Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Matematica

Analisi delle performance degli algoritmi A* e bidirezionali nel problema del Route Planning

Relatore:
Chiar.ma Prof.ssa
Marilena Barnabei

Correlatore:
Chiar.mo Prof.
Marco Antonio Boschetti

Presentata da:
Tommaso Saracchini

VI Sessione
Anno Accademico 2018-2019

Introduzione

Questa tesi ha lo scopo di analizzare, sotto diversi aspetti, alcuni degli algoritmi più noti che risolvono il problema del Route Planning, ovvero il problema di trovare il cammino più breve tra due punti geografici attraverso una rete stradale.
In particolare esamineremo le performance dell'algoritmo A* con due diverse euristiche, e di due algoritmi di Dijkstra bidirezionali su alcune mappe stradali morfologicamente diverse mettendo in evidenza pregi e difetti di questi.

Nel primo capitolo verranno introdotte le reti di flusso e, in particolare, sarà considerato il problema del flusso di costo minimo, di cui il problema del cammino di costo minimo è un caso particolare.
Nel secondo capitolo introdurremo le basi della teoria del calcolo computazionale utile a costruire un algoritmo e giudicare/misurare la sua efficienza.
Nel terzo capitolo vedremo un'implementazione in linguaggio Python degli algoritmi presi in esame ed infine, nel quarto ed ultimo capitolo, li andremo a testare su alcune città europee.

I dati stradali sono stati ottenuti dal sito open source OpenStreetMap.com, i codici sorgente degli algoritmi si possono trovare alla pagina web https://github.com/SarakkA/Inventions.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Reti di flusso

Siamo circondati da "reti" che consentono lo spostamento di flussi di "commodities" di natura diversa. Basti pensare all'energia elettrica, il gas, l'acqua che viene portata nelle nostre case. Telefoni e computer che si scambiano pacchetti dati permettendoci di comunicare anche oltre la nostra nazione. La rete stradale, ferroviaria, aerea per qualunque tipo di trasporto.
In ognuno di questi esempi vi è sempre un'entità, detta *commodity* (come un messaggio, un veicolo, un prodotto, una persona) che viene trasportata da un punto all'altro attraverso una rete.

I *grafi* modellizzano perfettamente le situazioni appena proposte. Pensiamo ad esempio alla rete stradale di una qualunque città, i *nodi* rappresentano incroci, gli *archi* indicano le varie strade che connettono un incrocio all'altro, il flusso in questo caso può essere un veicolo. Riportiamo nella Tabella 1.1 alcuni esempi tipici su come rappresentare una rete fisica in termini di nodi, archi e flussi.

Nella prossima sezione andremo a descrivere alcune delle molteplici problematiche legate ad una rete di flusso.

Applicazioni	Nodi	Archi	Flusso
Comunicazioni	Telefoni	Cavi	Messaggi
Sistemi di trasporto	Aeroporti, stazioni ferroviarie, incroci stradali	Strade, binari, tratte aeree	Passeggeri, veicoli, merci
Sistemi idraulici	Bacini, stazioni di pompaggio	Tubi	Acqua, gas
Circuiti integrati	Transistor	Collegamenti	Corrente elettrica

Tabella 1.1: Alcune reti fisiche viste come un grafo.

1.1 Problemi sui flussi

Interessanti sono le domande riguardanti una rete, ma quella a cui cercheremo di dare una risposta in questa tesi è la seguente: **qual è il modo più economico di attraversare una rete per andare da un punto ad un altro?**
Questo è noto come il *problema di cammino minimo* e trova applicazioni nei più svariati campi. Nei capitoli 3 e 4 andremo ad analizzarlo a fondo (su reti stradali) sia nel suo aspetto algoritmico che sperimentale andando a vedere, in base a diverse mappe, quale algoritmo risulterà più "performante"[1].
Per una più completa introduzione dell'argomento, andremo prima a definire uno dei problemi fondamentali riguardanti i flussi a costo minimo di cui, come vedremo, il problema del cammino di costo minimo è un caso particolare.

