

ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA E SCIENZE
INFORMATICHE

Analysis and Implementation of Algorithms for Bicriteria Shortest Paths Problems

Elaborato in
RICERCA OPERATIVA

Relatore:
Cha.mo Prof.
VIGO DANIELE

Presentata da:
ROSSOLINI ANDREA

Correlatore:
NÉRON EMMANUEL

Anno Accademico: 2018/2019
III sessione di laurea

English version available on: [https://bitbucket.org/AndreaRossolini/
bicriteria-algorithm/src/master/notes/](https://bitbucket.org/AndreaRossolini/bicriteria-algorithm/src/master/notes/)

Sommario

Questa tesi si pone l'obbiettivo di affrontare ed analizzare un problema di *pathfinding* partendo da un'analisi di alcuni algoritmi per la ricerca del percorso più breve su normali grafi, per poi ampliare lo studio e concentrarsi su algoritmi che calcolano molteplici percorsi su grafi che utilizzano due pesi per ogni arco. Gli algoritmi per i grafi ‘bicriteria’ (appunto che considerano due pesi su ogni arco) verranno analizzati, implementati e le loro soluzioni confrontate con gli altri algoritmi, al fine di individuare i più efficienti in termini di tempo di elaborazione e quelli che riescono a minimizzare al meglio i pesi degli archi dei cammini trovati, quindi valutando quantità e qualità delle soluzioni. Dato che, lo studio di algoritmi per la ricerca del percorso più breve in grafi classici è piuttosto celebre in letteratura, è ormai facile trovare implementazioni che permettano di risolvere questo problema. Verrà effettuata quindi una rapida implementazione ed analisi riguardante gli algoritmi ‘monocriteria’. Per agli algoritmi che lavorano in grafi con più pesi, per i quali è più difficile reperire implementazioni ed analisi, ci sarà una spiegazione più approfondita ed accurata, soffermandosi anche su casi particolari e indicando le scelte implementative fatte per ottimizzare al meglio le loro prestazioni. L’analisi dei risultati confronterà, come già detto, l’insieme delle soluzioni calcolate dai vari algoritmi ed i loro tempi di elaborazione. Inoltre i suddetti algoritmi verranno utilizzati su mappe di alcune città, prese come esempio, per poter fare un confronto visivo sui cammini minimi trovati ed i nodi visitati durante l’elaborazione degli algoritmi; in modo da semplificare e rendere più immediato il confronto tra le varie implementazioni.

*Dedico questo mio modesto lavoro a tutti coloro
Che mi sono stati vicini e mi hanno sostenuto
In questa piccola parte di un ancora lungo cammino
In particolare alla mia famiglia che non ha mai smesso di
credere in me...*

Indice

1	Introduzione	1
1.1	Formulazione Matematica	3
2	Algoritmi ‘Mono-criteria’	5
2.1	Algoritmo di Dijkstra	5
2.1.1	One to All	5
2.1.2	One to One	7
2.1.3	List of Candidate	7
2.2	“A Star”	7
2.3	Analisi dei Risultati	10
2.3.1	Performance a Confronto	10
2.3.2	Nodi Visitati	12
3	Algoritmi ‘Bi-criteria’	17
3.1	Dijkstra applicato ad un problema Bi-criteria	18
3.1.1	Implementazioni dell’algoritmo di Dijkstra con due criteri	19
3.2	Algoritmo di Label-setting	22
3.2.1	Implementazione	23
3.3	Algoritmo di Label-setting con minoranti pregressi	26
3.3.1	Algoritmo di label-setting con minoranti pregressi invertito	26
3.4	Algoritmo Bidirezionale Bi-criteria	27
3.4.1	Implementazione	29
3.5	Analisi dei risultati	32

3.5.1	Performance a confronto	32
3.5.2	Nodi visitati	35
4	Considerazioni Personal	41
4.1	Idee Scartate	41
4.2	Possibili Applicazioni	42
4.3	Difficoltà incontrate	42
4.4	Sviluppi futuri	43
5	Conclusioni	45
6	Dettagli implementativi	47
6.1	Informazioni sui grafi	47
6.2	Informazioni sulla macchina utilizzata	48

Elenco delle figure

1.1	Grafo d'esempio.	2
2.1	Pseudocodice dell'Algoritmo di Dijkstra (list of candidate) . . .	6
2.2	Pseudocodice dell'Algoritmo A*	9
3.1	Pseudocodice dell'Algoritmo di Dijkstra con due criteri	19
3.2	Soluzioni trovate da Dijkstra bi-criteria con ricerca binaria . .	23
3.3	Soluzioni trovate dall'algoritmo Label-setting	23
3.4	Pseudocodice dell'Algoritmo Label-setting	24
3.5	Caso peggiore dell'algoritmo di Label-setting	25
3.6	Pseudocodice dell'Algoritmo bidirezionale	31
3.7	Algoritmo Label-setting su Parigi (mappa orientata).	35
3.8	Algoritmo Label-setting su Berlino (mappa orientata).	36
3.9	Algoritmo Label-setting su Bologna (mappa non orientata) . . .	37
3.10	Algoritmo bidirezionale su Bologna (mappa non orientata). . .	38
3.11	Esempio del percorso su Bologna preso dal sito <i>Google Maps</i> . .	39
3.12	Algoritmo Label-setting (sopra) e Bidirezionale (sotto) su Cesena (mappa non orientata).	40

Elenco delle tavelle

1.1	Soluzione del grafo di esempio	2
2.1	Performance degli algoritmi mono-criteria (tabella generata per comodità con la libreria tkinter)	11
3.1	Output dei due algoritmi di Dijkstra bi-criteria iterativi con differenti precisioni	21
3.2	Output dei dell'algoritmo di Dijkstra bi-criteria con ricerca binaria	22
3.3	Risultati degli algoritmi applicabili a grafi orientati (I fogli excel sono stati generati per comodità tramite la libreria <i>xlsx-writer</i>).	33
3.4	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)	34

Capitolo 1

Introduzione

L’algoritmo di Dijkstra è largamente utilizzato in ambiti dove i problemi di routing riguardano grafi dove sono presenti valori statici non negativi. Questo celebre algoritmo però prende in considerazione una sola “dimensione” dei costi; se, ad esempio, ci fosse la necessità di calcolare il percorso in un grafo da un nodo *sorgente* ad uno *destinazione* questo troverebbe il percorso addizionando volta per volta il peso relativo all’arco che connette due nodi che vengono ”visitati” dall’algoritmo. Ma il problema si ha appunto nel momento in cui si hanno due o più criteri che relazionano due nodi; ad esempio la distanza e il pericolo. In questo caso si potrebbe scegliere tra diversi percorsi (seppur rappresentati all’interno di un grafo), ma non per forza il più corto è anche il più sicuro o vice versa. La ricerca di un percorso breve in termini di differenti criteri (molto più semplicemente in inglese ‘*Muticriteria Shortest Paths Problem*’) ha lo scopo di ottimizzare i costi dal nodo sorgente a quello di destinazione.

In generale non c’è una sola soluzione ottimale, bensì un insieme di possibili soluzioni, anche dette soluzioni “non dominate” (verrà in seguito approfondito il significato di questa espressione). L’obiettivo di questo progetto è dunque quello di implementare l’algoritmo migliore che possa trovare questo insieme di soluzioni non solo minimizzando il primo o il secondo peso, ma la loro relazione.

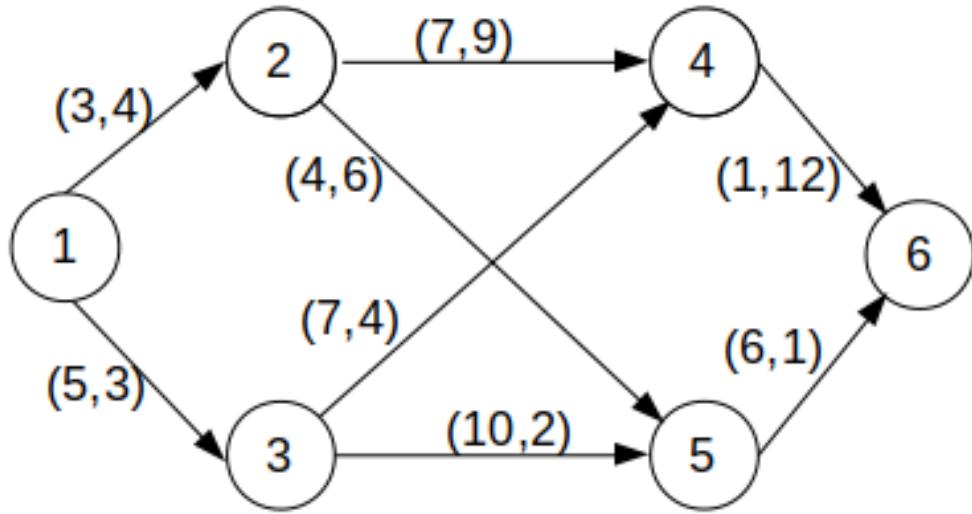


Figura 1.1: Grafo d'esempio.

p_1	$1 \rightarrow 2 \rightarrow 4 \rightarrow 6$	(11, 25)
p_2	$1 \rightarrow 2 \rightarrow 5 \rightarrow 6$	(13, 11)
p_3	$1 \rightarrow 3 \rightarrow 5 \rightarrow 6$	(21, 6)
p_4	$1 \rightarrow 3 \rightarrow 4 \rightarrow 6$	(13, 18)

Tabella 1.1: Soluzione del grafo di esempio

La fig.1.1 mostra un esempio di grafo a due criteri. Ogni nodo è numerato $\{1, 2, \dots, 6\}$ ed i pesi presenti sugli archi sono rappresentati come una coppia di valori (a, b) . Se considerassimo il primo peso la distanza e il secondo peso, ad esempio, il pericolo, avremo i percorsi p_1 e p_3 come il più corto ed il più sicuro rispettivamente, differentemente p_2 è una soluzione definita “non dominata”, infatti comprende un percorso più lungo rispetto a p_1 , ma al contempo più sicuro, mentre percorre un cammino più pericoloso, ma più corto rispetto a “ p_3 ”. p_4 infine segue un percorso che non risulta vantaggioso ne per il primo valore ne per il secondo, in quanto non porta nessun vantaggio

rispetto a p_2 (la definizione formale di “soluzione dominata” sarà formulata nel cap.3).

1.1 Formulazione Matematica

Sia definita $G(N, A)$ un grafo orientato composto da un insieme finito di nodi (o vertici) $N = \{0, 1, \dots, n\}$ e $A \subseteq N \times N$ di archi direzionati. Ogni arco può essere rappresentato come una coppia ordinata (i, j) , dove $i \in N$, $j \in N$ ed entrambi due nodi differenti ($i \neq j$) in $G(N, A)$.

Sia dunque definito $c_{i,j}^k$ dove $(i, j) \in A$ e $1 \leq k \leq 2$ (dato che si tratta di un grafo con due pesi ad ogni arco) che rappresentano i costi relativi ad ogni arco. Considerando quindi due nodi del grafo, chiamati s e t , dove $s, t \in N$, i quali rappresentano rispettivamente il nodo da cui iniziano i percorsi che cerchiamo (*source*) ed il nodo terminale (*target*). Un cammino $p_{s,t}$ può essere rappresentato come una sequenza di nodi e archi, avente la seguente forma: $p_{s,t} = \{s, (s, i_1), i_1, \dots, i_l, (i_l, t), t\}$.

È dunque possibile affermare che ogni $c_{i,j}^k$ rappresenta il costo di uno dei k pesi di ogni arco (i, j) , quindi il costo totale dell'intero cammino è rappresentato nel modo seguente:

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

Dove P è una soluzione

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

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

Il nostro scopo è quello di **minimizzare** la (1.2) o la (1.3) e dunque ottenere lo “spettro” di soluzioni che è presente tra le due. Ad esempio considerando

un grafo dove il primo peso rappresenta la lunghezza dell'arco ed il secondo il pericolo, minimizzando la (1.2) otterremmo il percorso più breve, altrimenti il più sicuro.

