

# Supervised Project “Bicriteria Paths Problem”

Andrea Rossolini

May 29, 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

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Mathematical formulation . . . . .	3
<b>2</b>	<b>Mono-criteria algorithms</b>	<b>5</b>
2.1	Dijkstra's algorithms . . . . .	5
2.1.1	One to all . . . . .	6
2.1.2	One to one . . . . .	6
2.1.3	List of candidate . . . . .	6
2.2	"A Star" algorithm . . . . .	6
2.3	Analysis of the results . . . . .	7
2.3.1	Time consuming . . . . .	8
2.3.2	Node visited . . . . .	8
<b>3</b>	<b>Bicriteria algorithms</b>	<b>10</b>
3.1	Dijkstra applied to bicriteria . . . . .	10
3.1.1	Application of Bicriteria Dijkstra . . . . .	11
3.2	Metodologia di lavoro . . . . .	13
3.3	Note di sviluppo . . . . .	14
<b>4</b>	<b>Commenti finali</b>	<b>17</b>
4.1	Autovalutazione e lavori futuri . . . . .	17
4.2	Difficoltà incontrate e commenti per i docenti . . . . .	17
<b>A</b>	<b>Guida utente</b>	<b>19</b>

# 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  $\{1, 2, \dots, 6\}$  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

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

Table 1.1: Graph's solution

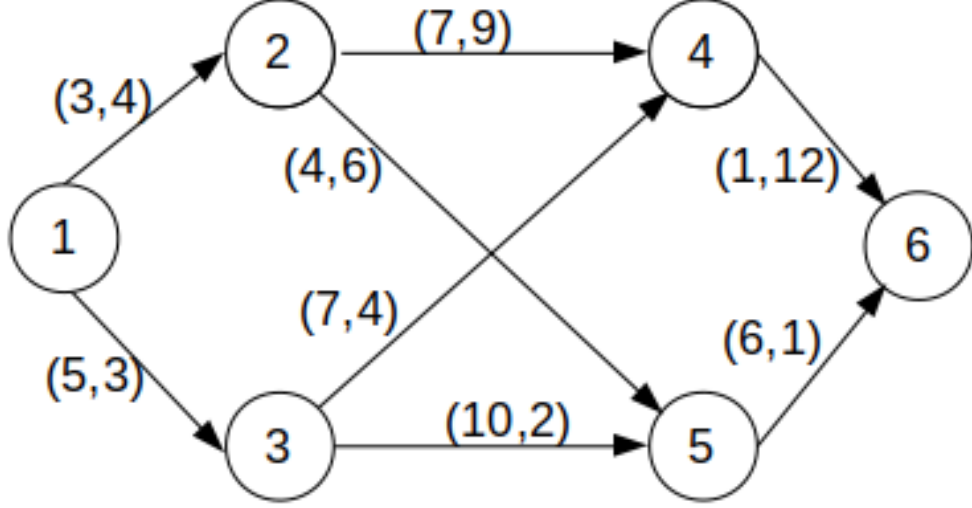


Figure 1.1: Graph example.

(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.

## 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 order pair  $(i, j)$ , where  $i \in N$ ,  $j \in N$  and both are two different nodes in  $G(N, A)$ .

Let define  $c_{i,j}^k$  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  $p_{s,t} = \{s, (s, i_1), i_1, \dots, i_l, (i_l, t), t\}$ .

So we said that each  $c_{i,j}^k$  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:

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

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

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

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             alt ← dist[u] + length(u, v)
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 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  $A \gg N$ , 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 inserted 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 <sup>1</sup> 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.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/A\\*\\_search\\_algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm)

### 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.

		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	0.00026535987854003906
	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	0.00375699969482422	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	0.0080306529987793	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	0.01497030258178711
	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:  $\alpha$ .

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  $\alpha$ .