Flusso di costo minimo

Presupponiamo che il lettore abbia familiarità con la struttura di grafo e sue generalità. I concetti essenziali della teoria dei grafi sono comunque richiamati nell'appendice A.

Sia $G = (N, A)$ un grafo semplice, orientato con n nodi e m archi. Ad ogni arco $(i, j) \in A$ è associato:

[1] Lasciamo per ora vago questo termine, lo definiremo più in dettaglio nel capitolo 2.

- un *costo* c_{ij} che sta a rappresentare la spesa da sostenere per unità di flusso su quell'arco (che, per lo scopo di questa tesi considereremo non negativo[2]);
- una *capacità* u_{ij} che denota il flusso massimo che può passare per quell'arco.

Inoltre, ad ogni nodo $i \in N$ associamo un valore $b(i)$ che indica la sua domanda/offerta:

- se $b(i) > 0$ allora il nodo prende il nome di *sorgente*;
- se $b(i) < 0$ lo chiameremo *pozzo*;
- se $b(i) = 0$ è detto *nodo di transito*.

La variabile che rappresenta il flusso sull'arco $(i, j) \in A$ la indichiamo con x_{ij}.

Il problema del flusso di costo minimo, *Minimun Cost Flow Problem* (MCFP), punta a minimizzare il costo complessivo dato da:

$$\sum_{(i,j)\in A} c_{ij}x_{ij}$$

soggetta ai seguenti vincoli:

- $x_{ij} \leq u_{ij}$ per ogni $(i, j) \in A$
- $\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(i,j)\in A} x_{ji} = b(i)$ per ogni $i \in N$

Quest'ultimo vincolo è noto come *legge di conservazione del flusso*, in cui la prima sommatoria indica il flusso uscente dal nodo $i \in N$, la seconda sommatoria denota il flusso entrante in esso.
In particolare, se prendiamo un nodo di transito il flusso entrante eguaglia il flusso uscente, se consideriamo una sorgente (un pozzo) la differenza delle sommatorie eguaglia la sua offerta (domanda).

[2]Il vincolo di non negatività degli archi è un vincolo molto forte, infatti, senza di questo nessuno degli algoritmi presentati nel terzo capitolo funzionerebbe.

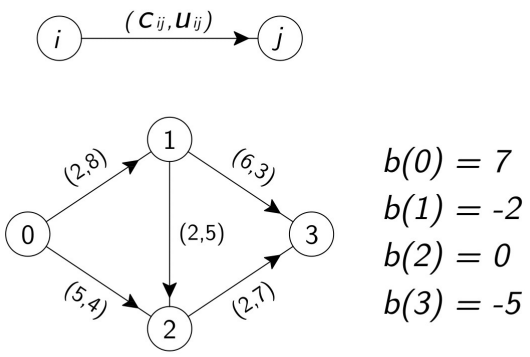


Figura 1.1: Una rete di flusso.

Cammino minimo

Come già accennato, in questo problema vogliamo trovare un percorso di costo minimo da una sorgente ad un pozzo.
Possiamo vederlo come caso particolare del problema precedente, in cui poniamo $b(s) = 1$, $b(t) = -1$ e $b(i) = 0$ per ogni altro nodo i del grafo, trascurando il vincolo di capacità. La soluzione del MCFP manda una sola unità di flusso dalla sorgente al pozzo attraverso il cammino di costo minimo.

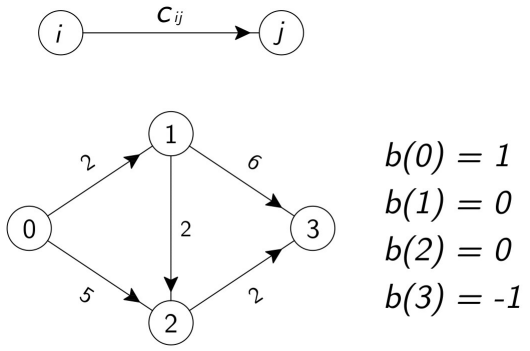


Figura 1.2: Una rete di flusso senza capacità.