Capitolo 2

Algoritmi ‘Mono-criterio’

In questo capitolo sono trattati esclusivamente i gli algoritmi che riguardano la ricerca di cammini minimi con un solo criterio di peso su ogni arco. L’analisi si concentra sull’implementazione e l’ottimizzazione dell’algoritmo di Dijkstra, spiegando alcune scelte implementative, e su di un altro celebre algoritmo in questo campo, l’*A-star*

2.1 Algoritmo di Dijkstra

L’algoritmo di Dijkstra è molto celebre in letteratura e tra chiunque abbia una conoscenza anche basica di algoritmi; è infatti molto usato per la ricerca dei cammini minimi in grafi connessi con archi aventi pesi positivi. Di seguito sono presenti brevi analisi di alcune interessanti implementazioni.

2.1.1 One to All

Questa implementazione è utile in caso si volessero trovare tutti i percorsi brevi da un nodo sorgente $s \in N$ ad ogni altro nodo del grafo, dunque non si ferma finché ogni nodo del grafo (raggiungibile da s) non viene visitato. A livello implementativo il nodo sorgente è realizzato in modo da contenere un dizionario contenente ogni nodo raggiunto, utilizzando come chiave l’indice

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

Figura 2.1: Pseudocodice dell’Algoritmo di Dijkstra (list of candidate)

del nodo e come valore la distanza (ovvero la somma dei pesi degli archi) minima che ha da s .

Implementando una coda di priorità si raggiunge una complessità di $O((|N| + |A|) \log_2(|N|))$ dove N è il numero dei vertici del grafo, mentre A è il numero degli archi. Il caso peggiore, dove $A \gg N$, raggiunge una complessità di $O(|A| \log_2(|N|))$.

2.1.2 One to One

A differenza della precedente, questa implementazione si concentra sul classico algoritmo di Dijkstra; non va quindi a visitare ogni singolo nodo del grafo bensì si fermerà quanto avrà raggiunto un nodo specifico $t \in N$. Non implementando una coda di priorità, andrà ad interrogare ogni nodo non visitato del grafo ad ogni ciclo, costruendo così il percorso nodo per nodo. Questo procedimento è indubbiamente esoso, infatti la sua complessità computazionale è $O(|N^2|)$.

2.1.3 List of Candidate

Questa ultima versione dell’algoritmo di Dijkstra (di cui è possibile leggere lo pseudocodice in fig.2.1) utilizza una coda di priorità particolare: gli elementi di questa coda infatti vengono aggiornati ad ogni ciclo. Considerano i nodi vicini di ogni nodo che viene visitato ed utilizza la loro distanza come valore di priorità; ogni volta che un nodo viene visitato viene anche rimosso dalla coda. Questa implementazione interroga molti meno nodi rispetto alle precedenti versioni, ma nonostante questo ha lo stesso caso peggiore di complessità dell’algoritmo ‘One to All’: $O((|N| + |A|) \log_2(|N|))$

2.2 “A Star”

L’algoritmo *A star* (conosciuto anche come A*) può essere considerata un’estensione dell’algoritmo di Dijkstra, poiché riesce ad ottenere una migliore

performance ed accuratezza usando l’euristica ¹. Per determinare in che modo A* deve estendere i suoi percorsi, necessita di minimizzare la seguente equazione:

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

Dove:

- n è il prossimo nodo del cammino
- $g(n)$ è il costo che ha accumulato il percorso dall’inizio (nodo s) fino al nodo n
- $h(n)$ è la funzione euristica

La funzione euristica, in questo caso, è la distanza più breve tra n ed il nodo finale (t) dunque *la linea retta* o meglio la **distanza euclidea** che lo separa dal ‘target’. In questo caso la complessità computazionale è relazionata ad h ed al numero di nodi esplorati per la costruzione del cammino minimo. Dunque il caso peggiore è $O(|N|) \equiv O(b^d)$ dove b è il numero medio dei successori per nodo e d i nodi esplorati per trovare la soluzione.

dettagli implementativi

Ad ogni iterazione:

1. Il nodo con il valore minore di $f(n)$ è estratto dalla coda (implementata come una coda di priorità)
2. Aggiorna i valori dei nodi vicini e li aggiunge alla coda in base alla priorità
3. L’algoritmo prosegue finché non viene raggiunto l’obiettivo

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

```

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

```

Figura 2.2: Pseudocodice dell’Algoritmo A*

La **Distanza Euidea** tra due punti è:

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

Dove i è un nodo del grafo e t è il nodo che vogliamo raggiungere, x e y rappresentano le coordinate Cartesiane (nei grafi di esempio utilizzati, dato che rappresentano delle città, vengono convertiti da latitudine e longitudine dei vari punti).

Dato che il calcolo della radice quadrata è molto costoso in termini di tempo, è necessario trovare un modo per evitare di ripetere questa operazione in caso un singolo nodo venga visitato più volte (nell’implementazione utilizzata viene salvato nel nodo e quindi non viene più ricalcolato).

2.3 Analisi dei Risultati

Di seguito sono presenti tabelle e figure che mostrano le caratteristiche degli algoritmi trattati nel capitolo. Le informazioni riguardo ad i grafi utilizzati per i test possono essere trovate nel capitolo 5, dedicato ai dettagli implementativi.

2.3.1 Performance a Confronto

Di seguito delle tabelle per visionare le performance degli algoritmi trattati nel corso di questo capitolo.

		Dijkstra one->all		Dijkstra one->one	
		weight	time (sec)	weight	time (sec)
Small <=2000	23755->27268	493	0.15652918815612793	493	1.7874250411987305
	15513->13984	1159	0.12487363815307617	1159	3.7926692962646484
	14591->26905	1227	0.11292171478271484	1227	3.0279810428619385
	7642->8365	1767	0.1174166202545166	1767	5.9814839363098145
	5456->27648	1887	0.12577557563781738	1887	8.55224061012268
AVG		1306.6	0.1275033473968506	1306.6	4.628359985351563
Medium 2000< <=5000	27257->23843	2471	0.1585693359375	2471	8.21094560623169
	6429->6531	2637	0.10044455528259277	2637	18.79756736755371
	234->14318	2638	0.10223102569580078	2638	21.64091420173645
	776->17207	3260	0.11026597023010254	3260	18.58165407180786
	8678->5881	4285	0.10606861114501953	4285	57.145034074783325
AVG		3058.2	0.11551589965820312	3058.2	24.875223064422606
Big >5000	15426->2070	6309	0.10060501098632812	6309	78.5000205039978
	26045->16486	7503	0.11378073692321777	7503	113.8067033290863
	3176->2586	10573	0.14719867706298828	10573	116.08205676078796
	22474->18289	14214	0.10036492347717285	14214	119.19370365142822
	22066->9174	15289	0.09669733047485352	15289	116.26782035827637
AVG		10777.6	0.1117293357849121	10777.6	108.77006092071534
tot		5047.466666666666	0.11824952761332194	5047.466666666666	46.091214656829834

		Dijkstra list of candidate		A*	
		weight	time (sec)	weight	time (sec)
Small <=2000	23755->27268	493	0.0006515979766845703	493	0.0003104209899902344
	15513->13984	1159	0.001478433609008789	1159	0.0006275177001953125
	14591->26905	1227	0.001099411010742188	1227	0.00026535987854003906
	7642->8365	1767	0.0019183158874511719	1767	0.0006570816040039062
	5456->27648	1887	0.003258943557739258	1887	0.0007936954498291016
	AVG	1306.6	0.0016634464263916016	1306.6	0.0005308151245117188
Medium 2000< <=5000	27257->23843	2471	0.003756999969482422	2471	0.00057220458984375
	6429->6531	2637	0.008446931838989258	2637	0.0024001598358154297
	234->14318	2638	0.009177207946777344	2638	0.001421213150024414
	776->17207	3260	0.00803065299987793	3260	0.00173187255859375
	8678->5881	4285	0.027533769607543945	4285	0.007309436798095703
	AVG	3058.2	0.01138911247253418	3058.2	0.0026869773864746094
Big >5000	15426->2070	6309	0.04720735549926758	6309	0.01497030258178711
	26045->16486	7503	0.07073330879211426	7503	0.01078176498413086
	3176->2586	10573	0.07935333251953125	10573	0.014153480529785156
	22474->18289	14214	0.09600615501403809	14214	0.03999781608581543
	22066->9174	15289	0.08968734741210938	15289	0.04842376708984375
	AVG	10777.6	0.07659749984741211	10777.6	0.025665426254272462
tot		5047.466666666666	0.029883352915445964	5047.466666666666	0.009627739588419596

Tabella 2.1: Performance degli algoritmi mono-criteria (tabella generata per comodità con la libreria tkinter)

La tabella 2.1 mostra le performance degli algoritmi trattati sulla mappa di Parigi, prendendo differenti punti di partenza e di arrivo e classificandoli in base al numero di nodi visitati (*small*, *medium*, *big*). Ovviamente il tempo di esecuzione dell'algoritmo *one to all* è costante, poiché visita tutti i nodi del grafo, ma è interessante vedere come rimanga più veloce dell'algoritmo *one to one*. Riguardo a gli altri due algoritmi (lista dei candidati e A*), il secondo è indubbiamente più vantaggioso del primo. La sua velocità di esecuzione è legata alla funzione euristica, quindi al calcolo della distanza euclidea, in un altro ambiente potrebbe non risultare egualmente vantaggioso.

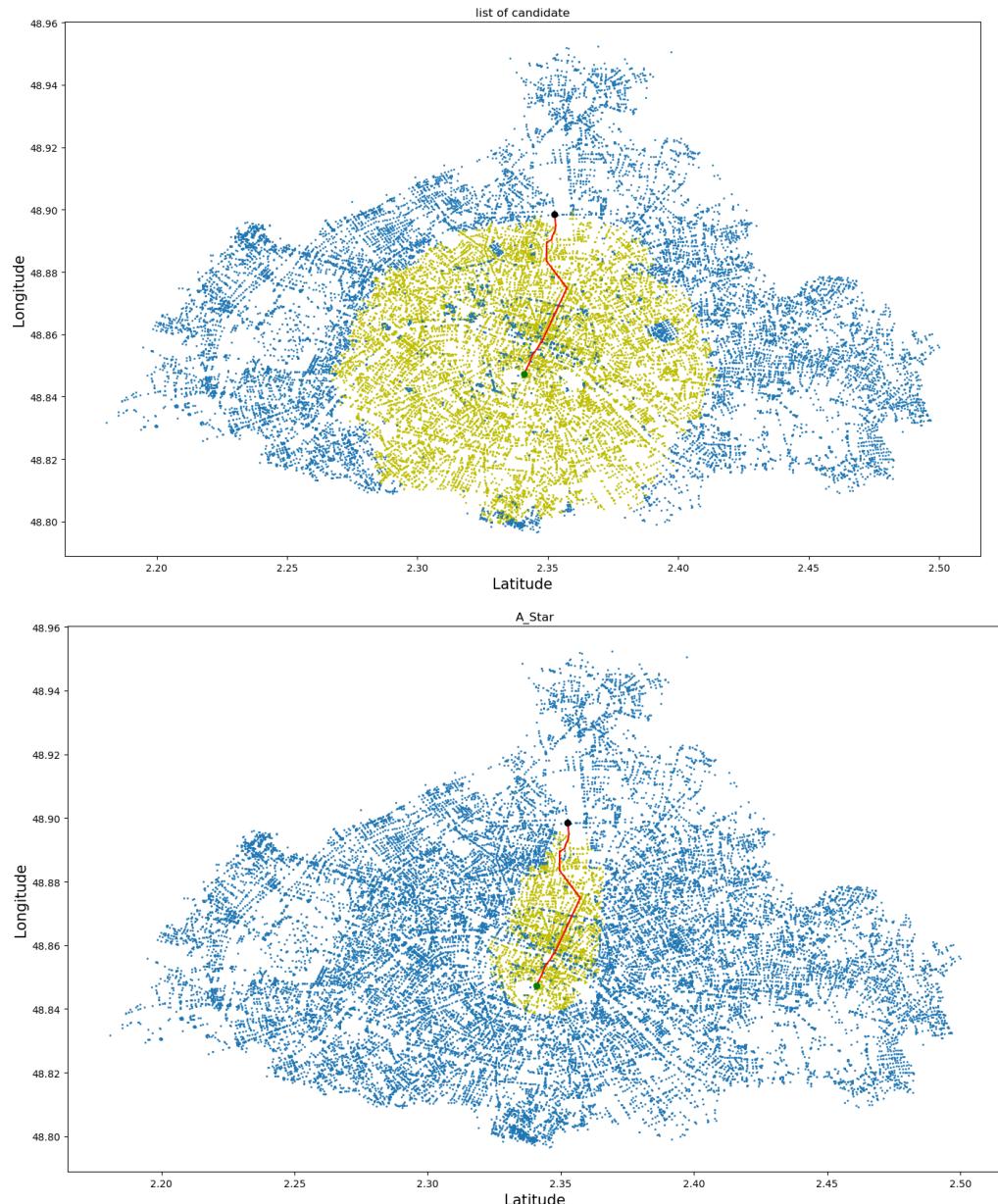
2.3.2 Nodi Visitati

I vari algoritmi trattati visitano i nodi in maniera differenti: ovviamente l'algoritmo *one to all* visiterà, come già detto, tutti nodi raggiungibili, l'algoritmo *one to one* e *list of candidate* visitano li stessi nodi, il secondo lo fa semplicemente molto più velocemente; l'algoritmo *A** invece visita i nodi in una maniera particolare (ovviamente dipendente dalla funzione euristica utilizzata). Questo infatti esplora un minor numero di nodi, che si traduce in un modesto risparmio di tempo di ricerca.

