Supervised Project "Bicriteria Paths Problem"

Andrea Rossolini

May 21, 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 the 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

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

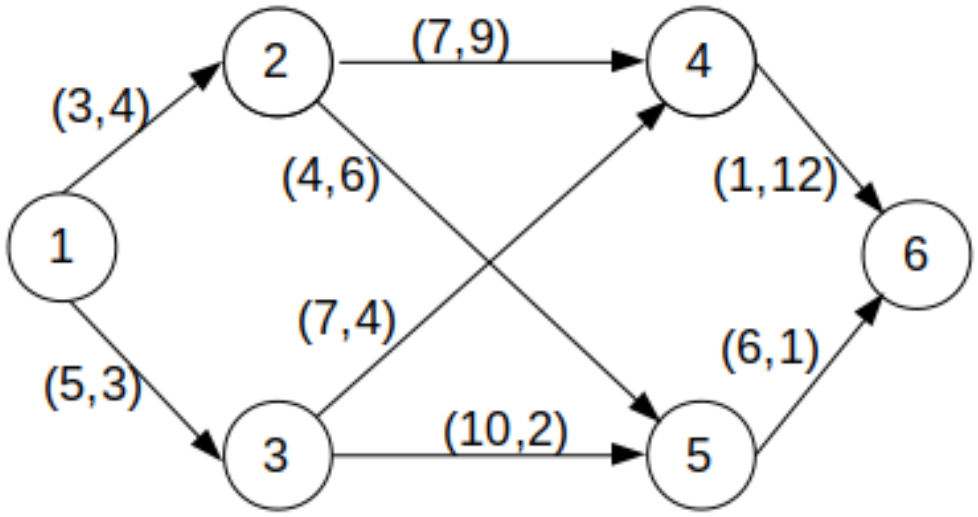

Figure 1.1: Graph example.

(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 $N = \{0, 1, \ldots, n\}$ 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.

Definition 1. Definition 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 $c^k(x) \leq c^k(y)\ \forall\ k \in [1, 2]$

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

========== Questa sezione spiega come le componenti principali del softwareh interagiscono fra loro. In particolare, qui va spiegato se e come è stato utilizzato il pattern architetturale model-view-controller (e/o entity-control-boundary). Se non è stato utilizzato, va spiegata in maniera molto accurata l'architettura scelta, giustificandola in modo appropriato.

Raccomandiamo di sfruttare la definizione del dominio fatta in fase di analisi per capire quale sia l'entry point del model, e di non realizzare un'unica macro-interfaccia che, spesso, finisce con l'essere il prodromo ad una "God class". Consigliamo anche di separare bene controller e model, facendo attenzione a non includere nel secondo strategie d'uso che appartengono al primo. Infine, attenzione al design dell'interazione fra view e controller: se ben progettato, sostituire in blocco la view non dovrebbe causare alcuna modifica nel controller.

2.1.1 Elementi positivi

- Si mostrano pochi, mirati schemi UML dai quali si deduce con chiarezza quali sono le parti principali del software e come interagiscono fra loro.

- Si mette in evidenza se e come il pattern architetturale model-view-controller è stato applicato, anche con l'uso di un UML che mostri le interfacce principali ed i rapporti fra loro.
- Si discute se sia semplice o meno, con l'architettura scelta, sostituire in blocco la view senza toccare minimamente il controller. Va da sé che, se cambiare la UI si trasforma in un bagno di sangue che impatta controller o modello, il design architetturale non è stato fatto in modo adeguato.

