5: Pseudocodice dell'Algoritmo Label-setting

Example:

Is possible to study the graph in figure 1.1 starting from node 1, so this node will have the label (0, 0, 1, *null*, 0, *null*) in its set. Then creating labels for its neighbors (2 and 3): (3, 4, 2, 1, 0, 0) and (5, 3, 3, 1, 0, 0) and put both in the queue. Extracting label owned by node 2 and do the same thing as before and so on, until the target node is reached. At any new label should be studied if it is feasible or not before putting it in the queue.

3.2.2 Lower bound improvement

The original thought was to make a preprocessing phase that would have retraced the path backwards and, in this way, calculates the distance (and danger) from the target to each visited node, to know in advice which would have been the lowest value for each of them. But i noticed that this thought doesn't work with graphs with edges between nodes oriented in more directions with different values for each direction. So I implemented a different solution, below there's the explanation.

This algorithm use a technique that try to improve the speed. This technique consist in a *preprocessing phase*, where, with Bicriteria Dijkstra algorithm, is possible to calculate the safest and the shortest path from the source to the target node and, for each visited node, store information about the distance and the danger of the visited node from so far. To calculate those values it uses Bicriteria Dijkstra with $\alpha = 1$ to find the optimal values for distance, and $\alpha = 0$ for the danger.

How it can be useful?
Is known that the research of the shortest and the safest path will generate the best value for distance and danger for each visited node. So, once calculated the shortest path and the safest, during the elaboration of label-setting algorithm, will be known the following information: the total value of the full path's distance (and danger), *preprocessing-visited node*'s best value of distance (and danger), the value of distance (and danger) calculated during the running of the algorithm.

Is possible to do this evaluation:

- pd = best path's distance.
- nd = best node's distance.
- ad = actual node's distance, calculated at that exact moment by the algorithm.
- td = temporary value of distance.

- res = future value of distance for that specific node.

$$td = pd - nd$$

$$res = td + ad$$

Doing the same thing with danger is possible to find the projection of the final label of that specific node and valuate if it is dominated or not.

3.2.3 Complexity

The worst-case example is represented in the graph below, where there is the remote possibility to find always feasible node for each iteration, that is when one weight is zero and the other is bigger than the last one, for each arc:

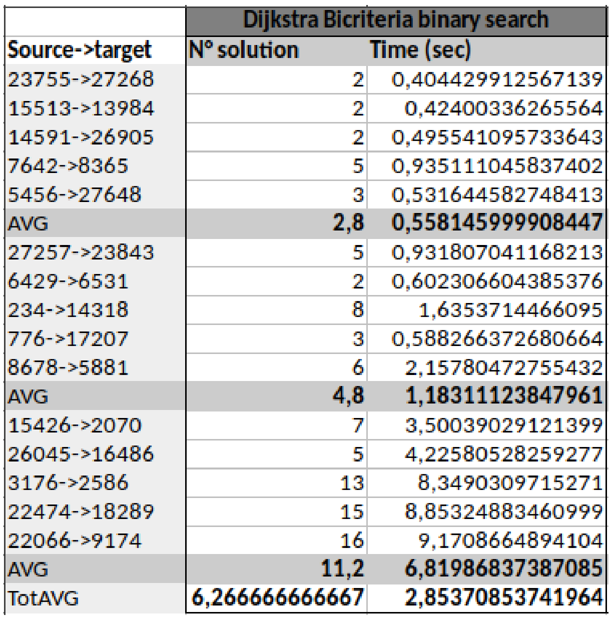
img/labelSettingComplexity.png

Figure 3.6: Worst-case label-setting algorithm's example

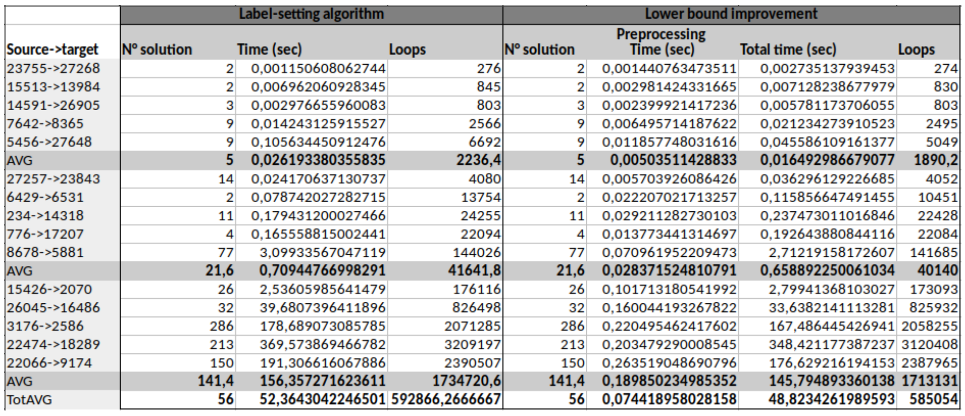
Is now possible to say that the complexity is $O(2^{n-1})$ where n is the number of nodes.