Formulando opportunamente il MCFP, possiamo anche risolvere il problema di trovare i percorsi più brevi dal nodo sorgente s a qualunque altro

vertice del grafo. Poniamo $b(s) = n - 1$ e $b(i) = -1$ per ogni altro nodo. La soluzione del MCFP manda una unità di flusso ad ogni altro nodo lungo il cammino più breve.

Altri Problemi

Data la loro importanza, vale la pena descrivere altri problemi come casi particolari del MCFP, che però non tratteremo in questo testo. Per il lettore interessato rimandiamo a [1].

Problema di circolazione: come può suggerire il nome, qui abbiamo solo nodi di transito, cioè $b(i) = 0$ per ogni $i \in N$.
Ci chiediamo, fissata una quantità di flusso nella rete, (che non può né diminuire né aumentare a causa della mancanza di sorgenti/pozzi) qual è la circolazione di costo minimo.
Un tipico esempio di applicazione riguarda la pianificazione di tratte aeree. in questo caso può essere interessante (e utile) non solo considerare un limite superiore per la capacità di ogni arco, ma anche un limite inferiore l_{ij} che permetta alla compagnia aerea di assicurare un servizio minimo per i suoi clienti. Chiaramente questo andrà a modificare il primo vincolo in MCFP in questo modo: $l_{ij} \leq x_{ij} \leq u_{ij}$

Problema di flusso massimo: questo problema può essere visto come complementare a quello di cammino minimo. Infatti in quest'ultimo abbiamo un costo, senza vincoli di capacità; al contrario, nel problema di flusso massimo non consideriamo costi ma prendiamo in considerazione le capacità. Qui vogliamo trovare la massima quantità di flusso che si può trasportare da una sorgente s ad un pozzo t. Formuliamo il problema come caso particolare di MCFP: poniamo $b(i) = 0$ per ogni $i \in N$, $c_{ij} = 0$ per ogni $(i, j) \in A$ e costruiamo un nuovo arco (t, s) con costo $c_{ts} = -1$ di capacità $u_{ts} = \infty$.
Con questa costruzione la soluzione di MCFP massimizza il flusso nell'ar-

co (t, s) e, dato che questo flusso deve viaggiare dalla sorgente al pozzo attraverso gli archi in A, massimizza il flusso nella rete originale.

1.2 Rappresentazione di un flusso sul calcolatore

L'efficienza di un algoritmo non dipende solamente dalla sua struttura, ma anche dal modo con cui immagazziniamo, estraiamo e aggiorniamo i dati relativi alla rete di flusso.
Generalmente, le informazioni di una rete sono di due tipi:

1. *Topologia della rete*, vale a dire il numero di nodi e archi e loro relazione;
2. *Dati associati*, cioè costi, capacità, domanda/offerta.

In questa sezione vediamo due modi di rappresentare una rete di flusso al calcolatore.

Matrice di adiacenza nodo-nodo

Con questo metodo memorizziamo i dati in una matrice M di dimensione $n \times n$ in cui vi è una colonna ed una riga per ogni nodo del grafo. Il generico elemento a_{ij} è uguale a 1 se $(i, j) \in A$, 0 altrimenti (Figura 1.3).

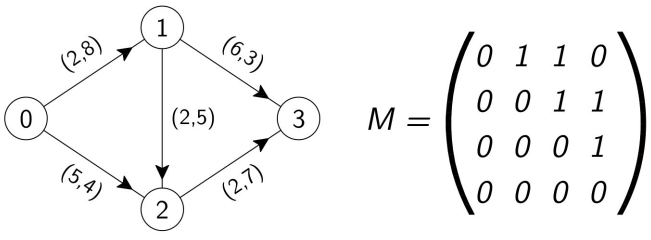


Figura 1.3: Il grafo di Figura 1.1 e sua matrice di adiacenza.