Nelle seguenti immagini si possono vedere le differenze, a livello di ricerca, dei due algoritmi. L'algoritmo di Dijkstra nel visitare i nodi crea una specie di circonferenza attorno al nodo iniziale, al contrario dell'A-star il quale sembra quasi seguire il cammino minimo, creando una sorta di forma ovaleggiante, tra il nodo sorgente e destinazione, attraversata dal cammino minimo.

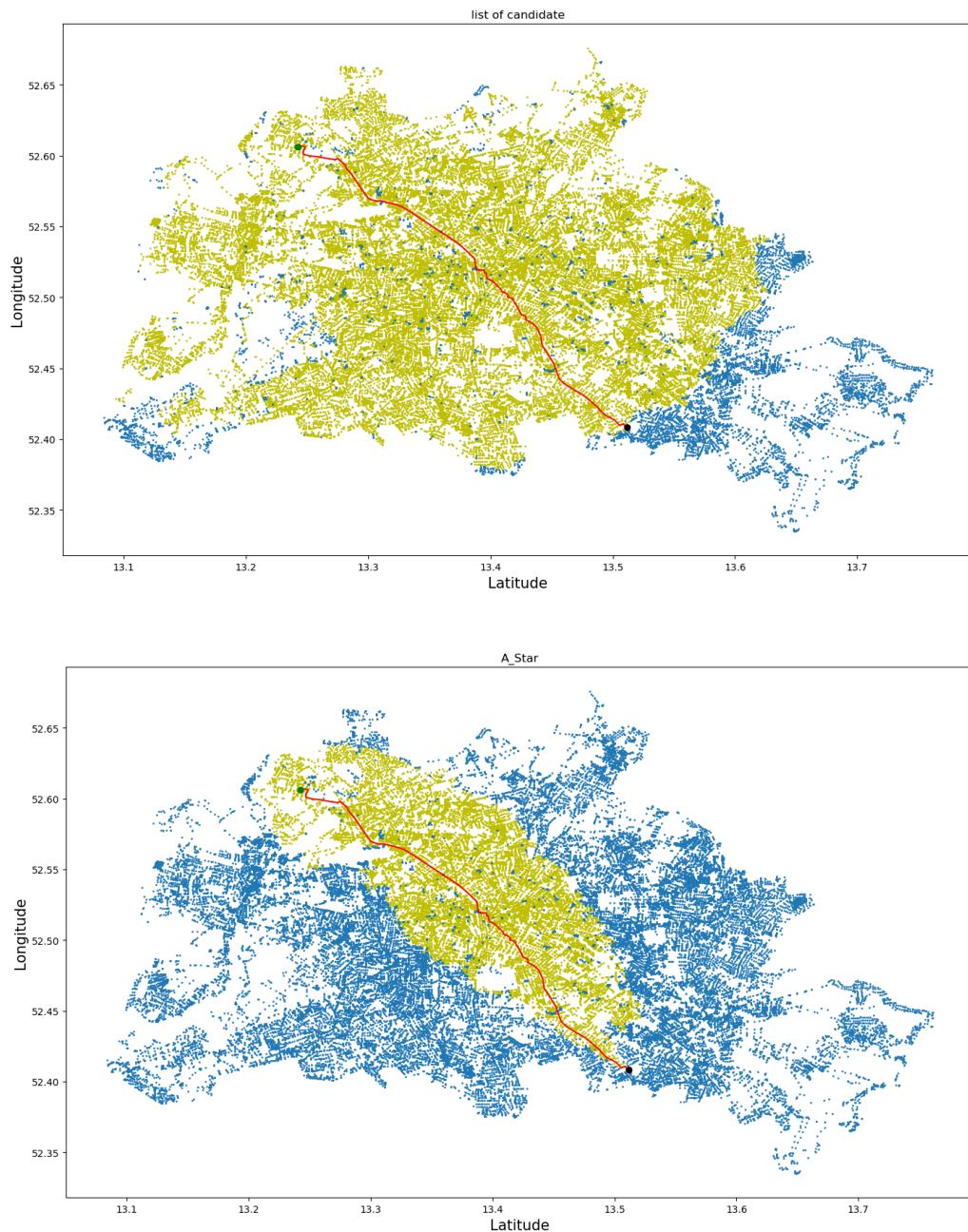
Legenda:

- **Blu:** Nodi del grafo.
- **Giallo:** Nodi visitati.
- **Verde:** Nodo sorgente.
- **Nero:** Nodo terminale.
- **Linea rossa:** Cammino minimo.