Elementi negativi

- L'architettura è fatta in modo che sia impossibile riusare il modello per un software diverso che affronta lo stesso problema.
- L'architettura è tale che l'aggiunta di una funzionalità sul controller impatta pesantemente su view e/o modello.
- L'architettura è tale che la sostituzione in blocco della view impatta sul controller o, peggio ancora, sul modello.
- Si presentano UML caotici, difficili da leggere.
- Si presentano UML in cui sono mostrati elementi di dettaglio non appartenenti all'architettura, ad esempio includenti campi o con metodi che non interessano la parte di interazione fra le componenti principali del software.
- Si presentano schemi UML con classi (nel senso UML del termine) che "galleggiano" nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
- Si presentano elementi di design di dettaglio, ad esempio tutte le classi e interfacce del modello o della view.
- Si discutono aspetti implementativi, ad esempio eventuali librerie usate oppure dettagli di codice.

Esempio

L'architettura di GLaDOS segue il pattern architetturale ECB. GLaDOS implementa l'interfaccia AI, ed è l'effettivo controller del sistema. Essendo una intelligenza artificiale, è una classe attiva. GLaDOS accetta la registrazione

Figure 2.1: Schema UML architetturale di GLaDOS

di Input ed Output. Gli Input rappresentano delle nuove informazioni che vengono fornite all'IA, ad esempio delle modifiche nel valore di un sensore, oppure un comando da parte dell'operatore. Questi input infatti forniscono eventi. Ottenere un evento è un'operazione bloccante: chi la esegue resta in attesa di un effettivo evento. Ogni volta che c'è un cambio alla situazione del soggetto, GLaDOS notifica i suoi Output, informandoli su quale sia la situazione corrente.

Con questa architettura, possono essere aggiunti un numero arbitrario di input ed output all'intelligenza artificiale.

In Figure 2.1 è esemplificato il diagramma UML architetturale.

2.2 Design dettagliato

In questa sezione si possono approfondire alcuni elementi del design con maggior dettaglio. Mentre ci attendiamo principalmente (o solo) interfacce negli schemi UML delle sezioni precedenti, in questa sezione è necessario scendere in maggior dettaglio presentando la struttura di alcune sottoparti rilevanti dell'applicazione. È molto importante che, descrivendo un problema, quando possibile si mostri che non si è re-inventata la ruota ma si è applicato un design pattern noto. È assolutamente inutile, ed è anzi controproducente, descrivere classe-per-classe (o peggio ancora metodo-per-metodo) com'è fatto il vostro software: è un livello di dettaglio proprio della documentazione dell'API (deducibile dalla Javadoc).

È necessario che ciascun membro del gruppo abbia una propria sezione di design dettagliato, di cui sarà responsabile. Ciascun autore dovrà spiegare in modo corretto e giustamente approfondito (non troppo in dettaglio, non superficialmente) il proprio contributo. È importante focalizzarsi sulle scelte che hanno un impatto positivo sul riuso, sull'estensibilità, e sulla chiarezza dell'applicazione. Usare correttamente i design pattern in questa sezione è molto importante: se vengono utilizzati, si ha la garanzia che quanto fatto sia conforme allo stato dell'arte.

Esattamente come nessun ingegnere meccanico presenta un solo foglio con l'intero progetto di una vettura di Formula 1, ma molteplici fogli di progetto che mostrano a livelli di dettaglio differenti le varie parti della vettura e le modalità di connessione fra le parti, così ci aspettiamo che voi, futuri ingegneri informatici, ci presentiate prima una visione globale del progetto, e via via siate in grado di dettagliare le singole parti, scartando i componenti

che non interessano quella in esame. Per continuare il parallelo con la vettura di Formula 1, se nei fogli di progetto che mostrano il design delle sospensioni anteriori appaiono pezzi che appartengono al volante o al turbo, vuol dire che c'è qualche grosso problema di design.