It is easy to deduce that plus the value of  $\alpha$  approaches 0 means to find safest paths, instead, more  $\alpha$  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  $\alpha$ ; then, using (3.1), the algorithm will choose the optimal path.

```

1  def BicriteriaDijkstra(source, target,  $\alpha$ ):
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 ←  $\alpha$ *(length(u, v)) + (1 -  $\alpha$ )*(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 Bicriteria 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  $\alpha$ .

It is very time consuming, because it has to call the same function a lot of time and probably with the same result, and its precision is related to the increasing value for  $\alpha$ .

#### 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 is more efficient and precise than the other one.

## Esempi di UML ben realizzati

In questa sezione ci si concentrerà sugli aspetti di personalità e sul funzionamento del reporting di GLaDOS.

Il sistema per la gestione della personalità utilizza il pattern Strategy, come da ???: le implementazioni di Personality possono essere modificate, e la modifica impatta direttamente sul comportamento di GLaDOS.

Sono state attualmente implementate due personalità, una buona ed una cattiva. Quella buona restituisce sempre una torta valida, mentre quella cattiva restituisce sempre una torta. Dato che le due personalità differiscono solo per il comportamento da effettuarsi in caso di percorso completato con successo, è stato utilizzato il pattern template method per massimizzare il riuso, come da ??. Il metodo template è `onSuccess()`, che chiama un metodo astratto e protetto `makeCake()`.

## Elementi positivi

- Si descrivono molto brevemente i componenti che si è deciso di sottoporre a test automatizzato.
- Si utilizzano suite specifiche (e.g. JUnit) per il testing automatico.
- Se sono stati eseguiti test manuali di rilievo, si elencano descrivendo brevemente la ragione per cui non sono stati automatizzati. Ad esempio, se tutto il team sviluppa e testa su uno stesso sistema operativo e si sono svolti test manuali per verificare, ad esempio, il corretto funzionamento dell'interfaccia grafica o di librerie native su altri sistemi operativi, può avere senso menzionare la cosa.

## Elementi negativi

- Non si realizza alcun test automatico.
- La non presenza di testing viene aggravata dall'adduzione di motivazioni non valide.
- Si descrive un testing di tipo manuale in maniera prolissa.

- Si descrivono test effettuati manualmente che sarebbero potuti essere automatizzati, ad esempio scrivendo che si è usata l'applicazione manualmente.

## 3.2 Metodologia di lavoro

Ci aspettiamo, leggendo questa sezione, di trovare conferma alla divisione operata nella sezione del design di dettaglio, e di capire come è stato svolto il lavoro di integrazione.

### Elementi positivi

- Si identifica con precisione il ruolo di ciascuno all'interno del gruppo, ossia su quale parte del progetto ciascuno dei componenti si è concentrato maggiormente.
- La divisione dei compiti è equa, ossia non vi sono membri del gruppo che hanno svolto molto più lavoro di altri
- La divisione dei compiti è coerente con quanto descritto nelle parti precedenti della relazione
- La divisione dei compiti è realistica, ossia le dipendenze fra le parti sviluppate sono minime
- Si identifica quale parte del software è stato sviluppato da tutti i componenti insieme.
- Si spiega in che modo si sono integrate le parti di codice sviluppate separatamente, evidenziando eventuali problemi. Ad esempio, una strategia è convenire sulle interfacce da usare (ossia, occuparsi insieme di stabilire l'architettura) e quindi procedere indipendentemente allo sviluppo di parti differenti. Una possibile problematica potrebbe essere una dimenticanza in fase di design architetturale che ha costretto ad un cambio e a modifiche in fase di integrazione. Una situazione simile è la norma nell'ingegneria di un sistema software non banale, ed il processo di progettazione top-down con raffinamento successivo è il così detto processo "a spirale".
- Si descrive in che modo è stato impiegato il DVCS.

## Elementi negativi

- Non si chiarisce chi ha fatto cosa.
- C'è discrepanza fra questa sezione e le sezioni che descrivono il design dettagliato.
- Tutto il progetto è stato svolto lavorando insieme invece che assegnando una parte a ciascuno.
- Non viene descritta la metodologia di integrazione delle parti sviluppate indipendentemente.
- Uso superficiale del DVCS.

## 3.3 Note di sviluppo

Questa sezione, come quella riguardante il design dettagliato va svolta **sin-  
golarmente da ogni membro del gruppo**.

Ciascuno dovrà mettere in evidenza eventuali particolarità del suo metodo di sviluppo, ed in particolare:

- Elencare le feature avanzate del linguaggio Java che sono state utilizzate. Le feature di interesse possono essere, ad esempio:
  - Uso avanzato dei generici (ad esempio costruzione di nuovi tipi generici, e uso di generici bounded)
  - Uso delle lambda expressions
  - Uso degli stream, degli optional o di altri costrutti monadici