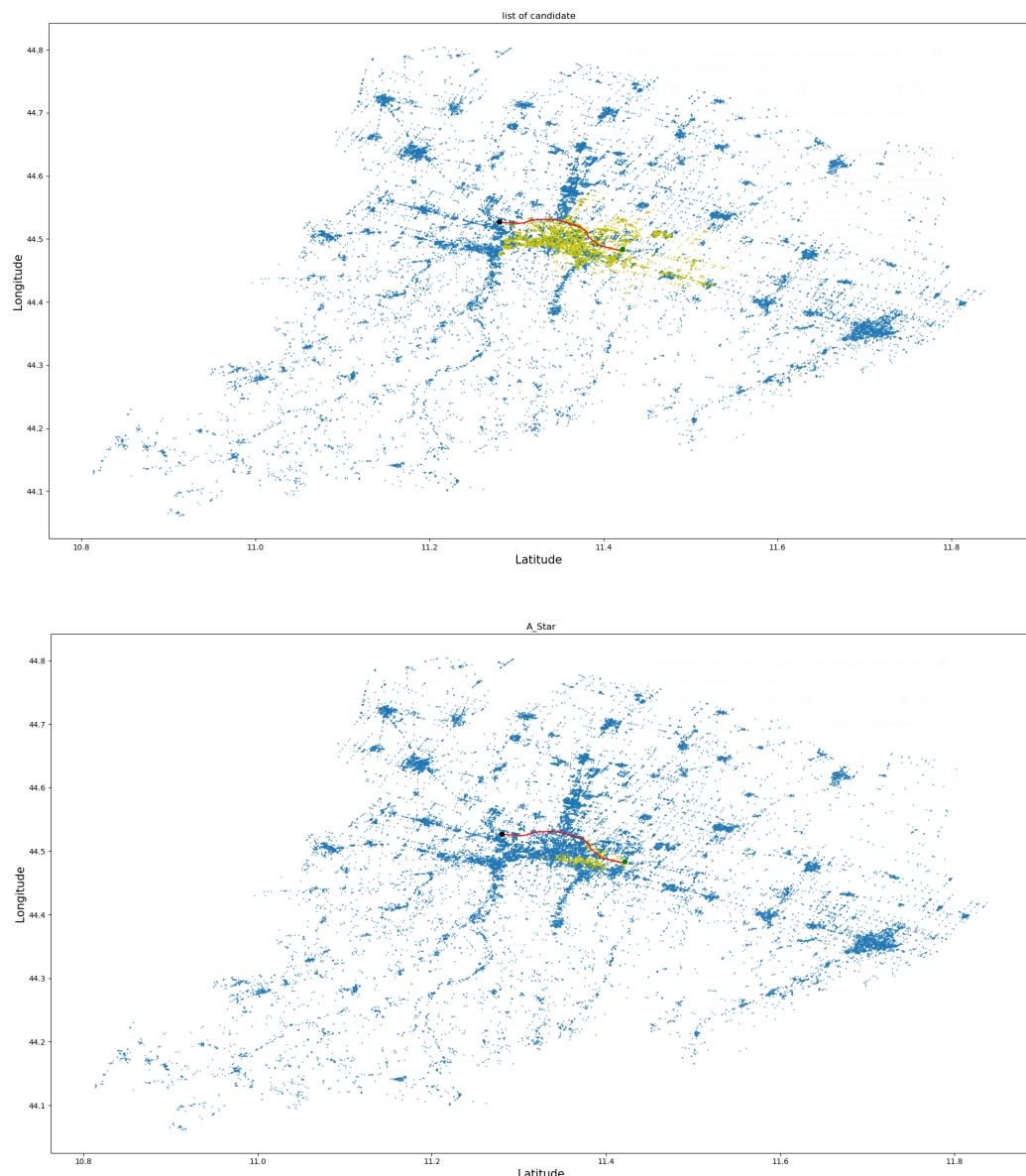
Source: 2070 → Target: 15426

Map: Parigi



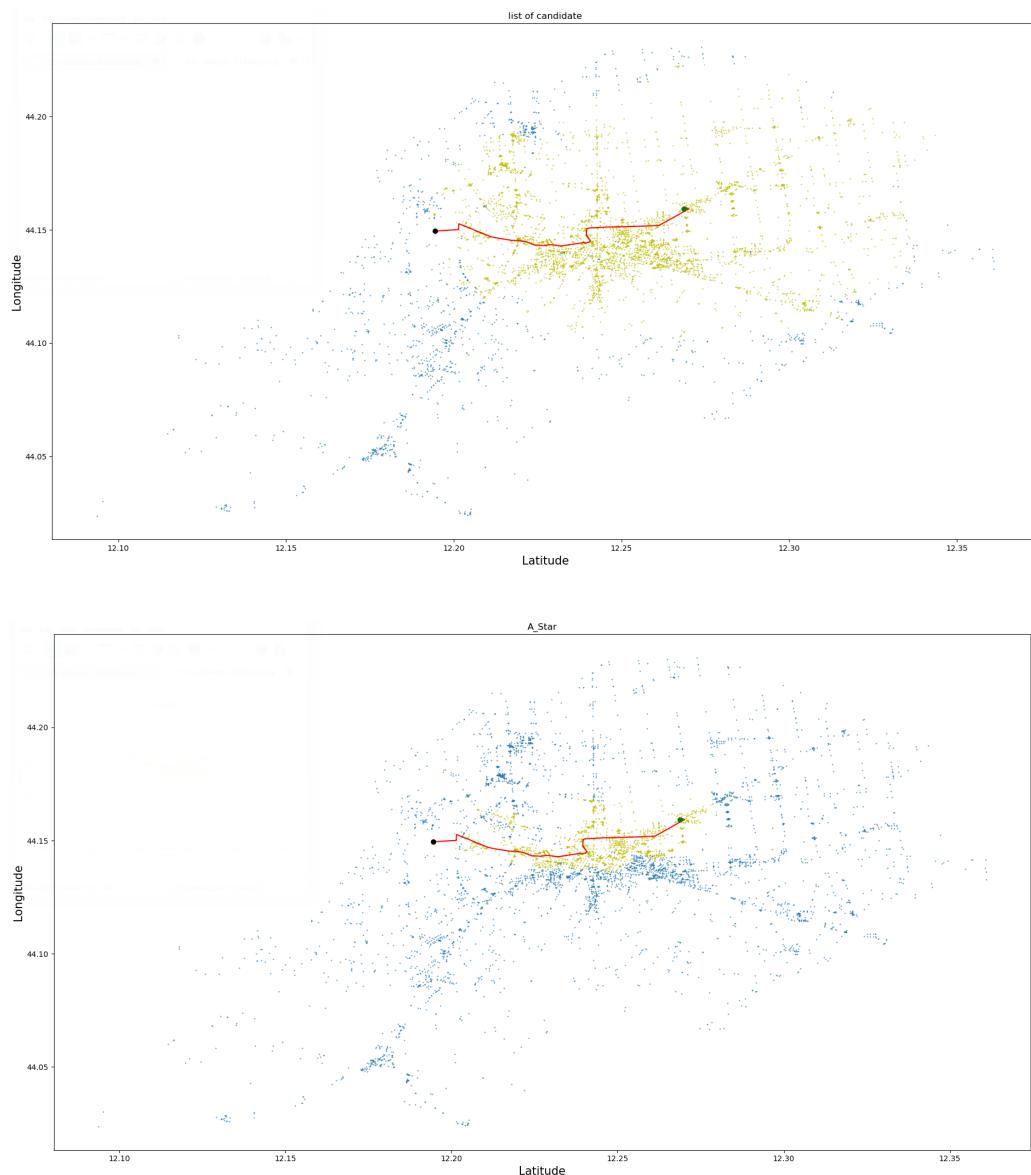
Source: 1000 → Target: 1510

Map: Berlino



Source: 1298 → Target: 23289

Map: Bologna (Provincia)



Source: 5221 → Target: 72

Map: Cesena (Comune)

Capitolo 3

Algoritmi ‘Bi-criteria’

In questo capitolo si andrà ad analizzare il problema dello studio dei percorsi brevi su grafi con più pesi, per poi studiare e confrontare gli algoritmi implementati, quindi valutarli a seconda dei risultati ottenuti. Tenendo in mente le espressioni matematiche formulate nel capitolo precedente possiamo soffermarci su alcune definizioni importanti:

Definizione 1. Concetto di Dominanza

Siano $X = (c^1(p_{s,t}), c^2(p_{s,t}))$, $Y = (c'^1(p_{s,t}), c'^2(p_{s,t}))$ due possibili cammini distinti da un nodo $s \in N$ ad un nodo $t \in N$. Si dice che X *domina* Y se e solo se $c^1(p_{s,t}) < c'^1(p_{s,t}) \wedge c^2(p_{s,t}) < c'^2(p_{s,t})$.

In questo caso X è detta **soluzione accettabile** (al contrario Y non verrebbe in alcun caso presa in considerazione).

Definizione 2. Inviluppo Convesso delle soluzioni

Sia S un insieme di soluzioni accettabili, l'inviluppo convesso (conosciuto anche come *Convex hull*) di X è il più piccolo insieme convesso contenente X. Dato che consideriamo due dimensioni per il calcolo del percorso, ci troviamo in uno spazio euclideo a due dimensioni. L'inviluppo convesso può quindi essere definito dalla seguente equazione:

$$\alpha \cdot c^1_{(p_{s,t})} + (1 - \alpha) \cdot c^2_{(p_{s,t})} = k \quad (3.1)$$

Dove k è una costante e $\alpha \in [0, 1]$.

Definizione 3. Fronte di Pareto

Il fronte di Pareto è l’insieme delle soluzioni ottimali, ovvero l’insieme di tutti i cammini *non-dominati*. Una soluzione può far parte del fronte di Pareto senza dominare altre soluzioni e senza far parte dell’inviluppo convesso delle soluzioni. È possibile vedere la differenza tra un fronte di Pareto e un inviluppo convesso delle soluzioni con le figure 3.2 e 3.3

3.1 Dijkstra applicato ad un problema Bi-criteria

Partendo dalla definizione classica di *grafo* è possibile fare un’astrazione e scrivere $G = (N, V, C)$, dove C è l’insieme dei costi. La suddetta definizione, quindi, è possibile estenderla ad un “grafo bi-criteria” come $G = (N, V, C^1, C^2)$, dove C^1 e C^2 sono gli insiemi dei due pesi; ma in questo modo l’algoritmo di Dijkstra non è applicabile. Utilizzando la 3.1 è possibile ridurre questa nuova definizione come a $G = (N, V, \alpha \cdot C_i^1 + (1 - \alpha) \cdot C_i^2)$ e, considerando che la 3.1 ha come risultato un elemento costante, sarà possibile un approccio simile agli algoritmi affrontati nel precedente capitolo. L’algoritmo di Dijkstra ‘*list of candidate*’ è il più vantaggioso, dato che non è possibile, in questo caso generico, individuare un funzione euristica che risolva in maniera ottimale l’equazione [11].

È facilmente comprensibile che utilizzando la 3.1, in base al valore dato ad α , otterremo un cammino che minimizza più un peso piuttosto che l’altro. Per esempio, ponendo il primo peso come *tempo di latenza* ed il secondo come *consumo energetico*, più il valore di α si avvicina a 0 più troveremo un percorso con una grande latenza ma un basso consumo energetico, mentre se questo valore si avvicina a 1 allora il percorso avrà meno latenza ma un consumo energetico inferiore.

La complessità computazionale è la stessa dell’algoritmo di Dijkstra mostrato precedentemente.

Poiché questo algoritmo ha alla base la formula dell’inviluppo convesso, significa che i risultati trovati andranno a creare la curva che rappresenta l’inviluppo convesso stesso.

3.1.1 Implementazioni dell’algoritmo di Dijkstra con due criteri

Nella figura 3.1 è riportato il codice che descrive il ragionamento della sezione precedente. Per utilizzare questo algoritmo ed ottenere un numero

```
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()
        if n = target do
            break
        for each neighbor v of n do
            alt ← alpha*firstW(n, v) + (1 - alpha)*secondW(n, v)
            alt ← alt + score[n]
            if alt < score[v] do
                score[v] ← alt
                prev[v] ← n
                Q.put(v, tmpScore)
    return score, prev
```

Figura 3.1: Pseudocodice dell’Algoritmo di Dijkstra con due criteri

soddisfacente di risultati è necessario richiamarlo ogni volta con diversi valori per α , ricordando che è compreso in $[0, 1]$. Sono stati quindi elaborati due processi per svolgere nella maniera più rapida e precisa questa operazione.

Iterazione Dijkstra bi-criteria

Questa è un implementazione piuttosto basilare, si basa semplicemente sul richiamare l'algoritmo partendo con un valore di α uguale a 0, per poi aumentarne questo valore di un numero arbitrario chiamato **precisione**; se il risultato è differente da quello precedente allora il cammino viene salvato ed il valore di α dimezzato, altrimenti si prosegue, fino a che non si raggiunge $\alpha = 1$. La precisione assumerà un valore compreso tra 0 e 1, più la precisione viene definita bassa, più significa che il suo valore si avvicina ad 1, più è alta, più si avvicina a 0; nelle tabelle (3.1) è possibile vedere l'algoritmo applicato con due differenti valori di precisione.

Questa soluzione richiede un tempo di elaborazione in relazione alla precisione utilizzata, infatti più questa è piccola, più volte verrà richiamata la funzione, dunque si otterranno differenti risultati (*soluzioni accettabili*); ma ciò significa anche che la funzione spesso darà dei risultati già individuati, sprecando cicli preziosi.

Dijkstra bi-criteria con ricerca binaria

Questa è una versione un po' meno grezza della precedente, dato che utilizza un algoritmo di ricerca binaria. Viene eseguita la funzione almeno due volte, una con $\alpha = 0$, poi una con $\alpha = 1$; se generano risultati differenti allora il valore di α viene dimezzato, e così via finché non si trovano più soluzioni. Seppur più efficiente rispetto ad un'iterazione con alta precisione, in termini di tempo e di risultati trovati non risulta essere perfetto.

Di seguito sono presenti delle tabelle (3.1) che mostrano i risultati tra due esecuzioni dell'iterazione dell'algoritmo bi-criteria di Dijkstra, con differenti precisioni, ed i risultati dell'implementazione con ricerca binaria (3.2). Nella tabella due algoritmi sono applicati sulla mappa di Parigi (vd. ch. 6), partendo dal nodo di indice 2000 e terminare in 2689.

Valore di α	PESO 1	PESO 2
Iterazione con precisione = 0.2		
$\alpha = 0$	9231	12827
$\alpha = 0.2$	7153	12884
$\alpha = 0.7$	6091	14230
$\alpha = 0.55$	6317	13754
$\alpha = 0.875$	5944	14824
$\alpha = 1$	5840	16763
AVGtime: 2.112018585205078		
Iterazione con precisione = 0.05		
$\alpha = 0$	9231	12827
$\alpha = 0.05$	7153	12884
$\alpha = 0.525$	6317	13754
$\alpha = 0.7125000000001$	6091	14230
$\alpha = 0.8062500000004$	6030	14430
$\alpha = 0.8531250000005$	5944	14824
$\alpha = 0.9265625000006$	5874	15649
$\alpha = 1$	5840	16763
AVGtime: 7.837886095046997		

Tabella 3.1: Output dei due algoritmi di Dijkstra bi-criteria iterativi con differenti precisioni

Da questi risultati è infatti possibile notare come una bassa precisione permette di avere un modesto insieme di risultati in tempi ragionevolmente bassi, differentemente si può dire dell'iterazione con alta precisione (ovvero la seconda tabella, dove il valore della precisione è di 0.05), infatti questo è in grado di trovare più soluzioni rispetto al precedente, ma al costo di un tempo di elaborazione maggiore. Con la ricerca binaria è possibile ottenere un ottimo rapporto tra il numero di soluzioni individuate ed il tempo di elaborazione, anzi, in questo caso trova addirittura più risultati rispetto al-

Valore di α	PESO 1	PESO 2
Dijkstra con ricerca binaria		
$\alpha = 1$	5840	16763
$\alpha = 0$	9231	12827
$\alpha = 0.5$	7153	12884
$\alpha = 0.75$	6091	14230
$\alpha = 0.625$	6317	13754
$\alpha = 0.875$	5944	14824
$\alpha = 0.812$	6030	14430
$\alpha = 0.937$	5874	15649
$\alpha = 0.9680000000001$	5855	16199
$\alpha = 0.9660000000001$	5863	15959
$\alpha = 0.9740000000001$	5847	16493
AVGtime: 4.851694107055664		

Tabella 3.2: Output dei dell’algoritmo di Dijkstra bi-criteria con ricerca binaria

l’algoritmo di Dijkstra bi-criteria con alta precisione, in tempi ridotti (ma pur sempre maggiori rispetto all’algoritmo che itera con una bassa precisione). Nel corso di questo studio vedremo che non sarà sempre così.

3.2 Algoritmo di Label-setting

Come precedentemente scritto, l’algoritmo di Dijkstra applicato a due criteri non è in grado di trovare tutte le soluzioni possibili (con qualsiasi livello di precisione), bensì solo quelle che compongono l’inviluppo convesso. Dunque se noi volessimo trovare un più ampio insieme di soluzioni non dominate, dunque se volessimo trovare il *Fronte di Pareto*, dobbiamo affidarci ad un algoritmo di Label-setting [18] [27]. Di seguito è presente un’esempio di quanto detto, dove vengono messi a confronto le soluzioni trovate da Dijkstra applicato a due criteri e l’algoritmo trattato in questo capitulo-

lo. Nell'esempio vengono utilizzati la distanza ed il pericolo come valori che contraddistinguono i due pesi in questione.