3.3 Analysis of results

Now it is possible to analyze the results. In the figure below (3.4.2) is possible to see an extract generated by the execution of the three algorithms explained before. Is possible to note that the nodes used for the execution are the same used in monocriteria's execution (randomly generated), in this way is possible to make some comparison, but is obvious that the monocriteria is faster than bicriteria.



	Dijkstra Bicriteria binary search	
Source->target	N° solution	Time (sec)
23755->27268	2	0,404429912567139
15513->13984	2	0,42400336265564
14591->26905	2	0,495541095733643
7642->8365	5	0,935111045837402
5456->27648	3	0,531644582748413
AVG	**2,8**	**0,55814599908447**
27257->23843	5	0,931807041168213
6429->6531	2	0,602306604385376
234->14318	8	1,6353714466095
776->17207	3	0,588266372680664
8678->5881	6	2,15780472755432
AVG	**4,8**	**1,18311123847961**
15426->2070	7	3,50039029121399
26045->16486	5	4,22580528259277
3176->2586	13	8,3490309715271
22474->18289	15	8,85324883460999
22066->9174	16	9,1708664894104
AVG	**11,2**	**6,81986837387085**
TotAVG	**6,26666666667**	**2,85370853741964**



	Label-setting algorithm			Lower bound improvement			
Source->target	N° solution	Time (sec)	Loops	N° solution	Preprocessing Time (sec)	Total time (sec)	Loops
23755->27268	2	0,001150608062744	276	2	0,001440763473511	0,002735137939453	274
15513->13984	2	0,006962060928345	845	2	0,002981424331665	0,007128238677979	830
14591->26905	3	0,002976655960083	803	3	0,002399921417236	0,005781173706055	803
7642->8365	9	0,014243125915527	2566	9	0,006495714187622	0,021234273910523	2495
5456->27648	9	0,105634450912476	6692	9	0,011857748031616	0,045586109161377	5049
AVG	**5**	**0,026193380355835**	**2236,4**	**5**	**0,00503511428833**	**0,016492986679077**	**1890,2**
27257->23843	14	0,024170637130737	4080	14	0,005703926086426	0,036296129226685	4052
6429->6531	2	0,078742027282715	13754	2	0,022207021713257	0,115856647491455	10451
234->14318	11	0,179431200027466	24255	11	0,029211282730103	0,237473011016846	22428
776->17207	4	0,165558815002441	22094	4	0,013773441314697	0,192643880844116	22084
8678->5881	77	3,09933567047119	144026	77	0,070961952209473	2,71219158172607	141685
AVG	**21,6**	**0,70944766998291**	**41641,8**	**21,6**	**0,028371524810791**	**0,658892250061034**	**40140**
15426->2070	26	2,53605985641479	176116	26	0,101713180541992	2,79941368103027	173093
26045->16486	32	39,6807396411896	826498	32	0,160044193267822	33,6382141113281	825932
3176->2586	286	178,689073085785	2071285	286	0,220495462417602	167,486445426941	2058255
22474->18289	213	369,573869466782	3209197	213	0,203479290008545	348,421177387237	3120408
22066->9174	150	191,306616067886	2390507	150	0,263519048690796	176,629216194153	2387965
AVG	**141,4**	**156,357271623611**	**1734720,6**	**141,4**	**0,189850234985352**	**145,794893360138**	**1713131**
TotAVG	**56**	**52,3643042246501**	**592866,2666667**	**56**	**0,074418958028158**	**48,8234261989593**	**585054**

Figure 3.7: Output of all relevant algorithms in terms of number of solution and time spent (excel table generated with *xlsxwriter* library).

The tables have been slightly modified to make reading easier, but the values are original

3.4 Bidirectional Bi-criteria Algorithm

3.4.1 Implementation

```
function BidirectionAlgorithm(source, target):
    create priorityQueue Q[f]   //forward
    create priorityQueue Q[b]   //backward
    create list $L_{result}$
    originLabel $\leftarrow$ (0, 0, source, Null, 0, Null)
    source.add(originLabel)
    Q[f].put(originLabel)
    targetLabel $\leftarrow$ (0, 0, source, Null, 0, Null)
    source.add(targetLabel)
    Q[b].put(targetLabel)
    while [$min_i$(Q[f]) + $min_i$(Q[b])]
                        is not dominated by any R$\in L_{results}$ do
        d $\leftarrow$ getDirection()    //forward o backward
        actualLabel $\leftarrow$ Q[d].extract_min()
        owner $\leftarrow$ actualLabel[2]
        for each neighbor v of owner do
            firstW $\leftarrow$ weightOne(owner, v) + actualLabl[0]
            secondW $\leftarrow$ weightTwo(owner, v) + actualLabel[1]
            vIndex $\leftarrow$ v.labelIndex()
            predIndex $\leftarrow$ owner.labelIndex()
            label $\leftarrow$ (firstW, secondW, v, owner, vIndex, predIndex)
            if label is not dominated by v.labels(d) do
                v.labelsAdd(label, d)   //rimuove i label dominati
                Q[d].put(label)
                if v.labels(!d) is not empty do   //!d = direzione opposta
                    result $\leftarrow$ combine(label, v.labels(!d))
                    $L_{results}$.addResults(results)
```

