Supervised Project "Bicriteria Paths Problem"

Andrea Rossolini

May 31, 2019

Abstract

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

As is known, Dijkstra's algorithm is 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 I will focus on the operations of optimization. The most important part of the paper is the one that analyze the paths of a graph with two weights for each arches of it, one value represent the distance (also present in the monocriteria problem) and the other represent the danger of that arch. So the implementation will find not only the shortest and safest path, but also all the paths (not dominated) that take intermediate values. Two different algorithms will be shown for the analysis of the bicriteria problem.

Contents

Chapter 1

Introduction

Dijkstra's algorithm, as already mentioned above, is widely used to find shortest path in routing problems that's use graphs with not-negative, 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 our study considers two criteria for choosing the wanted path: the first cost defines the distance, while the second one represent the danger. So, the problem with Dijkstra is that we will found a path very short, but very dangerous, or vice versa. Concerning multicriteria shortest path problem is intended to determine a path that optimizes the costs from a source to the target, but, in general, there is no a single optimal solution, so the goal of this project is to determinate a set of feasible and not-dominated solution, founding different paths based on the two criteria, so is not sufficient minimizing distance or danger, but the relations between this two values.

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

In table 1.1 are shown all the possible graph's iterations, $p1$ and $p3$ respectively the shortest and the safest paths, $p2$ is a not-dominated solution (it is longer than $p1$ but safer, more dangerous then $p3$, but, in this case, shorter) and $p4$ is a dominated path, so is useless to us.

$p1$	$1 \to 2 \to 4 \to 6$	(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

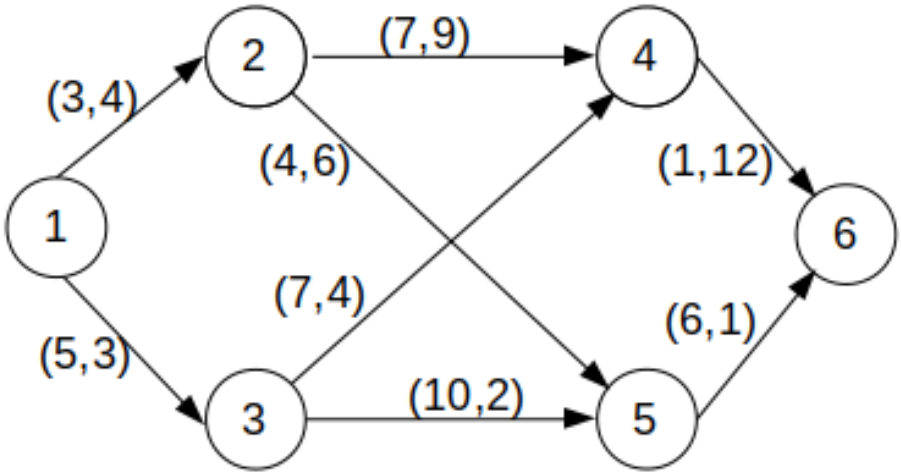

Figure 1.1: Graph example.

1.1 Mathematical formulation

Let $G(N, A)$ denotes direct network which is composed of a finite set $N = \{0, 1, \ldots, n\}$ 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 order 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 we are talking about a double criterion problem) represent the cost which we are referring to. We 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 we want to find one or more paths. We can qualify a path $p_{s,t}$ 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 we said that each $c^k_{i,j}$ refers to one of the two costs of each arch (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 will be explained the algorithms which concern the single criteria routing. The analysis focuses on the evolution and optimization of the following 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 arches with positive weights. Different implementation of this algorithm are present in this paper, as follows are all explained.

```
1  def DijkstraListOfCandidate(source, target):
2      dist[source] ← 0                          // Initialization
3
4      create priorityQueue Q
5
6      while Q is not empty:                     // The main loop
7          u ← Q.extract_min()                   // Remove and return best vertex
8          for each neighbor v of u:             // only v that are still in Q
9              alt ← dist[u] + length(u, v)
10             if alt < dist[v]
11                 dist[v] ← alt
12                 prev[v] ← u
13                 Q.decrease_priority(v, alt)
14
15     return dist, prev
```

Figure 2.1: Pseudocode - Dijkstra (List of Candidates)

2.1.1 One to all

This implementation is useful to find **all the shortest path** 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 implement 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 from this to the previous implementation is that this doesn't use a priority queue, but the algorithm will interrogate each not-visited node every loop; 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 an implementation that uses a priority queue. The elements of this queue are insert by each new visited node, so the queue's elements are the neighbors of the visited node; in this way the algorithm needs to interrogate only some nodes and not all graphs. The list of candidate algorithm has the same *worst-case complexity* of the *One to all* algorithm: $O((|N| + |A|) \log_2(|N|))$.

This implementation is faster than the previous: details of improvement are visualized and studied in the dedicated section.

2.2 "A Star" algorithm

The A Star algorithm (or 'A*') is almost an extension of Dijkstra's algorithm, but it achieves better performance and accuracy by using (generically) heuristics. To determinate which of its paths to extend, A* does so based on the cost of the path and an estimate of the cost required to expand the path to the goal.