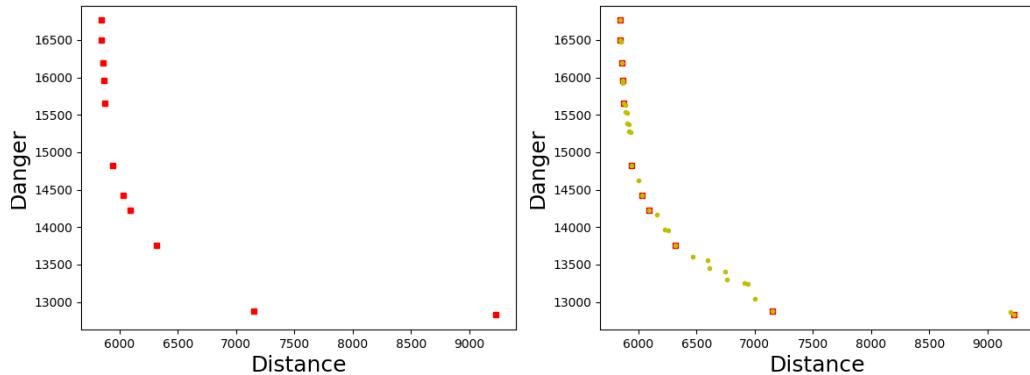


Figura 3.2: Soluzioni trovate da Figura 3.3: Soluzioni trovate dall'algoritmo di Dijkstra bi-criteria con ricerca binaria goritmo Label-setting

Questo esempio è stato calcolato utilizzando la mappa di Parigi, coprendo il tragitto che va dal nodo 2000 al nodo 2689.

Nel grafo in figura 3.3 sono presenti solo alcuni punti appartenenti alla *convex hull* (quadrati rossi), mentre nel grafico affianco snono presenti tutte le soluzioni ottenute con l'algoritmo di label-setting (punti gialli). Infatti, come è anche possibile vedere, nel grafico a destra alcuni punti gialli trovano corrispondenza in tutti i punti trovati nel grafico a sinistra. Le soluzioni trovate quindi dall'algoritmo di label-setting sono tutte accettabili, dunque il suo insieme di soluzioni è più completo.

3.2.1 Implementazione

Questo algoritmo è basato, da come si può intuire dal nome, sul creare e salvare appunto dei ‘*label*’ dove sono contenute le informazioni sul percorso che si affronta per andare da un nodo ad un altro del grafo. I label sono strutturati in questo modo:

$$(peso1, peso2, propriet, pred, indicePropriet, indicePred)$$

I primi due parametri rappresentano i costi del cammino dal nodo sorgente, il terzo ed il quarto sono rispettivamente il nodo possessore del label ed il suo

predecessore (o vero il nodo antecedente, da dove “arriva” il cammino), gli ultimi due valori invece, utilizzati per il ‘backtracking’ del percorso, indicano la posizione del label in corrispondenza del loro nodo possessore e del loro predecessore.

```

function DijkstraAlgorithm(source, target):
    originLabel ← (0, 0, source, Null, 0, Null)
    source.add(originLabel)
    create priorityQueue Q
    Q.put(originLabel, 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) + actualLabel[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())

```

Figura 3.4: Pseudocodice dell’Algoritmo Label-setting

Come si può notare dalla prima parte dello pseudocodice, l’algoritmo subisce una grande influenza da parte dell’algoritmo di Dijkstra data la somiglianza. Ogni volta che un label viene calcolato si vano in contro a tre differenti casi:

- i. Il label calcolato è dominato, dunque l’algoritmo lo scarta.

- ii. Il label calcolato domina altri label, quindi lui viene aggiunto alla lista dei label mentre i dominati rimossi.
- iii. Il label calcolato non domina e non viene dominato da nessuno, dunque viene aggiunto alla lista dei label.

Esempio:

È possibile fare un esempio di esecuzione dell'algoritmo osservando la figura 1.1 partendo dal nodo 1. Come prima cosa verrà creato il label avente i seguenti valori: $(0, 0, 1, \text{null}, 0, \text{null})$, poi, per ogni vicino (2 e 3), i label: $(3, 4, 2, 1, 0, 0)$ e $(5, 3, 3, 1, 0, 0)$ e inseriti all'interno della coda. Viene di seguito estratto il label che ha come possessore il nodo 2 e poi continua in questo modo fino ad arrivare al nodo 6. Ogni nuovo label trovato viene quindi studiato per capire se domina, è dominato o semplicemente è una soluzione accettabile.

Complessità computazionale

Il caso peggiore che potrebbe essere incontrato dall'algoritmo è esplicato dalla figura 3.5, ovvero la remota possibilità di trovare una soluzione accettabile e non dominante ad ogni iterazione, dunque dove il primo peso è zero ed il secondo è più grande del corrispettivo peso che lo precede:

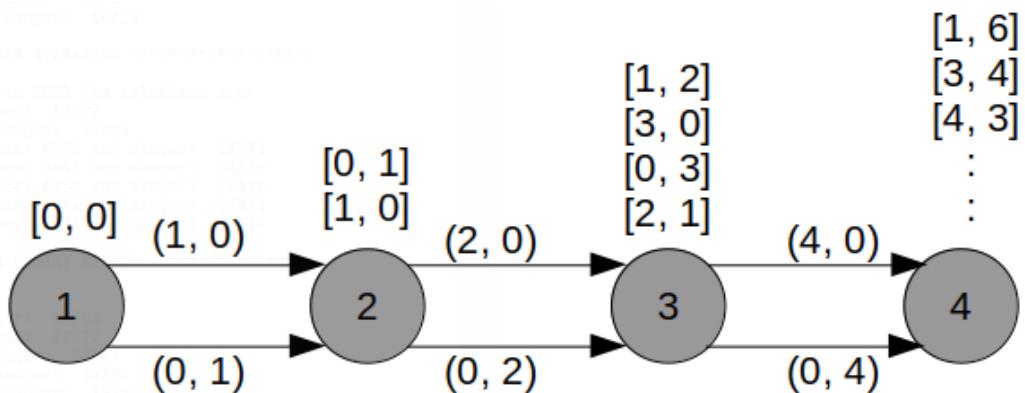


Figura 3.5: Caso peggiore dell'algoritmo di Label-setting

É possibile dunque affermare che la complessità di questo algoritmo è $O(2^{n-1})$ dove n è il numero di nodi.

3.3 Algoritmo di Label-setting con minoranti pregressi

Volendo migliorare i tempi di esecuzione dell’algoritmo di label-setting si può pensare di affiancare alla sua computazione una fase di *pre-elaborazione*. Questa fase può essere effettuata sfruttando l’algoritmo di Dijkstra, applicato a due criteri, visto precedentemente. Il vantaggio risiederebbe nella possibilità di conoscere alcuni dei label più vantaggiosi di diversi nodi sparsi lungo il percorso, in questo modo sarà possibile, per l’algoritmo di label-setting, eliminare in anticipo i cammini che risultano avere una proiezione dominata. Dunque, per svolgere il ragionamento sopracitato si prendono due nodi, come sempre s e t entrambi in $\in N$, e si esegue l’algoritmo di Dijkstra bi-criteria due volte, la prima con $\alpha = 0$, al seconda con $\alpha = 1$, minimizzando così entrambi i pesi. Al termine di ciò siamo a conoscenza del label, appartenente a t con il primo peso più piccolo ed il secondo peso più grande e del label con il peso numero due più piccolo ed il primo più grande; ma non solo, in questo modo troviamo anche tutti i label che appartengono al percorso che porta ai label di t ; presupponendo che questi appartengano ai ‘fronti di pareto’ dei nodi del percorso, l’algoritmo può capire se i label in cui si va incontro durante la computazione dell’algoritmo di label-setting sono dominati oppure no

3.3.1 Algoritmo di label-setting con minoranti pregressi invertito

Similmente a come descritto subito sopra, si può realizzare un’altra implementazione, che comprende una fase di *pre-elaborazione*, simile alla precedente. Considerando sempre s e t , come nodi rispettivamente di partenza e

di arrivo, si esegue l'algoritmo di Dijkstra bi-criteria al contrario, ovvero da t a s , sempre con $\alpha = 0$ e $\alpha = 1$. Come descritto prima si ricavano alcuni dei label non dominati per ogni nodo del cammino, ma questa volta a partire, appunto, dal fondo. Dunque sommando il valore di uno dei pesi trovato nella fase di pre-elaborazione con il corrispettivo valore calcolato dall'algoritmo in esecuzione, si può trovare la sua proiezione e dedurre se questa sarà dominata o meno.

Il concetto, molto complesso a parole, risulta, in realtà, basilare nel ragionamento; può essere riassunto e semplificato in questo modo:

- pp = miglior “peso 1” del percorso (calcolato nella pre-elaborazione).
- pn = miglior “peso 1” del nodo (“).
- ap = valore *attuale* del “peso 1” calcolato fino a quel momento dall’algoritmo.
- res = proiezione del peso da quello specifico nodo.

$$res = pd - nd + ad$$

Poiché siamo a conoscenza delle due soluzioni (sia con $\alpha = 0$ che = 1) calcolate con Dijkstra, è possibile constatare se la proiezione dei label per ogni peso.

Questa implementazione funziona però unicamente su grafi non orientati, al contrario della versione “non invertita”.

3.4 Algoritmo Bidirezionale Bi-criteria

In letteratura gli algoritmi bidirezionali sono molto elogiati per la loro capacità di accelerare la ricerca all’interno di una rete [10] [29], questa sezione si concentrerà dunque sull’analisi di un algoritmo bidirezionale in grado di calcolare alcuni risultati del fronte di Pareto. L’idea alla base di questo algoritmo è di alternare una ricerca normale, quindi dal nodo di partenza al nodo finale, ad una invertita, ovvero dal nodo finale a quello iniziale. È

facile dedurre che anche questo algoritmo è efficace unicamente in grafi non orientati, altrimenti troverebbe dei cammini non percorribili, inoltre le sue performance sono dipendenti anche dalla configurazione del grafo preso in considerazione. Infatti il livello di accelerazione della ricerca, rispetto ad una ricerca effettuata con altri algoritmi i.g. Label-setting, potrebbe variare in base a caratteri quantitativi e qualitativi della rete; i grafi presi in considerazione per i test sono piuttosto uniformi ed equilibrati, dunque si avrà una discreta accelerazione.

L’alternanza tra la ricerca all’avanti e quella all’indietro può essere implementata in diversi modi, in questa ricerca, per semplicità, vengono equamente alternati uno alla volta, ma sono possibili diverse soluzioni, come ad esempio considerare il label più piccolo (a livello lessicografico) indifferentemente dalla sua direzione di ricerca. Differenti scelte potrebbero avere un impatto sulle performance, ma sarebbero comunque irrilevanti rispetto al guadagno, in termini di tempo, che si andrà comunque ad ottenere.

Abbiamo detto che la ricerca viene effettuata in due direzioni, l’andamento di queste due ricerche separate è simile a quello di un algoritmo unidirezionale, quindi nuovi label vengono calcolati a partire dai vicini di un nodo ed aggiunti al set dei label di questi. Si avranno due set, uno per ogni direzione, per ogni nodo, dove i label dominati vengono rimossi per far posto a quelli dominanti. Quando si va ad investigare un nodo in cui sono presenti dei label “provenienti” dalla direzione opposta, si combinano i due percorsi e si effettuano le dovute valutazioni di dominanza, in caso il cammino trovato risulta essere una soluzione accettabile viene aggiunto alla lista delle soluzioni. Ogni qual volta vengano combinati dei percorsi che portano ad una soluzione accettabile, tutte le eventuali soluzioni dominate da quest’ultima vengono rimosse dalla lista che contiene i risultati (i rimanenti andranno a comporre il fronte di Pareto).

3.4.1 Implementazione

Come già detto l'algoritmo riprende dei concetti ormai appurati, come ad esempio la struttura dei label, che rimangono invariati rispetto agli algoritmi precedenti, e l'andamento dell'algoritmo in relazione alla singola direzione.