Figure 3.8: Pseudocodice dell'Algoritmo bidirezionale

Particular Stop Condition Case

3.4.2 Performance comparison

Excluding bicriteria Dijkstra's algorithm, which is advantageous only with nodes very far from each other, at the expense of the number of solution; the other two label-setting implementations results to be very similar; in fact *lower bound improvement* algorithm seems to be advantageous only with nodes in a medium-large distance from each other, in others case is pretty in line with the normal label setting algorithm. It is possible also to see that the *lower bound improvement* makes less loops than the other algorithm, but this difference becomes increasingly irrelevant with the increasing distance between the nodes.

Nodo di arrivo	Soluzioni trovate	Tempo impiegato (sec)	Soluzioni trovate	Tempo impiegato (sec)
	Dijkstra Iter. **prec.** = 0.05		Dijkstra binary search	
10	5	10.06989	5	6.02213
100	23	32.07635	12	11.89208
500	84	96.60857	23	82.71155
700	123	210.80490	27	23.74903
5000	593	664.78488	46	23.74327

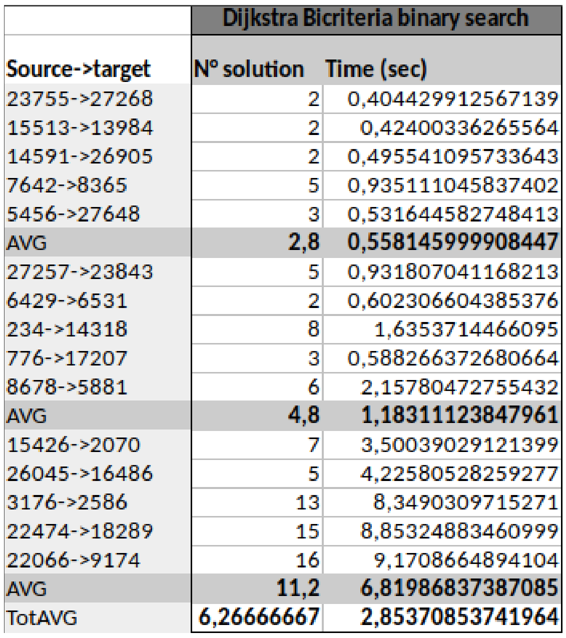
Nodo di arrivo	Soluzioni trovate	Tempo impiegato (sec)	Soluzioni trovate	Tempo impiegato (sec)
	Label-setting		bidirectional algorithm	
10	10	0.00040	4	0.00019
100	116	1.05214	66	0.12038
300	663	98.73498	242	2.4432
500	1867	925.68206	193	22.14023
700	3278	4795.59632	429	141.73064

Nodo di arrivo	Soluzioni trovate	Tempo impiegato (sec)		
		Dijkstra preprocessing	lower bound improvement	lower bound reversed
10	10	0.00011	0.00050	0.00055
100	116	0.00095	0.60899	0.69986
300	663	0.00402	74.03828	87.22029
500	1867	0.00486	930.81834	927.77959
700	3278	0.05327	4887.63521	4931.99105

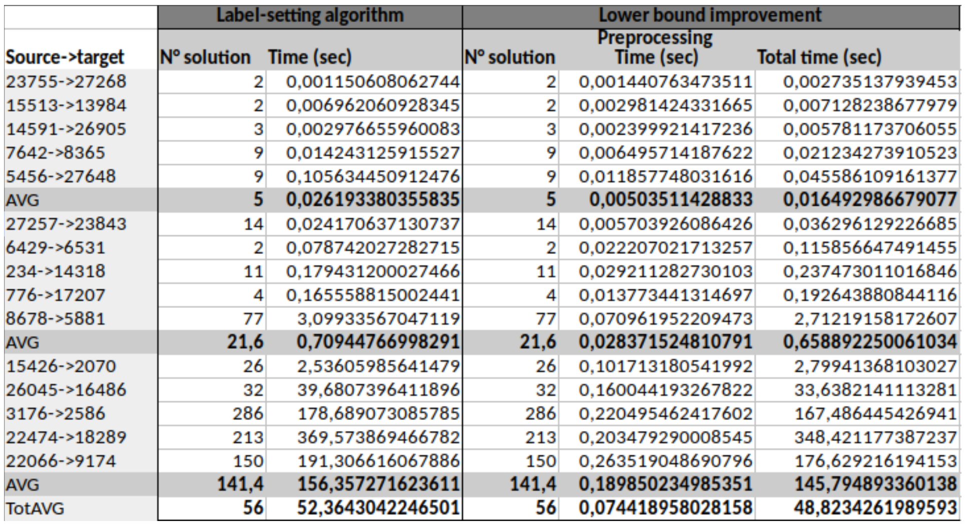
Table 3.2: Risultati degli algoritmi bicriteria in un grafo non orientato con 200000 nodi connessi a cascata tramite archi dai valori casuali (tutti i cammini partono da 0 e i nodi sono collegati ad altri 4 nodi, due antecedenti e due precedenti)

3.4.3 Node visited

Despite the lower bound algorithm does less loops, the visited nodes are less but almost the same than the label-setting algorithm, so only one image is shown for both algorithms, since the difference is almost imperceptible.