Elementi positivi

- Ogni membro del gruppo discute le proprie decisioni di progettazione, ed in particolare le azioni volte ad anticipare possibili cambiamenti futuri (ad esempio l'aggiunta di una nuova funzionalità, o il miglioramento di una esistente).
- Si identificano ed utilizzano numerosi design pattern.
- Ogni membro del gruppo identifica i pattern utilizzati nella sua sottoparte.
- Si mostrano gli aspetti di design più rilevanti dell'applicazione, mettendo in luce la maniera in cui si è costruita la soluzione ai problemi descritti nell'analisi.
- Si tralasciano aspetti strettamente implementativi e quelli non rilevanti, non mostrandoli negli schemi UML (ad esempio, campi privati) e non descrivendoli.
- Si mostrano le principali interazioni fra le varie componenti che collaborano alla soluzione di un determinato problema.
- Ciascun design pattern identificato presenta una piccola descrizione del problema calato nell'applicazione, uno schema UML che ne mostri la concretizzazione nelle classi del progetto, ed una breve descrizione della motivazione per cui tale pattern è stato scelto.
- La divisione in package rispecchia l'architettura e consente di navigare facilmente il sorgente, dando brevi giustificazione alle scelte fatte.

Elementi negativi

- Il design del modello risulta scorrelato dal problema descritto in analisi.
- Si tratta in modo prolisso, classe per classe, il software realizzato.
- Non si presentano schemi UML esemplificativi.

Figure 2.2: Rappresentazione UML del pattern Strategy per la personalità di GLaDOS

Figure 2.3: Rappresentazione UML dell'applicazione del pattern Template Method alla gerarchia delle Personalità

- Non si individuano design pattern, o si individuano in modo errato (si spaccia per design pattern qualcosa che non lo è).
- Si producono schemi UML caotici e difficili da leggere, che comprendono inutili elementi di dettaglio.
- Si presentano schemi UML con classi (nel senso UML del termine) che "galleggiano" nello schema, non connesse, ossia senza relazioni con il resto degli elementi inseriti.
- Si tratta in modo inutilmente prolisso la divisione in package, elencando ad esempio le classi una per una.
- La divisione in package non rispecchia l'architettura, o è caotica.

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 Figure 2.2: 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 Figure 2.3. Il metodo template è onSuccess(), che chiama un metodo astratto e protetto makeCake().

Per quanto riguarda il reporting, è stato utilizzato il pattern Observer per consentire la comunicazione uno-a-molti fra GLaDOS ed i sistemi di output. Il suo utilizzo è esemplificato in Figure 2.4

Figure 2.4: Il pattern Observer è usato per consentire a GLaDOS di informare tutti i sistemi di output in ascolto

Figure 2.5: Schema UML mal fatto e con una pessima descrizione, che non aiuta a capire. Don't try this at home.

Esempio di pessimo diagramma UML

In Figure 2.5 è mostrato il modo **sbagliato** di fare le cose. Questo schema è fatto male perché:

- È caotico.
- È difficile da leggere e capire.
- Vi sono troppe classi, e non si capisce bene quali siano i rapporti che intercorrono fra loro.
- Si mostrano elementi implementativi irrilevanti, come i campi e i metodi privati nella classe `AbstractEnvironment`.
- Se l'intenzione era quella di costruire un diagramma architetturale, allora lo schema è ancora più sbagliato, perché mostra pezzi di implementazione.
- Una delle classi, in alto al centro, galleggia nello schema, non connessa a nessuna altra classe, e di fatto costituisce da sola un secondo schema UML scorrelato al resto
- Le interfacce presentano tutti i metodi e non una selezione che aiuti il lettore a capire quale parte del sistema si vuol mostrare.

Chapter 3

Sviluppo

3.1 Testing automatizzato

Il testing automatizzato è un requisito di qualunque progetto software che si rispetti, e consente di verificare che non vi siano regressioni nelle funzionalità a fronte di aggiornamenti. Per quanto riguarda questo progetto è considerato sufficiente un test minimale, a patto che sia completamente automatico. Test che richiedono l'intervento da parte dell'utente sono considerati *negativamente* nel computo del punteggio finale.

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 **singolarmente 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 scopiazzati 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 algorigmo.
- 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.