La caratteristica più rilevante riguarda la **condizione di stop**: mentre in tutte le implementazioni viste fin'ora gli algoritmi terminavano nel momento in cui la lista contenente tutti i label temporanei si svuotava, qui il *loop* si interrompe quando la somma dei label più piccoli delle liste (in entrambe le direzioni) risulta un valore dominato da almeno un elemento contenuto nella lista dei risultati; in poche parole:

L'algoritmo termina quando $[\min_i(Q_f) + \min_i(Q_b)]$ è dominato da un qualsiasi $R \in L_{results}$, dove f e b abbreviano *forward* e *backward*.

L'immagine 3.6 mostra lo pseudocodice relativo.

Caso particolare della condizione di stop

Considerando C_i un cammino (ricongiunto nel nodo i) che risulta essere una soluzione accettabile del percorso, ma che non è stato ancora trovato, implica che C_i non è dominato da nessuna soluzione $R \in L_{result}$, quindi anche da nessun r_i (risultato che si congiunge nel nodo i). Per la condizione di stop $r_i \leq [\min_i(Q_f) + \min_i(Q_b)]$ allora:

$$\exists i : C_i = c_i^f + c_i^b < r_i \leq [\min_i(Q_f) + \min_i(Q_b)]$$

Dunque

$$\exists i : c_i^f + c_i^b < [\min_i(Q_f) + \min_i(Q_b)] \quad (3.2)$$

Dato che C_i non è ancora stato trovato, significa che $\forall i : c_i^f \geq \min_i(Q_f) \vee c_i^b \geq \min_i(Q_b)$, ma è impossibile che accadano entrambi contemporaneamente, altrimenti il cammino sarebbe dominato ed andrebbe contro la (3.2). Dunque $\exists i : c_i^f \geq \min_i(Q_f) \wedge (c_i^b < \min_i(Q_b) \vee c_i^b = \min_i(Q_b)) \vee (c_i^f < \min_i(Q_f) \vee c_i^f =$

$$\min_i(Q_f)) \wedge c_i^b \geq \min_i(Q_b)$$

Considerando solo uno dei due casi, poiché l’altro deducibile allo stesso modo, si assume che

$$\exists i : (c_i^f < \min_i(Q_f) \vee c_i^f = \min_i(Q_f)) \wedge c_i^b \geq \min_i(Q_b)$$

Quindi il label sul nodo i ricercato a partire dal nodo terminale non è stato ancora calcolato; mentre quello che parte dal sorgente sì, di conseguenza sono stati calcolati anche i label dei suoi vicini (altrimenti non sarebbe $< \min_i(Q_f)$). Sia C^b il label che andrà a comporre sul nodo i il cammino finale, consideriamo il predecessore e chiamiamolo C'^b . Al contempo sia C^f il label sul nodo i proveniente dall’origine, quindi il nodo che possiede C^f è lo stesso che possederebbe C^b . C^f è un label che è stato calcolato e visitato (il che significa che è **permanente**), considero dunque C'^f un label calcolato **a partire** da C^f , quindi appartenente ad un nodo vicino; quest’ultimo label può essere sia un label permanente, che **temporaneo**, ovvero solamente calcolato, ma non ancora visitato. Al contrario il label C'^b , non essendo ancora stato neanche calcolato C^b , può essere temporaneo o non ancora calcolato. Analizzando le varie possibilità si ricade in quattro diversi casi:

C'^f è permanente e C'^b è temporaneo: Il cammino viene trovato dall’algoritmo e la soluzione aggiunta al fronte di Pareto.

C'^f è permanente e C'^b non calcolato: Ci riconduce alla situazione iniziale dove $\exists i : (c_i'^f < \min_i(Q_f) \vee c_i'^f = \min_i(Q_f)) \wedge c_i'^b \geq \min_i(Q_b)$.

C'^f e C'^b sono temporanei: Il cammino non è ancora stato trovato, ma entrambi i label sono stati calcolati, ci si riconduce quindi ad un caso simile al primo.

C'^f è temporaneo e C'^b non calcolato: Essendo il label all’interno della coda, significa che $C'^f \geq \min_i(Q_f)$, allo stesso modo $C'^b \geq \min_i(Q_b)$ poiché non è stato neanche calcolato. Quest’ultimo caso va contro

all'equazione 3.2, quindi non esiste una soluzione che può far parte del fronte di Pareto.

```

function BidirectionAlgorithm(source, target):
    create priorityQueue Q[f]    //forward
    create priorityQueue Q[b]    //backward
    create list Lresults
    originLabel ← (0, 0, source, Null, 0, Null)
    source.add(originLabel)
    Q[f].put(originLabel)
    targetLabel ← (0, 0, target, Null, 0, Null)
    source.add(targetLabel)
    Q[b].put(targetLabel)
    while [mini(Q[f]) + mini(Q[b])]   

        is not dominated by any R ∈ Lresults do
        d ← getDirection()    //forward o backward
        actualLabel ← Q[d].extract_min()
        owner ← actualLabel[2]
        for each neighbor v of owner do
            firstW ← weightOne(owner, v) + actualLabel[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(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 ← combine(label, v.labels(!d))
                    Lresults.addResults(results)

```

Figura 3.6: Pseudocodice dell'Algoritmo bidirezionale

3.5 Analisi dei risultati

Di seguito vengono effettuati dei test degli algoritmi trattati nel capitolo su grafi di diverso tipo e dimensioni.

Dove possibile sono state utilizzate le stesse mappe e gli stessi punti utilizzati nell’analisi dei risultati del capitolo precedente, in questo modo è possibile fare anche un confronto con gli algoritmi mono-criteria riguardo ai nodi visitati ed ai percorsi trovati.

3.5.1 Performance a confronto

In questa sezione vengono analizzati i risultati degli algoritmi trattati nel capitolo.

Dalle tabelle 3.3 si riprendono i nodi utilizzati nelle 2.1, sempre utilizzando la mappa di Parigi. Essendo questa rappresentata tramite un grafo orientato sono stati applicati tutti gli algoritmi bicriteria possibili. Subito di seguito invece è presente un test Dove vengono misurate le performance di tutti gli algoritmi trattati in relazione ai cammini trovati ed al tempo impiegato.

Dijkstra Bicriteria binary search		
Source->target	Nº 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

Source->target	Label-setting algorithm		Lower bound improvement		
	Nº solution	Time (sec)	Nº 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

Tabella 3.3: Risultati degli algoritmi applicabili a grafi orientati (I fogli excel sono stati generati per comodità tramite la libreria *xlsxwriter*).

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 pre-processing	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

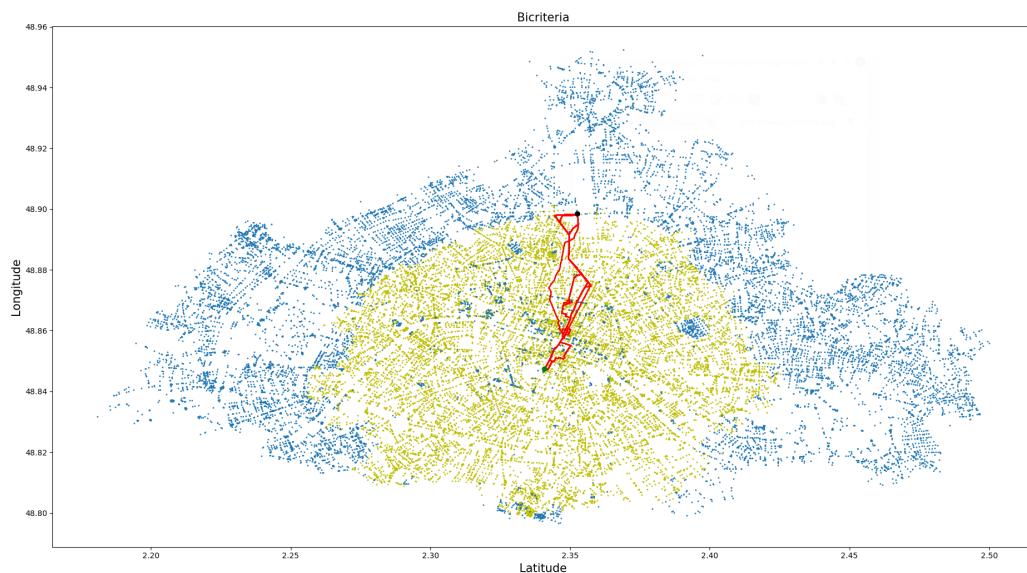
Tabella 3.4: Risultati degli algoritmi bicriteria in un grafo non orientato con 200000 nodi connessi a cascata tramite archi dai valori casuali (tutti i cammini partono da 0 e i nodi sono collegati ad altri 4 nodi, due antecedenti e due precedenti)

3.5.2 Nodi visitati

Come nel capitolo precedente le immagini mostrano i nodi ed i percorsi trovati dagli algoritmi in diversi grafi.

Legenda:

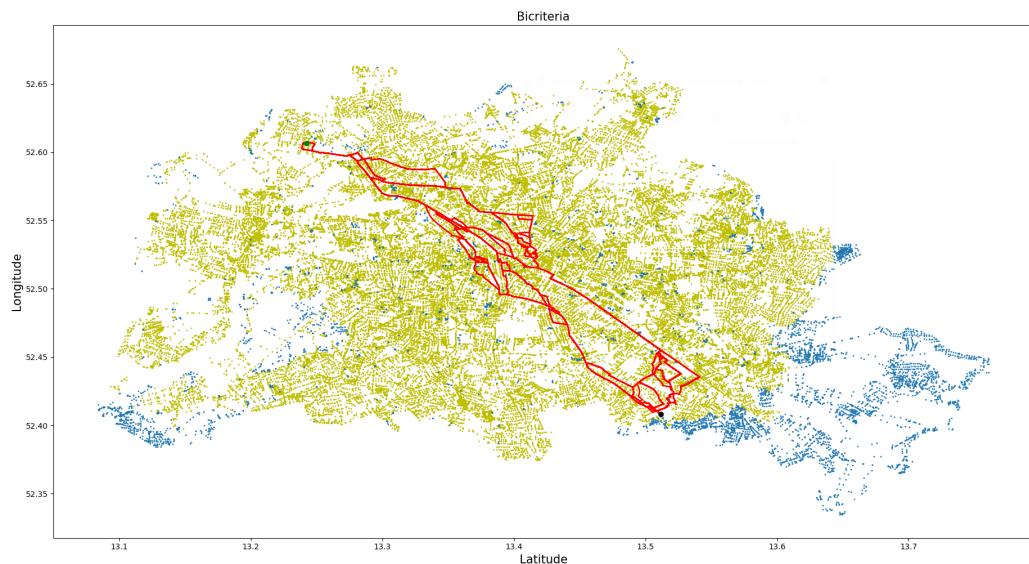
- **Blu:** Nodi del grafo.
- **Giallo:** Nodi visitati.
- **Verde:** Nodo sorgente.
- **Nero:** Nodo terminale.
- **Linea rossa:** Cammino minimo.



Source: 2070 → Target: 15426

Map: Parigi

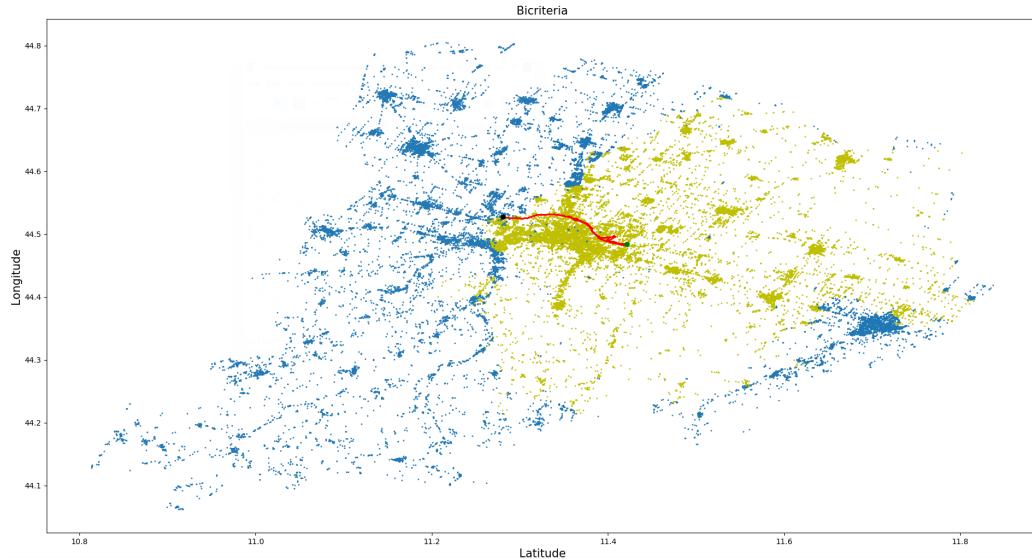
Figura 3.7: Algoritmo Label-setting su Parigi (mappa orientata).