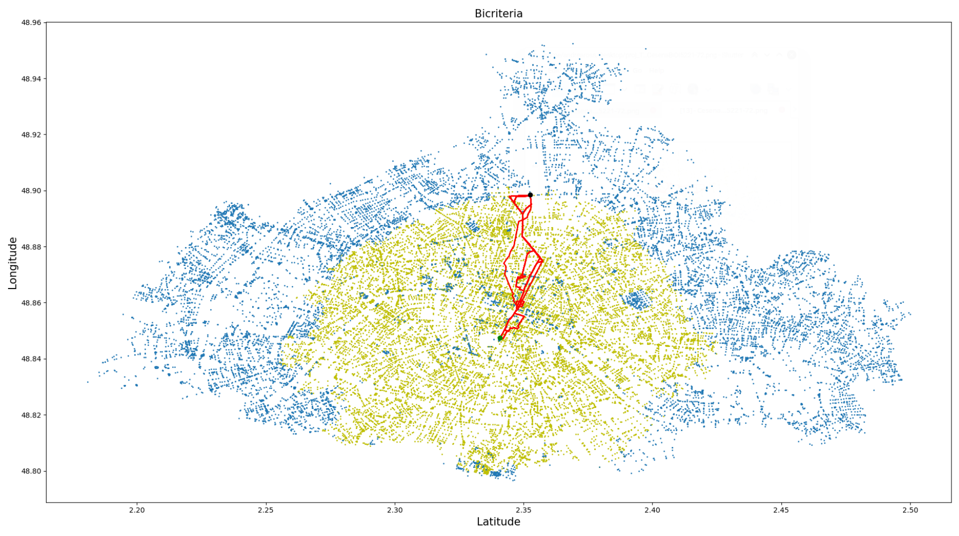


	Dijkstra Bicriteria binary search	
Source->target	**N° solution**	**Time (sec)**
23755->27268	2	0,404429912567139
15513->13984	2	0,42400336265564
14591->26905	2	0,495541095733643
7642->8365	5	0,935111045837402
5456->27648	3	0,531644582748413
AVG	**2,8**	**0,558145999908447**
27257->23843	5	0,931807041168213
6429->6531	2	0,602306604385376
234->14318	8	1,6353714466095
776->17207	3	0,588266372680664
8678->5881	6	2,15780472755432
AVG	**4,8**	**1,18311123847961**
15426->2070	7	3,50039029121399
26045->16486	5	4,22580528259277
3176->2586	13	8,3490309715271
22474->18289	15	8,85324883460999
22066->9174	16	9,1708664894104
AVG	**11,2**	**6,81986837387085**
TotAVG	**6,26666667**	**2,85370853741964**