Possiamo fare alcune osservazioni:

- Dato che la matrice M immagazzina solo dati legati alla topologia del grafo, abbiamo bisogno di altre matrici delle stesse dimensioni per memorizzare anche costi e capacità;
- Con questo metodo è possibile accedere molto velocemente alle informazioni (ad esempio, verificare se $(i, j) \in A$);
- Tra i kn^2 elementi della matrice (k è il numero di matrici da memorizzare), solo km sono diversi da zero e questo metodo potrebbe risultare poco efficiente per rappresentare grafi sparsi.

Liste di adiacenza

Definiamo la lista di adiacenza $A(i)$ del nodo i come quell'insieme dei nodi j tali che $(i, j) \in A$. A livello implementativo possiamo pensare $A(i)$ come un array contenente gli indici dei nodi adiacenti al nodo i.
Memorizzando le liste di adiacenza di ogni nodo abbiamo informazioni riguardanti la topologia del grafo. Per memorizzare anche costi e capacità possiamo procedere nel seguente modo: invece di memorizzare nell'array $A(i)$ solo il valore corrispondente all'indice del nodo adiacente, memorizziamo una lista con un numero variabile di *campi* in base a quante informazioni vogliamo memorizzare. Un campo è sempre occupato dal nodo adiacente j, possiamo poi aggiungere un campo per il costo e un campo per la capacità come mostrato in Figura 1.4.

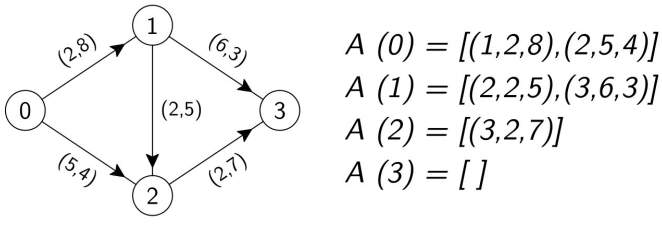


Figura 1.4: Il grafo di Figura 1.1 e sue liste di adiacenza.

Capitolo 2

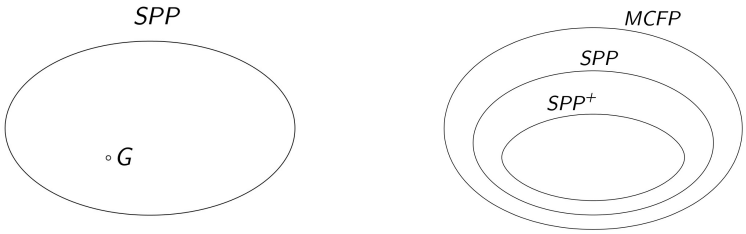
Algoritmi e complessità computazionale

In questo capitolo ci occuperemo di definire il concetto di efficienza per un algoritmo, che fino ad ora abbiamo lasciato vago, e andremo ad esporre alcuni dei metodi più utilizzati per confrontare algoritmi.

Un *algoritmo* è una serie di istruzioni da eseguire per risolvere un problema. Una *istanza* è un caso specifico del problema P con parametri fissati, ad esempio, il grafo di Figura 1.2 è un'istanza del problema di cammino minimo. Diciamo che un algoritmo risolve un problema P se, per ogni istanza di P, esso restituisce una soluzione.

In quest'ottica, formalizziamo inoltre quanto detto nel capitolo 1, cioè che il problema di cammino minimo (abbreviato SPP, dall'inglese *Shortest Path Problem*) è un caso particolare del MCFP. Nel senso che un algoritmo che risolve MCFP, applicato ad un'istanza di SPP, risolve SPP.

La Figura 2.1 dà una rappresentazione insiemistica di quanto appena detto in particolare in (a) il grafo di Figura 1.2 è visto come elemento dell'insieme SPP, mentre in (b) abbiamo che l'insieme SPP^+ (cioè il problema di cammino minimo a costi non negativi) è visto come sottoinsieme (cioè come caso



(a) Il grafo di Figura 1.2 come istanza di SPP

(b) $\text{SPP}^+ \subset \text{SPP} \subset \text{MCFP}$

Figura 2.1: Visione insiemistica dei problemi P.

particolare) del SPP che a sua volta, è sottoinsieme (un caso particolare) del MCFP.