So A* select nodes that minimize:

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

- n is the next path's node
- $g(n)$ 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 [1] 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 $f(x)$ 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 visited.

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.

2.3 Analysis of the results

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

2.3.1 Time consuming

In the figure 2.2 are shown some results, from 15 different iteration, classified in three "set" that groups different path's length.

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

		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	0.0006515979766845703	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	0.0002653987854003906
	7642->8365	1767	0.1174166202545166	1767	5.9814839363098145	1767	0.0019183158874511719	1767	0.0006570816040039062
	5456->27648	1887	0.12577557563781738	1887	8.55224061012268	1887	0.0032589435577392578	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	0.00730943679809570 3
AVG		3058.2	0.11551589965820312	3058.2	24.875223064422606	3058.2	0.011389112472534 18	3058.2	0.0026869773864746094
Big >5000	15426->2070	6309	0.10060501098632812	6309	78.5000205039978	6309	0.04720735549926758	6309	0.01497030258178711
	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 2.2: Performance of the mono-criteria algorithms

As possible to get from the table, we can recognize that the first column, '*one→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→one* algorithm is easily to recognize that the implementation without a priority queue is unsustainably slow, especially in the longest paths. From the latest two algorithm is possible to see that the *A Star*'s implementation is more the 3 times faster than the *List of candidate*, than from his part is near to be ten times more performing than the one to all implementation (which uses a priority queue).

2.3.2 Node visited

All the implementation are going to find the same paths between nodes; in particular all the implementation (except for the *A Star*'s algorithm) explore the same nodes, instead, the 'not-Dijkstra's algorithm', explores less number of nodes, this means that it makes less loops during the elaboration and research of the shortest path.

In the following images 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 his nature, is more direct and it reaches the target exploring an "oval" between the starting nodes and the target. So is easy to see and understand how the last implementation is more efficient than the others.

Chapter 3

Bicriteria algorithms

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

Definition 1. Feasible solution
Let x, y be two distinct feasible path from a source s to a target t. We say x *dominates* y if and only of $c^k(x) \leq c^k(y)\ \forall\ k \in [1, 2]$.

Definition 2. Pareto front
For a given set of feasible solutions there is a subset of optimal solutions. It is simpler to understand graphically.

3.1 Dijkstra applied to bicriteria

So, by the fact that we have to deal with two criterion, it's possible to introduce another variable: α.
We also can now set the following variables: the distance $dist = c^1$ and the danger $dang = c^2$.
According with the function:

$$\alpha * dist + (1 - \alpha)dang = k \tag{3.1}$$

where k is constant and $\alpha \in [0, 1]$.

We can say that our graph $G = (N, V, dist, dang)$ is reducible to $G = (N, V, \alpha * dist + (1-\alpha)dang)$ and we can address the problem as the previous case, so using Dijkstra's algorithm (list of candidates) giving a value to α.
It is easy to deduce that plus the value of α approaches 0 means to find safest paths, instead, more α is near to 1 means that we'll found shortest paths.

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

```
1  def BicriteriaDijkstra(source, target, α):
2      weight[source] ← 0                          // Initialization
3
4      create priorityQueue Q
5
6      while Q is not empty:                       // The main loop
7          u ← Q.extract_min()                     // Remove and return best vertex
8          for each neighbor v of u:               // only v that are still in Q
9              alt ← α*(length(u, v)) + (1 - α)*(dang(u, v))
10             alt ← alt + weight[u]
11             if alt < weight[v]
12                 weight[v] ← alt
13                 prev[v] ← u
14                 Q.decrease_priority(v, alt)
15
16     return dist, prev
```

Figure 3.1: Pseudocode - Dijkstra with two criterion

3.1.1 Application of Bicrtieria Dijkstra

The aforementioned algorithm is able to identify the most of the solution To use aforementioned algorithm, have been implemented two different functions.

Bicriteria Dijkstra iteration

This is a very raw version, its operation is based on recalling the pathfinding function, a lot of time, but every time with a different value for α.

The time consuming is related to the chosen precision (the increasing value of α), because it has to call the same function a lot of time and probably with the same result. So more precision means more results, but with a large waste of time.

Bicriteria Dijkstra with binary research

This version is an "evolution" of the last one, because it uses *binary research*. It is based on calling the function at least two time: with $\alpha = 0$ and $\alpha = 1$, then, if the result is 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 other one. The output on figure 3.2 shown that with a precision of (e.g.) 0.2 the *Bicriteria iteration* is pretty faster than the binary research; while, with a