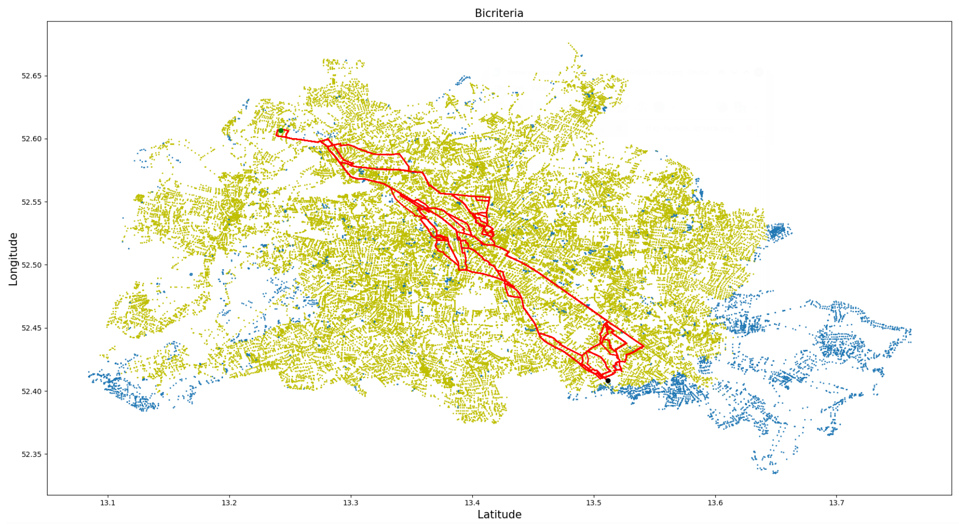
	Label-setting algorithm		Lower bound improvement		
Source->target	**N° solution**	**Time (sec)**	**N° solution**	**Preprocessing Time (sec)**	**Total time (sec)**
23755->27268	2	0,001150608062744	2	0,001440763473511	0,002735137939453
15513->13984	2	0,006962060928345	2	0,002981424331665	0,007128238677979
14591->26905	3	0,002976655960083	3	0,002399921417236	0,005781173706055
7642->8365	9	0,014243125915527	9	0,006495714187622	0,021234273910523
5456->27648	9	0,105634450912476	9	0,011857748031616	0,045586109161377
AVG	**5**	**0,026193380355835**	**5**	**0,00503511428833**	**0,016492986679077**
27257->23843	14	0,024170637130737	14	0,005703926086426	0,036296129226685
6429->6531	2	0,078742027282715	2	0,022207021713257	0,115856647491455
234->14318	11	0,179431200027466	11	0,029211282730103	0,237473011016846
776->17207	4	0,165558815002441	4	0,013773441314697	0,192643880844116
8678->5881	77	3,09933567047119	77	0,070961952209473	2,71219158172607
AVG	**21,6**	**0,70944766998291**	**21,6**	**0,028371524810791**	**0,658892250061034**
15426->2070	26	2,53605985641479	26	0,101713180541992	2,79941368103027
26045->16486	32	39,6807396411896	32	0,160044193267822	33,6382141113281
3176->2586	286	178,689073085785	286	0,220495462417602	167,486445426941
22474->18289	213	369,573869466782	213	0,203479290008545	348,421177387237
22066->9174	150	191,306616067886	150	0,263519048690796	176,629216194153
AVG	**141,4**	**156,357271623611**	**141,4**	**0,189850234985351**	**145,794893360138**
TotAVG	**56**	**52,3643042246501**	**56**	**0,074418958028158**	**48,8234261989593**

Table 3.1: Risultati degli algoritmi applicabili a grafi orientati (I fogli excel sono stati generati per comodità tramite la libreria *xlsxwriter*).



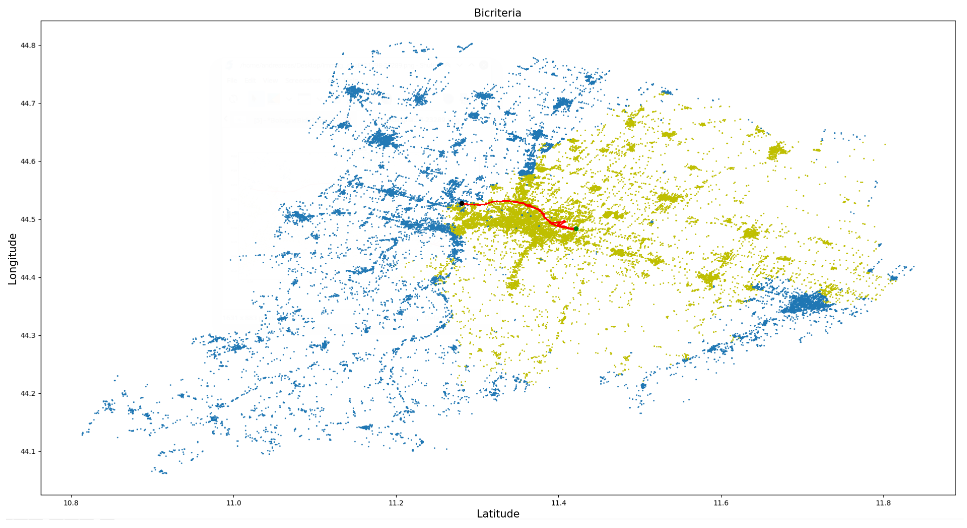
Source: 2070 → Target: 15426
Map: Paris

Figure 3.9: Label-setting algorithm (Paris oriented map).