2.1 Analisi della complessità computazionale di un algoritmo

Il tempo di esecuzione di un algoritmo sembra essere una ragionevole metrica per misurare la sua efficienza.
Tipicamente un algoritmo esegue operazioni elementari come assegnazione, operazioni aritmetiche e operazioni logiche. La somma di queste, divise per il numero di operazioni al secondo che una data macchina può eseguire (che d'ora in avanti indicheremo con k), determina il suo tempo di esecuzione.
Vediamo ora due metodi per esaminare l'efficienza di un algoritmo:

- *analisi empirica*: valuta l'algoritmo testandolo su alcune istanze significative. Ci dà una stima del tempo di esecuzione "in pratica" (data un'istanza di P, ci aspettiamo che questo impieghi circa x secondi);

- *analisi worst-case*: come suggerisce il nome, valuta l'algoritmo in base alla sua peggior performance. Non ci dà una stima, bensì un limite superiore del suo tempo di esecuzione (data un'istanza di P, questo impiegherà al massimo x secondi).

Entrambi hanno dei pro e dei contro. Per quanto riguarda l'analisi empirica, uno stesso algoritmo eseguito su classi di istanze diverse produce tempistiche diverse, questo può portare a risultati contraddittori e a stimare in maniera "soggettiva" l'algoritmo. Il limite worst-case risulta più facile da determinare rispetto alla stima empirica, però è possibile che alcune istanze "patologiche" causino l'innalzamento del limite superiore peggiorando la stima della sua efficienza sotto questa analisi.
Ad ogni modo, la worst-case analisys è conclusiva per stabilire la performance in generale per qualsiasi possibile istanza. Proprio per questo fattore "oggettivo" tale analisi è quella più utilizzata dalla comunità scientifica.
In questa tesi, in particolare nei capitoli 3 e 4, ci serviremo di entrambe le analisi. L'analisi worst-case ci servirà per scegliere gli algoritmi che, a priori, possiedono un buon limite worst-case; una volta selezionati gli algoritmi, li andremo a testare su classi di istanze diverse, per vedere la loro performance "in pratica".

2.1.1 Analisi Worst-Case

Finora abbiamo stabilito cosa intendiamo per problema, che cos'è un'istanza di un problema e come valutare un algoritmo in due diversi modi, ma non abbiamo mai fatto riferimento alla *grandezza* di un'istanza.
Per *grandezza*, in generale, intendiamo la quantità di dati in input che diamo all'algoritmo e che questo deve processare al fine di restituire una soluzione. In particolare, nel caso dei problemi di flusso, la *grandezza* fa riferimento al numero n di nodi e al numero m di archi. Ci aspettiamo che maggiore sia la quantità di informazioni, maggiore sia il tempo di esecuzione.

Definiamo quindi la *funzione di complessità* per un algoritmo (o *complessità di un algoritmo*) come quella funzione che restituisce il tempo massimo di esecuzione che esso impiega per risolvere qualunque istanza di quella grandezza.
Supponiamo che la funzione di complessità di un algoritmo, per un problema

di flusso, sia $f(m,n) = \frac{mn}{k}$. Allora il tempo di esecuzione per risolvere una qualunque istanza di P con n nodi ed m archi è di al massimo $\frac{mn}{k}$ secondi.

Notazione O-grande

La notazione O-grande è utilizzata per descrivere il comportamento asintotico delle funzioni, più precisamente per descrivere un *limite asintotico superiore* della funzione di complessità. Data la natura dell'analisi worst-case, questa fa al caso nostro.