  - Uso della reflection
  - Uso di parti di libreria non spiegate a lezione (networking, JavaScript via Nashorn, eccetera...)

Si faccia molta attenzione a non scrivere banalità, elencando qui features di tipo “core”, come le eccezioni, o le inner class.

- Descrivere eventuali approfondimenti fatti rispetto a quanto trattato nel corso (ad esempio l'utilizzo di un logger, o l'accesso alla rete, o l'uso di librerie grafiche particolari)
- Descrivere le librerie utilizzate nella propria parte di progetto. Si ricorda che l'utilizzo di librerie è valutato *positivamente*.



- Sviluppo di algoritmi particolarmente interessanti *non forniti da alcuna libreria*

In questa sezione è anche bene evidenziare eventuali pezzi di codice scopi-  
azzati da Internet o da altri progetti (pratica che tolleriamo ma che non  
raccomandiamo).

## Elementi positivi

- Si elencano gli aspetti avanzati di linguaggio che sono stati impiegati
- Si elencano le librerie che sono state utilizzate
- Si descrivono aspetti particolarmente complicati o rilevanti relativi all'implementazione, ad esempio, in un'applicazione performance critical, un uso particolarmente avanzato di meccanismi di caching, oppure l'implementazione di uno specifico algoritmo.
- Se si è utilizzato un particolare algoritmo, se ne cita la fonte originale. Ad esempio, se si è usato Mersenne Twister per la generazione dei numeri random, si cita [1].
- Si identificano parti di codice prese da altri progetti, dal web, o comunque scritte in forma originale da altre persone. In tal senso, si ricorda che agli ingegneri non è richiesto di re-inventare la ruota continuamente: se ci cita debitamente la sorgente è tollerato fare uso di di snippet di codice per risolvere velocemente problemi non banali. Nel caso in cui si usino snippet di codice di qualità discutibile, oltre a menzionarne l'autore originale si invitano gli studenti ad adeguare tali parti di codice agli standard e allo stile del progetto. Contestualmente, si fa presente che è largamente meglio fare uso di una libreria che copiarsi pezzi di codice: qualora vi sia scelta, si preferisca la prima via.

## Elementi negativi

- Si elencano feature core del linguaggio invece di quelle segnalate.
- Si descrivono aspetti di scarsa rilevanza, o si scende in dettagli inutili.
- Sono presenti parti di codice sviluppate originalmente da altri che non vengono debitamente segnalate. In tal senso, si ricorda agli studenti che i docenti hanno accesso a tutti i progetti degli anni passati, a Stack Overflow, ai principali blog di sviluppatori ed esperti Java e ai blog

dedicati allo sviluppo di soluzioni e applicazioni (inclusi blog dedicati ad Android e allo sviluppo di videogame). Conseguentemente, è *molto* conveniente *citare* una fonte ed usarla invece di tentare di spacciare per proprio il lavoro di altri.

# Chapter 4

## Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

### 4.1 Autovalutazione e lavori futuri

#### Cosa scrivere

**È richiesta una sezione per ciascun membro del gruppo.** Ciascuno dovrà autovalutare il proprio lavoro, elencando i punti di forza e di debolezza in quanto prodotto. Si dovrà anche cercare di descrivere *in modo quanto più obiettivo* il proprio ruolo all'interno del gruppo. Si ricorda, a tal proposito, che ciascuno studente è responsabile solo della propria sezione: non è un problema se ci sono opinioni contrastanti, a patto che rispecchino effettivamente l'opinione di chi le scrive. Nel caso in cui si pensasse di portare avanti il progetto, ad esempio perché effettivamente impiegato, o perché sufficientemente ben riuscito da poter esser usato come dimostrazione di esser capaci progettisti, si descriva brevemente verso che direzione portarlo.

### 4.2 Difficoltà incontrate e commenti per i docenti

Questa sezione, opzionale, può essere utilizzata per segnalare ai docenti eventuali problemi o difficoltà incontrate nel corso o nello svolgimento del progetto, può essere vista come una seconda possibilità di valutare il corso (dopo quella offerta dalle rilevazioni della didattica) avendo anche conoscenza delle modalità e delle difficoltà collegate all'esame, cosa impossibile da fare us-

ando le valutazioni in aula per ovvie ragioni di tempistiche. È possibile che alcuni dei commenti forniti vengano utilizzati per migliorare il corso in futuro: sebbene non andrà a vostro beneficio, potreste fare un favore ai vostri futuri colleghi. Ovviamente il contenuto della sezione non impatterà il voto finale.

# Appendix A

## Guida utente

Capitolo in cui si spiega come utilizzare il software. Nel caso in cui il suo uso sia del tutto banale, tale capitolo può essere omesso.

### **Elementi positivi**

- Si istruisce in modo semplice l'utente sull'uso dell'applicazione, eventualmente facendo uso di schermate e descrizioni.

### **Elementi negativi**

- Si descrivono in modo eccessivamente minuzioso tutte le caratteristiche, anche minori, del software in oggetto.
- Manca una descrizione che consenta ad un utente qualunque di utilizzare almeno le funzionalità primarie dell'applicativo.

# 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.