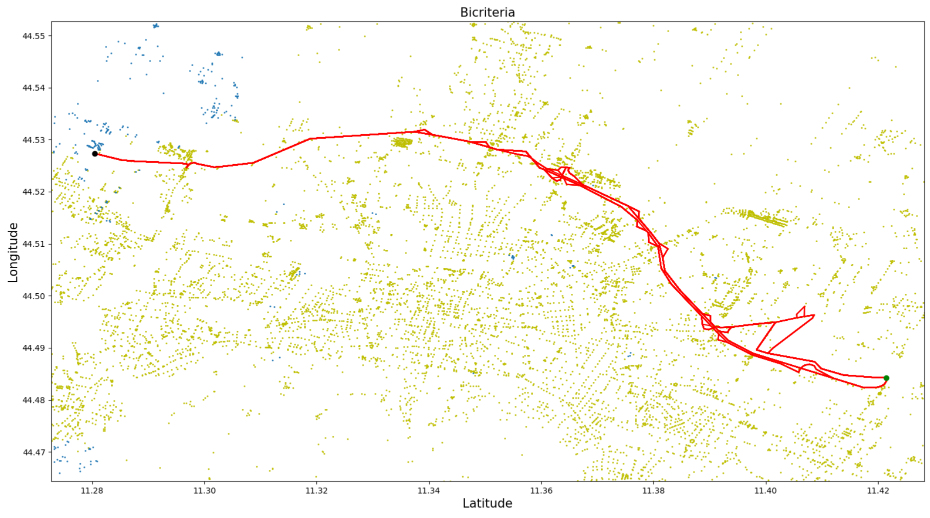


Source: 1000 → Target: 1510
Map: Berlino

Figure 3.10: Label-setting algorithm (Berlin oriented map).

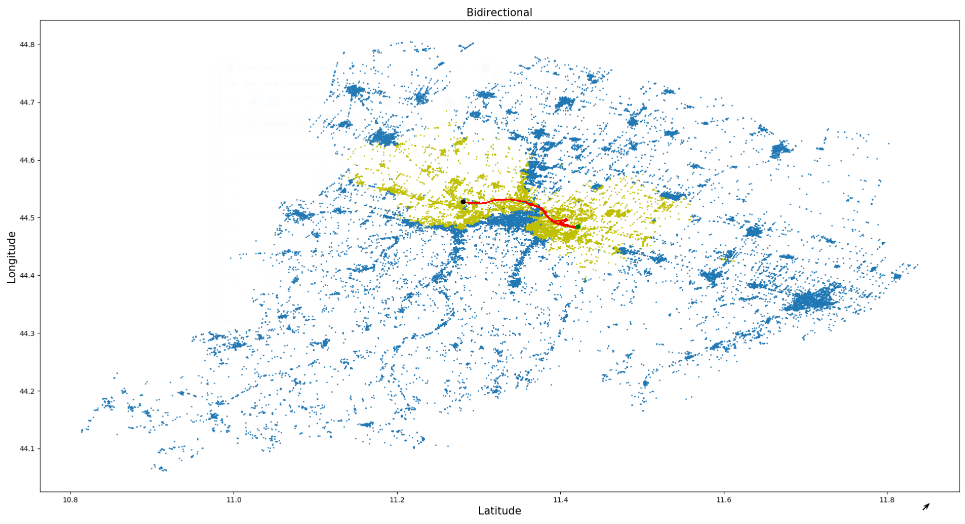


Source: 1298 → Target: 23289
Map: Bologna (Province)

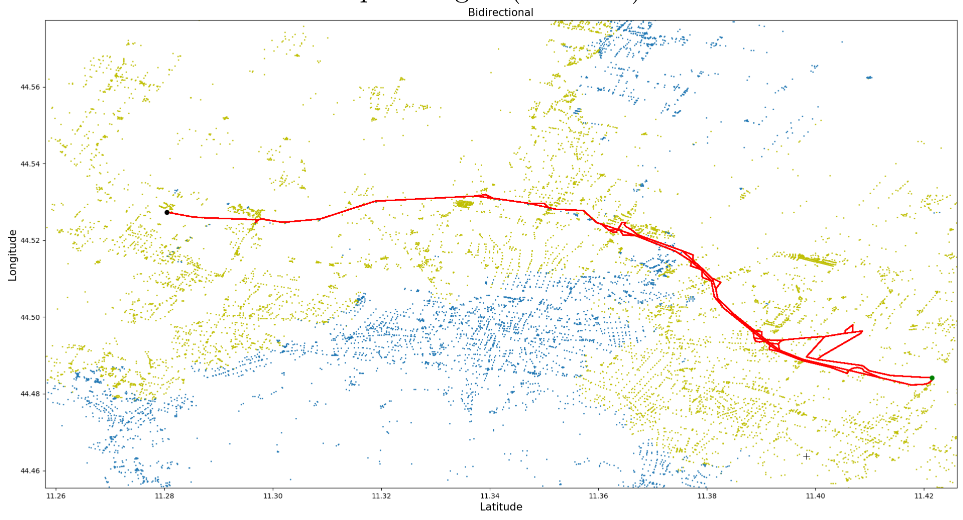


Source: 1298 → Target: 23289
Map: Bologna (Province)
detail

Figure 3.11: Label-setting algorithm Bologna (not-oriented map).



Source: 1298 → Target: 23289
Map: Bologna (Province).



Source: 1298 → Target: 23289
Map: Bologna (Province)
detail

Figure 3.12: Bidirectional algorithm Bologna (not-oriented map).

Chapter 4

Implementation details

This chapter explains some of the main implementation choices made during the development of the project.

4.1 Python

I've decide to choose this language (also because recommended by the professor) because it is a modern and constantly growing language, used in many IT fields. it is very versatile and is incredibly supported by the community, so is possible to find different guides, libraries and tips on the internet.