Definizione 1. Siano $f(n)$ e $g(n)$ due funzioni positive. Abbiamo che $f(n) = O(g(n))$ se esistono due costanti c e n_0 positive tali che $f(n) < cg(n)$ per ogni $n > n_0$.

A parole, abbiamo che $g(n)$ è un limite asintotico superiore di $f(n)$, e se quest'ultima corrisponde ad una funzione di complessità, diciamo allora che *la complessità dell'algoritmo è dell'ordine di* $O(g(n))$.

Supponiamo di avere due algoritmi, A e B, le cui rispettive funzioni di complessità siano $f_A(n) = \frac{1}{k}(n+1)^2$ e $f_B = \frac{1}{k}(3n^2+n)$, con k il numero di operazioni al secondo di una data macchina.
Visto che sono funzioni diverse, queste (chiaramente) ci daranno tempi diversi una volta fissata un'istanza del problema, ma quello che ci interessa realmente è considerare la *crescita asintotica* delle funzioni di complessità.
Si verifica facilmente che $f_A(n) = O(n^2)$ e che $f_B(n) = O(n^2)$ e ciò significa che, in questa notazione, non vi è distinzione tra i due algoritmi, essi si comportano asintoticamente alla stessa maniera.
Se invece dovessimo scegliere tra un algoritmo C con complessità $f_C(n) = \frac{1}{k}(3n^3 - 2n)$ cui risulta essere dell'ordine di $O(n^3)$ e l'algoritmo A, opteremmo per quest'ultimo in quanto la complessità asintotica è minore.

Dagli esempi appena mostrati possiamo osservare due proprietà fondamentali della notazione O-grande:

- **Ci consente di ignorare le costanti** sconosciute dovute al compilatore e alla macchina, ma anche di formulare alcune ipotesi semplificative. Inizialmente abbiamo definito le operazioni elementari come assegnazione, operazioni aritmetiche e logiche assumendo implicitamente che l'algoritmo le esegua in egual tempo, ma questo non è per niente vero. Ad ogni modo, queste variazioni di velocità possono essere limitate da una costante moliplicativa, che in questa notazione sono irrilevanti.

- **Indica solo il termine dominante** della funzione di complessità ignorando termini con basso tasso di crescita che risulterebbero, per n grande abbastanza, insignificanti rispetto a quelli con tasso di crescita più elevato.

Ora, finalmente, possediamo gli strumenti per comparare algoritmi su di uno stesso problema facendo riferimento all'ordine della funzione di complessità. Ma come stabiliamo se un algoritmo è, di per sé, efficiente?
Un'idea che ha preso piede universalmente è la seguente: un algoritmo è considerato efficiente se ha *complessità polinomiale* e cioè se $f(n) = O(p(n))$ con $p(n)$ polinomio di grado qualsiasi.

n	$\log n$	$\sqrt{n}$	n^2	n^3	2^n
10	3.32	3.16	100	10^3	10^3
100	6.64	10	10^4	10^6	1.27×10^{30}
1'000	9.97	31.62	10^6	10^9	1.07×10^{301}
10'000	13.29	100	10^8	10^{12}	0.99×10^{3010}

Tabella 2.1: Tasso di crescita delle funzioni più comuni.

Ci sono molteplici motivazioni per preferire un algoritmo con complessità polinomiale, una delle quali è che siamo interessati maggiormente al comportamento degli algoritmi con input molto grandi.

Come vediamo dalla Tabella 2.1, la funzione esponenziale ha una crescita "esplosiva", di conseguenza un algoritmo con quest'ordine di complessità farebbe fatica a restituire una soluzione perfino per input relativamente piccoli.

Capitolo 3

Shortest path problem

In questo capitolo andremo ad esporre alcuni degli algoritmi più noti per risolvere il problema di cammino minimo, analizzandoli sotto vari aspetti. In particolare li utilizzeremo per trovare il cammino più breve tra due punti geografici attraverso una rete stradale (in letteratura viene chiamato problema del "Route Planning"). Prima di definire nel dettaglio il problema, è opportuno fare alcune considerazioni sul tipo di grafo che modellizza questa situazione.