smallest value for precision (little value means more precision), the *Bicriteria iteration* becomes slowest and with less results than the *binary search*.

```
****************** DIJKSTRA BICRITERIA ITERATION ******************
AVGtime: 1.107857894897461
Using a precision of 0.2: From 2000 to 2689 the solutions are:
With α = 0 => distance: 9231 and danger:  12827
With α = 0.2 => distance: 7153 and danger:  12884
With α = 0.7 => distance: 6091 and danger:  14230
With α = 0.55 => distance: 6317 and danger:  13754
With α = 0.875 => distance: 5944 and danger:  14824

****************** DIJKSTRA BICRITERIA ITERATION ******************
AVGtime: 5.258308053016663
Using a precision of 0.05: From 2000 to 2689 the solutions are:
With α = 0 => distance: 9231 and danger:  12827
With α = 0.05 => distance: 7153 and danger:  12884
With α = 0.5249999999999999 => distance: 6317 and danger:  13754
With α = 0.7125000000000001 => distance: 6091 and danger:  14230
With α = 0.8062500000000004 => distance: 6030 and danger:  14430
With α = 0.8531250000000005 => distance: 5944 and danger:  14824
With α = 0.9265625000000006 => distance: 5874 and danger:  15649

***************** DIJKSTRA BICRITERIA BINARY SEARCH *****************
AVGtime: 4.4512746095657345
From 2000 to 2689 the solutions are:
With α = 1 => distance: 5840 and danger:  16763
With α = 0 => distance: 9231 and danger:  12827
With α = 0.5 => distance: 7153 and danger:  12884
With α = 0.75 => distance: 6091 and danger:  14230
With α = 0.625 => distance: 6317 and danger:  13754
With α = 0.875 => distance: 5944 and danger:  14824
With α = 0.812 => distance: 6030 and danger:  14430
With α = 0.937 => distance: 5874 and danger:  15649
With α = 0.9680000000000001 => distance: 5855 and danger:  16199
With α = 0.9660000000000001 => distance: 5863 and danger:  15959
With α = 0.9740000000000001 => distance: 5847 and danger:  16493
```

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

3.2 Label-setting algorithm

We said that the algorithm that use Dijkstra is not able to find all feasible solutions, but only those that make the *Pareto front*; so this algorithm is useful if we want to know all the set of non-dominated solutions. Below a visualization on what we are talking about.

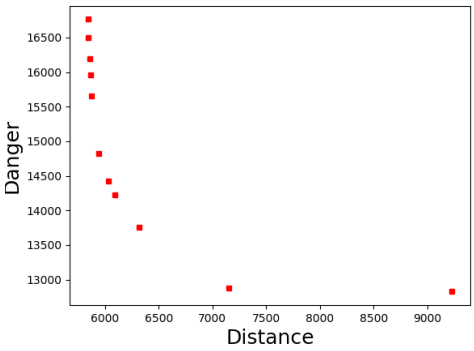

Figure 3.3: Graph generated by *Bicriteria Dijkstra with binary search*

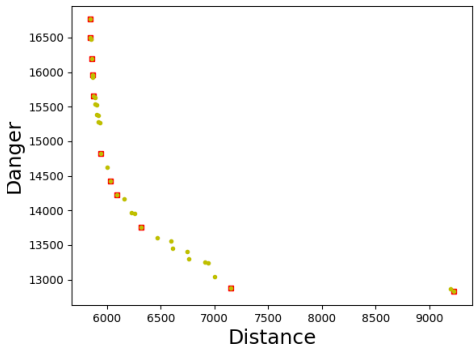

Figure 3.4: Graph generated by *Label-setting algorithm*

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 *Pareto front* (red squares), 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 (Pareto front). The solutions found by label-setting algorithm are all feasible, so it is more complete, in fact, as it is possible to notice, there are no points with both values danger and distance greater than the others.

3.2.1 Implementation

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

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

As it is deductible the first two parameter are the costs of the path from the source, the third and four elements indicate the node that own 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 his label set (used for backtracking). As always 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 (vi) otherwise get the label with smallest distance's value from the queue and calculate the label for all his neighbors.

iii. Check if the calculated label is or less a non-dominated label. There's can be three different solution: the calculated label is dominated, so the algorithm will discards it; the calculated label dominate other labels, so we can delete those label; 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 it is put in the queue, using the distance as a criteria for his priority.

v. Return to step (ii).

vi. End.

Example:

We can 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 his set. Then creating labels for his 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 we reach the target node. At any new label we have to study if it is feasible or less before put 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 to improve the speed, especially when it has to find paths between nodes far from each other. 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 of the visited node from the source, when we calculate the shortest path ($\alpha = 1$), and, in the same way, the danger counted from the beginning ($\alpha = 0$).

How it can be useful?

We know that the research of the shortest and the safest path will generate the better value for distance and danger in any case and for each visited

node. So, once calculated the value of distance of the shortest path and the value of danger for the safest, we also know, for a node elaborated in preprocessing phase, this information: the total value of the full path's distance (and danger), node's better value of distance (and danger), actual value of distance (and danger) calculated during the running of label-setting 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 less.

3.2.3 Complexity

Chapter 4

Analyse the results

Chapter 5

Implementation choices

5.0.1 Nodes as objects

5.0.2 Pandas & MatPlotLib

5.0.3 Testing

Chapter 6

Personal considerations

6.0.1 Genetic algorithm

6.0.2 Difficulties encountered

Bibliography

[1] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998.