4.1.1 Binary queue

Theoretically the best implementation of Dijkstra's algorithm is with a *Fibonacci heap* ($O(|N| \log_2 |N| + |A|)$), but according with [1] and after some verification I noticed that using a priority queue, implemented thanks to the *heapq* included in the python standard library, was faster in the extractions and insertions; this because it uses C implementations.

4.2 Nodes as objects

I decided to use the object paradigm for the nodes of the graph, because, due to the nature of the study, it was more convenient to add and remove attributes or functions to the object "node" and simplified my work. Also because it generates a clearer and more readable code.
This implementation use the following attribute in the "object node":

- **index**: Indicates the index of the node.
- **longitude & latitude**.

[1] `https://dnshane.wordpress.com/2017/02/14/benchmarking-python-heaps/`

- **x & y**: Cartesian coordinates of longitude and latitude.
- **neighbors**: A list with all information about the neighbors (distance and node).
- **visited**: Indicates if the node is visited or not.
- **predecessor**: the node predecessor this node.
- **minWeight**: the total weight from the source node.
- **shortestPaths**: stores information about all the nodes from the source to this node. (Used only in *one to all* implementation)
- **euclidean**: stores the euclidean distance from this node to target. (Used only in *A** implementation)
- **distance & danger**: store the values from the source so far. (Used only in bicriteria algorithm).
- **labelList**: A list of all label of this node. (Used only in *label-setting* algorithm).

4.3 Pandas & Matplotlib

For parsing the ".CSV" files I decided to use *Pandas* library. It is not the only library that can interact with CSV files, but it is very recommended for deal with a large amount of data. Pandas is an open-source Python library that provides high performance data analysis tools and easy to use data structures. It is also a key part of the Anaconda distribution and works extremely well in Jupyter notebooks to share data, code, analysis results, visualizations, and narrative text. For more information about *Panda* read this article [2].

Matplotlib is another open-source Python library for 2D graphs based on the famous math library NumPy. Matplotlib has a lot of documentation on the web and it pretty simple to make graphs (I used this library to make all images that represents cities and paths). Provides object-oriented APIs that allow you to insert graphics into applications using the generic GUI toolkit, so is possible to re-use the graphs for other implementations [3].

[2] `https://realpython.com/python-csv/`
[3] `https://matplotlib.org/`

4.4 Testing

I realized two files for *testing in the small* the code, using simple terminal output; one is for the monocriteria algorithms and the other is for the bi-criterias. This two files have a lot of commented code, used for showing different information, like the number of results, backtracking, time, graphs and so on. For a global test and for a good visualization of data I used matplotlib to generate tables (like the one in figure 2.3) and graphs, but also the library *xlsxwriter* to make a rapid visualization table on a spreadsheet (3.4.2).

Chapter 5

Personal considerations

I have developed this project during my Erasmus period, I have hocked spirit and time in order to best meet the requested objective. It was the first time that I did a study of this kind.

5.1 Other possible solution

Genetic algorithm

The following is a personal and unimplemented possible solution, maybe interesting to see and with a possible application or study for the future. It has to be taken only like a conjecture made for pure passion and self-interest.

This conjecture will try to find a solution in a not-polynomial time.

The **first population**, generated with random genes, will travel, starting from the source node, trough the graph guided by the genes. Not all genotype will get to the target, but the ones which arrived at the destination have accumulated a value for the distance and for the danger. The solution that are arrived to the target must be stored, but, unlike what has been done so far, the dominated solutions don't have to be removed. In this way it is possible to avoid *local optima* (a set of solutions that seems to be optimal only within a set of neighbors), it will be the task of the ***fitness functions*** and the *selection* to "clean" solutions. Therefore it is possible to implement two *fitness functions*, one for valuate the distance and one for the danger, associating to all the remaining solution some parameters, based on the results of the fitness and the travel time. Anyway it is not granted that from two good solutions a third one is good too and neither that from two solutions with a certain fitness values it is generated a third one with the same fitness values. To solve these problems generally is used some criteria to chose a set from all the solution and use that for the **crossover**

phase. For example it is often used a selection that attribute a "probability of extraction", based on the fitness function, to some result and then select some of them considering the probability. For time reasons is preferred to chose two or more solution based only on the best fitness, paying attention to chose the shortest and the safest paths. Using methods of ***crossovers*** and ***mutation*** (random changes inside the genotype, useful to improve the fitness function or to improve the research; usually a mutation has a probability value with which it can happen) a new population (generation 2) is generated; now is possible to restart the algorithm with the new population.

Going over with this method, after some iteration, is possible to get a certain number of optimal solutions, which will compose a sort of *Pareto front*.
For terminate the process is possible to put some condition; for example:

- Reached a certain number of solution that satisfied a minimum criteria (e.g. a set of solution reach a certain fitness value).
- Reached a certain number of generations (useful to limit the time consuming).
- Solution too similar for each iteration.

for further information see the bibliography.

5.2 Other possible application

It would be interesting to see also the same algorithm used to study a path that uses, instead of danger, a factor of pollution or energy consumption, to find a set of "eco-paths".