Source: 1000 → Target: 1510

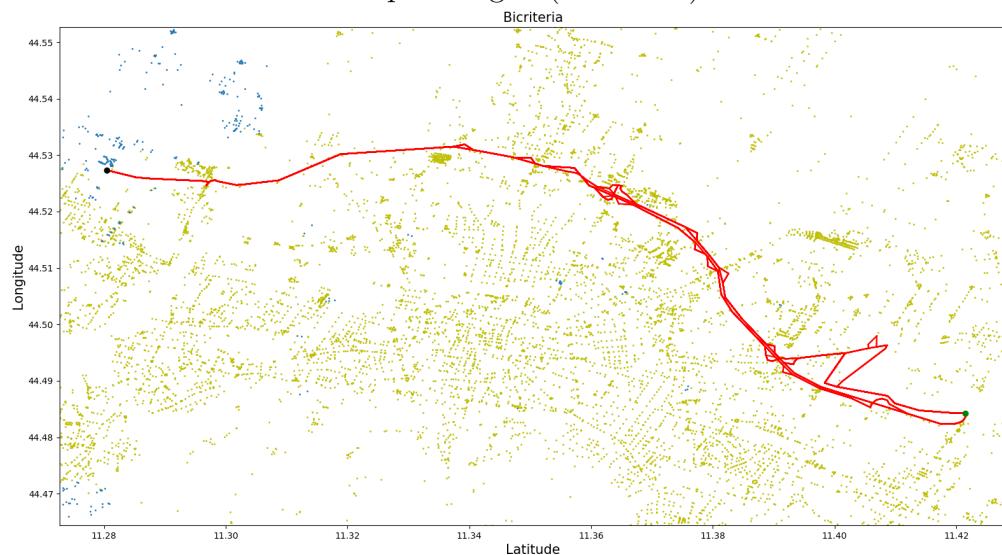
Map: Berlino

Figura 3.8: Algoritmo Label-setting su Berlino (mappa orientata).



Source: 1298 → Target: 23289

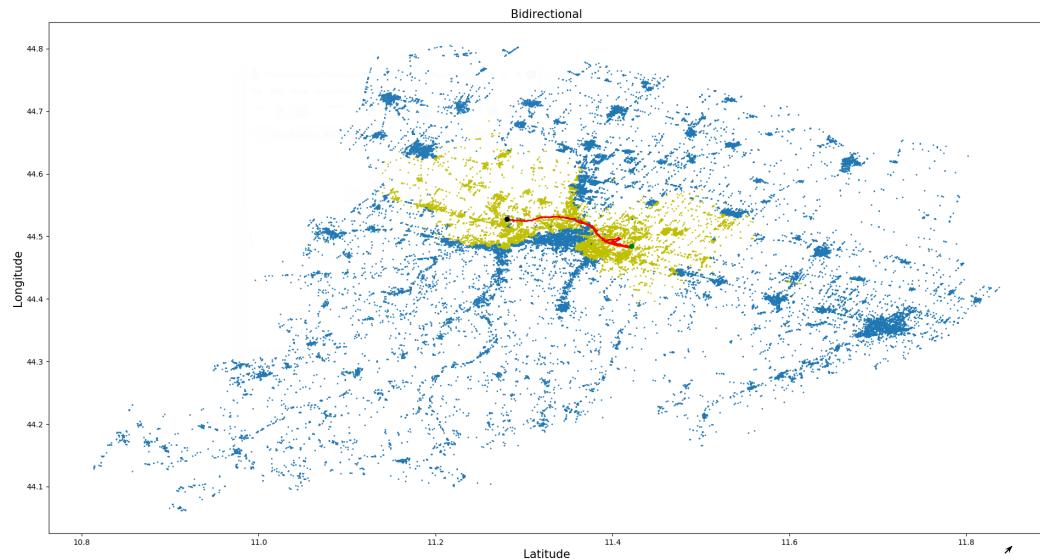
Map: Bologna (Provincia)



Source: 1298 → Target: 23289

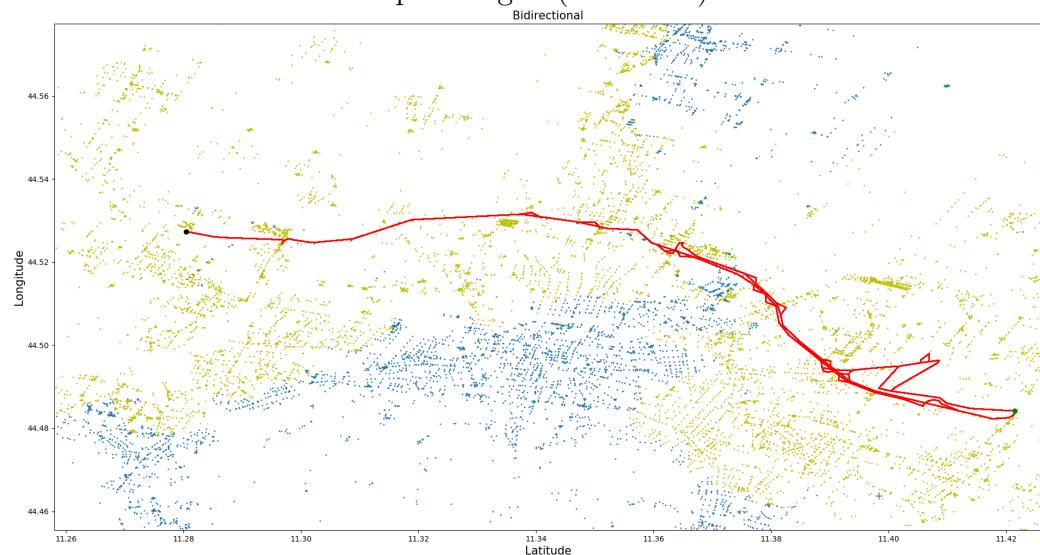
Map: Bologna (Provincia)
particolare

Figura 3.9: Algoritmo Label-setting su Bologna (mappa non orientata).



Source: 1298 → Target: 23289

Map: Bologna (Provincia)



Source: 1298 → Target: 23289

Map: Bologna (Provincia)

detttaglio

Figura 3.10: Algoritmo bidirezionale su Bologna (mappa non orientata).

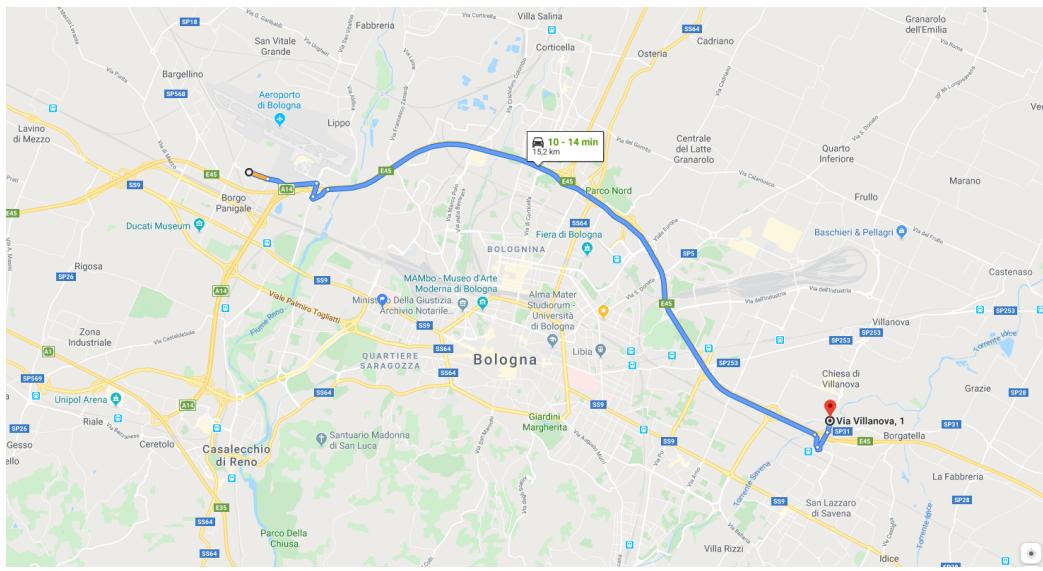
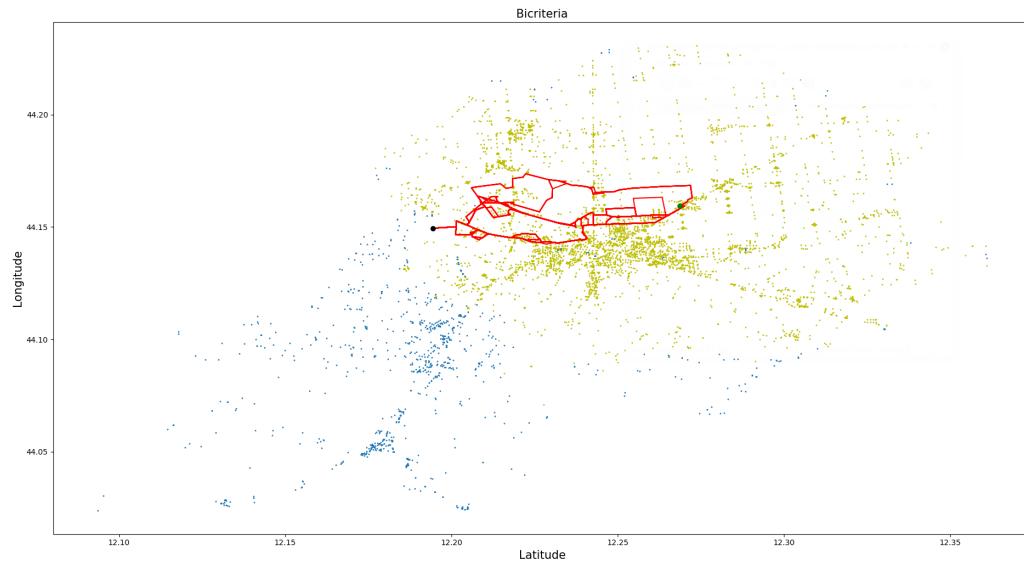
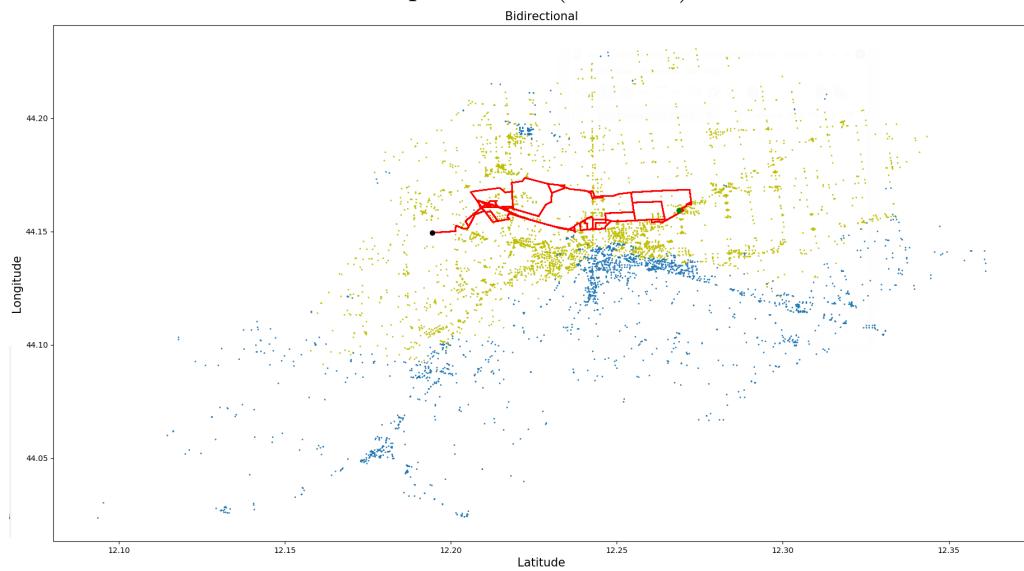


Figura 3.11: Esempio del percorso su Bologna preso dal sito *Google Maps*.



Source: 5221 → Target: 72

Map: Cesena (Comune)



Source: 5221 → Target: 72

Map: Cesena (Comune)

dettaglio

Figura 3.12: Algoritmo Label-setting (sopra) e Bidirezionale (sotto) su Cesena (mappa non orientata).

Capitolo 4

Considerazioni Personali

Nel seguente capitolo parlerò dei miei pensieri e delle esperienze personali riscontrate durante la stesura di questa tesi triennale.

4.1 Idee Scartate

Algoritmo Genetico

L'idea iniziale era di trovare una soluzione in un tempo non polinomiale ad un problema polinomiale. L'idea è stata scartata, anche sotto suggerimento del relatore, poiché ci si sarebbe approcciati al problema in una maniera eccessivamente ed inutilmente complessa, inoltre è stato dimostrato in [5] che non risulta essere affatto vantaggioso, [16] e [24] affermano invece che sì, risulterebbero vantaggiosi, ma soltanto in particolari situazioni.

Algoritmo bidirezionale con l'utilizzo di thread

Nonostante avessi già implementato l'algoritmo bidirezionale, ho scartato questa idea poiché sarebbe stato inutile paragonarla agli altri algoritmi.

4.2 Possibili Applicazioni

Molti esempi su come sia possibile applicare un algoritmo che lavora su grafi con due pesi sono già stati fatti nel testo, si è parlato di “eco-path”, ovvero la ricerca di cammini minimi in un ambiente dove è importante il risparmio di energia, oppure, si può calcolare un percorso stradale che come valori, invece di considerare la classica distanza e velocità media, considera il tempo medio di percorrenza e il consumo di carburante (e.g. una strada in salita, oppure una strada dove ci si ferma e si riparte molto spesso consuma più carburante di una più lineare, ma più lunga). O ancora, se considero il primo peso come la distanza o il tempo ed il secondo come la pericolosità di un pericolo potrei ottenere diversi percorsi montani con diversi livelli di difficoltà, dove una persona, in base alle sue conoscenze può decidere dove andare. Oltre a questi esempi di pura immediatezza, gli algoritmi analizzati possono essere utilizzati nel campo del ‘*decision making*’, piuttosto che in algoritmi per l’intelligenza artificiale. Nel campo della ricerca operativi, quindi l’ottimizzazione, può essere applicato al problema dei trasporti. Le possibilità sono enormi.

Per approfondire questo aspetto sono consigliati [4] [1] [19]

4.3 Difficoltà incontrate

Le difficoltà principali sono state riscontrate soprattutto nella stesura degli algoritmi in relazione con le mappe che mi sono state fornite (vd. ch5). Inoltre l’implementazione degli algoritmi con minoranti sono stati piuttosto macchinosi e non immediati, seppur il ragionamento dietro il loro sviluppo sia basilare. Nonostante nelle mappe di Parigi e di Berlino conoscessi già alcuni cammini minimi (poiché mi sono stati forniti insieme alle mappe), testare gli algoritmi che lavorano unicamente su grafi non orientati (bidirezionale e minorante invertito) è stato piuttosto complesso. Per verificarli ed effettuare i primi ragionamenti, prima di iniziare a scrivere il codice, ho dovuto fare

diverse congetture e ragionamenti su carta, che mi hanno impegnato non poco tempo.

4.4 Sviluppi futuri

Sarebbe interessante approfondire l'argomento considerando algoritmi che studiano i cammini minimi in grafi con più di due criteri, i cosiddetti “*Multi Objective Shortest Path Problems*” molto studiati in letteratura. Le basi in realtà ci sono già, poiché l'algoritmo di Label-setting può essere facilmente esteso per lo studio in differenti criteri, stesso discorso vale per l'algoritmo bidirezionale. Per quanto riguarda l'algoritmo di Dijkstra applicato a più criteri il discorso cambia, poiché la formula applicabile diverrebbe piuttosto complessa all'aumentare dei criteri per i cammini.

Capitolo 5

Conclusioni

L’obbiettivo era di studiare, paragonare e migliorare alcuni algoritmi per lo studio di percorsi brevi in grafi con più criteri di peso, dopo diverse implementazioni e test è possibile trarre alcune conclusioni: L’algoritmo di Dijkstra applicato a due criteri (sezione 3.1) risulta assolutamente poco performante quando si analizzano piccole porzioni di grafo o grafi di dimensioni modeste, inoltre riescono ad individuare uno scarso numero di risultati. Il discorso cambia quando questo viene applicato ad ambienti di dimensioni più importanti; a questo punto il discorso si complica; tramite l’algoritmo di Dijkstra Iterativo si trovano diverse soluzioni, nonostante necessita di tempi di esecuzione abbastanza elevati, mentre con la ricerca binaria i tempi si accorciano notevolmente (figura 3.4), ma anche il numero dei risultati diminuisce. L’algoritmo di Label-setting rimane piuttosto equilibrato, ma nel momento in cui si vanno analizzare molti nodi il tempo di elaborazione cresce esponenzialmente. I risultati trovati rispecchino sempre il fronte di Pareto delle soluzioni. Entrambe le ottimizzazioni realizzate con i minoranti pregressi non risultano troppo efficaci; infatti nonostante ammortizzino leggermente i tempi quando si trattano grafi di medie dimensioni, sembrano allinearsi, se non peggiorare, le prestazioni dell’algoritmo di label-setting. Ultimo ma non ultimo, l’algoritmo bidirezionale ha concluso tutti i test con tempi ottimi rispetto agli altri algoritmi, sia dove sono presenti pochi nodi da esplorare, sia

quando questi aumentano, il tempo di esecuzione cresce infatti in maniera più lineare rispetto al Label-setting. Il problema dell'algoritmo Bidirezionale sono le soluzioni trovate, queste infatti risultano essere piuttosto scarse rispetto all'algoritmo di Label-setting, ma maggiori in confronto all'algoritmo di Dijkstra.

In sintesi l'algoritmo di Label-setting è il migliore in termini di rapporto tempo/soluzioni in grafi poco consistenti, mentre, quando si va in contro ad un leggermente più corposo numero di nodi, i tempi performati dalle ottimizzazioni con i minoranti risultano essere vantaggiose. Nel momento in cui bisogna trattare reti più importanti e con molti archi, l'algoritmo bidirezionale è un ottimo compromesso, come anche gli algoritmi di Dijkstra, che trovano comunque, in base alle necessità, un sufficiente ‘range’ di soluzioni.

Capitolo 6

Dettagli implementativi

Immagini, codice sorgente e mappe (nodi e archi) possono essere liberamente recuperate dal repository¹. Il linguaggio utilizzato per implementare gli algoritmi è il **Python**. È stata fatta questa scelta per pura praticità; utilizzando altri linguaggi di programmazione di più basso livello (come ad esempio il linguaggio C) si sarebbero potute ottenere prestazioni migliori, ma dato che sono stati fatti dei confronti su diversi risultati la scelta del linguaggio non è stata rilevante.

6.1 Informazioni sui grafi

I grafi delle mappe di Parigi e di Berlino mi sono state gentilmente fornite dal direttore dell'università “Polytech Tours” (Univeristé de Tours, Francia). I punti delle mappe sono numerati e sono su sistema di riferimento spaziale WGS84 (EPGS:4326 - lo stesso utilizzato da Google Maps), mentre gli archi direzionali hanno dei valori casuali.

La mappa della provincia di Bologna e del comune di Cesena sono state invece fornite dall'azienda Optit, che opera a Cesena. Queste mappe sono il risultato di un ritaglio fatto da parte dell'azienda e poi inviatomi tramite diversi file in formato ‘.shp’ (3 file per mappa, un file conteneva gli id degli archi e la loro

¹<https://bitbucket.org/AndreaRossolini/bicriteria-algorithm/>

lunghezza, uno i punti di arrivo, di partenza e se la strada è a senso unico o meno, l'ultimo file conteneva i punti nella mappa, con riferimento spaziale EPSG:32632). Il formato è stato convertito e le coordinate ritradotte per avere lo stesso riferimento con Parigi e Berlino.

Inoltre per effettuare alcuni test sono state generate casualmente dei grafi, è possibile consultare il file ‘.py’ che le gestiva. La mappa di Parigi è orientata, composta da ~ 30000 nodi e ~ 64500 archi; quella di Berlino, anch’essa orientata, è composta da ~ 60000 nodi e ~ 150000 archi; la mappa della provincia di Bologna, invece, è presente sia con archi orientati che non, ed è formata da ~ 40000 nodi e ~ 100000 archi; Cesena, presente in due versioni come Bologna, possiede ~ 6000 nodi e ~ 16000 archi. Ringrazio dunque l’azienda Optit, che si è resa disponibile per permettermi di testare il mio codice sulle mappe da loro fornитomi. I grafi generati in maniera ‘randomica’ vengono descritti dal testo o dalle didascalie quando utilizzati nei test.

6.2 Informazioni sulla macchina utilizzata

Tutti i test sono stati effettuati su di una macchina con le seguenti caratteristiche:

– 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

– Memoria

8192 MB , 2x 4 GB PC3-12800 DDR3L

Ringraziamenti

Prima di tutti ringrazio, ovviamente, i miei genitori che mi hanno, in origine, convinto a fare l'università e mi hanno sostenuto nonostante le mie reazioni emotive del tutto imprevedibili e tutte le necessità economiche del caso.

Oltre a loro ringrazio l'enorme supporto datomi dai miei compagni di corso: dai miei fantastici vicini di casa (galde e coro), che mi hanno fatto ridere ed insieme a loro ho condiviso ansie e speranze; a tutti coloro i quali sono qui oggi per festeggiare con me questo primo traguardo, di una lunga serie.

In particolare ringrazio Filo che mi è stato vicino in questo ultimo periodo e mi ha aiutato a portare avanti un'importante serie di pessime scelte, insegnandomi che se hai diversi piani forse è meglio non farne nessuno. Un ringraziamento importante va anche fatto alle mie amiche di Ravenna che mi hanno fatto capire che se la società ti esclude, in realtà sei soltanto traslato in un'altra società parallela di esclusi. Ovviamente un ringraziamento particolare va a tutti i miei amici di Fano, che, nonostante il mio comportamento snob degli ultimi anni, sono rimasti degli Amici con la "a" maiuscola, sempre disponibili per ogni cosa, in particolare all'Elena, che ha "salvato" il 2019 dall'essere un degli anni più difficili da sopportare. Un ringraziamento particolare va a Edo, che mi ha sempre sostenuto e su cui posso sempre contare; a Franci, punto di riferimento personale e persona dalle mille risorse, all'Ari e all'Illa, che nonostante la loro presenza quasi sovrannaturale sono sempre disponibili.

Grazie babbo (anche se dicevi che non mi sarei riuscito a laureare) e grazie mamma, grazie ai nonni, mastri di vita, molto più utili di quello che magari credono di essere, che, ascoltandoli, mi hanno insegnato molte più cose di quelle che qualsiasi professore di questa università abbia fatto fin'ora.

Bibliografia

- [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. Problèmes multicritères. 1975.