5.3 Difficulties encountered

The main difficulties encountered were, not so much the implementations, but the optimizations of the algorithms to obtain better performance. As well the study behind the developing of the label-setting algorithm, in particular to implement it in order to obtain an efficient backtracking.

5.4 All not-standard python's libraries used:

- pandas
- matplotlib
- tkinter
- xlsxwriter

5.5 Computer info

All tests were performed on a PC with the following characteristics:

– CPU

product:	Intel(R) Core(TM) i5-4300U CPU @ 1.90GHz - 2.90GHz
Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4

Bibliography

[1] J. Abram and I. Rhodes. Some shortest path algorithms with decentralized information and communication requirements. *IEEE Transactions on Automatic Control*, 27(3):570–582, 1982.

[2] J. A. Azevedo and M. Costa. Madeira jjers, and martins eqv (1993) an algorithm from the ranking of shortest paths. *European Journal of Operational Research*, 69:97–106.

[3] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[4] O. Castillo, L. Trujillo, and P. Melin. Multiple objective genetic algorithms for path-planning optimization in autonomous mobile robots. *Soft Comput.*, 11:269–279, 02 2007.

[5] S. Chao. Green routing: An investigation of bicriterion shortest path problems.

[6] L. de Lima Pinto, C. T. Bornstein, and N. Maculan. The tricriterion shortest path problem with at least two bottleneck objective functions. *European Journal of Operational Research*, 198(2):387–391, 2009.

[7] S. Demeyer, P. Audenaert, B. Slock, M. Pickavet, and P. Demeester. Multimodal transport planning in a dynamic environment. In *2008 Intelligent Public Transport Systems (IPTS-2008)*, pages 155–167, 2008.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[9] Y. Disser, M. Müller-Hannemann, and M. Schnee. Multi-criteria shortest paths in time-dependent train networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 347–361. Springer, 2008.

[10] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum*, 22(4):425–460, 2000.

[11] R. Eranki. Pathfinding using a*(a. *Star*, pages 1–5, 2002.

[12] T. Gal. A note on size reduction of the objective functions matrix in vector maximum problems. In *Multiple Criteria Decision Making Theory and Application*, pages 74–84. Springer, 1980.

[13] X. Gandibleux. *Multiple criteria optimization: state of the art annotated bibliographic surveys*, volume 52. Springer Science & Business Media, 2006.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[15] P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *ACM SIGART Bulletin*, (37):28–29, 1972.

[16] J. Legriel, C. Le Guernic, S. Cotton, and O. Maler. Approximating the pareto front of multi-criteria optimization problems. pages 69–83, 2010.

[17] E. d. Q. V. Martins and J. Santos. The labelling algorithm for the multiobjective shortest path problem. *Departamento de Matematica, Universidade de Coimbra, Portugal, Tech. Rep. TR-99/005*, 1999.

[18] E. Q. V. Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[19] J.-A. Meyer and D. Filliat. Map-based navigation in mobile robots:: Ii. a review of map-learning and path-planning strategies. *Cognitive Systems Research*, 4(4):283–317, 2003.

[20] E. Neron, O. Bellenguez-Morineau, and M. Heurtebise. Decomposition method for solving multi-skill project scheduling problem. 2006.

[21] N. J. Nilsson. Principles of artificial intelligence, tioga pub. *Co., Palo Alto, CA*, 476, 1980.

[22] L. L. Pinto and M. M. Pascoal. On algorithms for the tricriteria shortest path problem with two bottleneck objective functions. *Computers & Operations Research*, 37(10):1774–1779, 2010.

[23] A. Raith and M. Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):1299–1331, 2009.

[24] M. Samadi and M. F. Othman. Global path planning for autonomous mobile robot using genetic algorithm. pages 726–730, 2013.

[25] V. Sastry, T. Janakiraman, and S. I. Mohideen. New algorithms for multi objective shortest path problem. *Opsearch*, 40(4):278–298, 2003.

[26] P. Serafini. Some considerations about computational complexity for multi objective combinatorial problems. In *Recent advances and historical development of vector optimization*, pages 222–232. Springer, 1987.

[27] A. J. Skriver and K. A. Andersen. A label correcting approach for solving bicriterion shortest-path problems. *Computers & Operations Research*, 27(6):507–524, 2000.

[28] A. J. Skriver et al. A classification of bicriterion shortest path (bsp) algorithms. *Asia Pacific Journal of Operational Research*, 17(2):199–212, 2000.

[29] A. Stentz. Optimal and efficient path planning for partially known environments. pages 203–220, 1997.

[30] B. S. Stewart and C. C. White III. Multiobjective a. *Journal of the ACM (JACM)*, 38(4):775–814, 1991.

[31] Z. Tarapata. Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *International Journal of Applied Mathematics and Computer Science*, 17(2):269–287, June 2007.

[32] P. Vincke. Problémes multicritères. 1975.