3.1 Definizione e vincoli nel route planning

Sebbene l'analisi degli algoritmi che seguirà sia applicabile a diversi scenari, in questa tesi vengono considerati solo grafi associati a reti stradali. Questo ci porta a fare la seguente assunzione:

Il grafo di una rete stradale è *orientato*, *connesso*, *sparso*, con *costi positivi*.

Orientato perché è bene tenere in considerazione strade a senso unico, soprattutto in centro città; *connesso* perché assumiamo l'esistenza di un

cammino per ogni coppia di nodi; con *costi positivi* perché questi potrebbero rappresentare lunghezze, tempi di percorrenza o una loro combinazione.
Di notevole importanza è il fatto di avere a che fare con un grafo *sparso*. In questa situazione prediligiamo la rappresentazione per liste di adiacenza piuttosto che attraverso una matrice, risparmiando un'enorme quantità di memoria. Inoltre, i risultati empirici in [7] mostrano che, per una generica mappa urbana, il massimo numero di archi incidenti ad un nodo (sia uscenti che entranti) è di circa 6. Possiamo quindi porre un limite superiore al numero di archi incidenti ad un generico nodo che sia indipendente dal numero n di nodi del grafo: questo ci permette di abbassare il limite superiore relativo al numero degli archi da $m = O(n^2)$ (per un grafo generico), a $m = O(n)$ con n numero di nodi e m numero degli archi. Definiamo ora il problema:

Sia $G = (N, A)$ un grafo orientato con costi $c_{ij} \geq 0$ per ogni arco $(i, j) \in A$. Chiamiamo s il nodo sorgente e t il nodo destinazione. Vogliamo minimizzare la quantità

$$\sum_{(i,j)\in A} c_{ij} x_{ij}$$

Soggetta ai seguenti vincoli:

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(i,j)\in A} x_{ji} = \begin{cases} 1 & \text{se } i = s \\ -1 & \text{se } i = t \\ 0 & \text{altrimenti} \end{cases}$$

$$x_{ij} \geq 0$$

Il vincolo di non negatività del flusso e la legge di conservazione del flusso garantiscono che la soluzione, e cioè la matrice X in cui il generico elemento x_{ij} rappresenta la quantità di flusso che passa nell'arco (i, j), abbia solo valori interi non negativi[1], che in questo caso possono essere solo binari:

[1]Questo perché la matrice dei coefficienti dei vincoli di conservazione del flusso è totalmente unimodulare. Il lettore interessato può approfondire in [1].

$$x_{ij} = \begin{cases} 1 & \text{se } (i,j) \text{ appartiene al cammino minimo} \\ 0 & \text{altrimenti} \end{cases}$$

Questo problema, così come è stato formulato, è chiamato *single-pair*[2] cioè si preoccupa di trovare il cammino minimo tra una coppia di nodi fissati, ed è quello per cui vogliamo trovare un metodo di risoluzione efficiente.
Andiamo comunque a distinguere alcune varianti la cui teoria ci servirà in seguito:

- *Single-source*: trovare il cammino minimo da un dato nodo ad ogni altro vertice del grafo;
- *Single-target*: trovare il cammino minimo che raggiunge un nodo fissato partendo da ogni altro vertice del grafo (può essere visto come un single-source invertendo la direzione degli archi);
- *All-pair*: trovare il cammino minimo tra ogni coppia di vertici (di cui non ci occuperemo in questa tesi).

3.2 Algoritmi

In questa sezione andremo ad esporre nel dettaglio gli algoritmi che permettono di risolvere il problema di cammino minimo single-pair. Ad ogni modo, visto che ci servirà in seguito, presenteremo inizialmente l'algoritmo sviluppato da Dijkstra che risolve il problema single-source. Il motivo per cui partiamo da questo è che tutti gli algoritmi che seguiranno sono costruiti sulla base della sua struttura.

Strutture Dati

Prima di tuffarci negli algoritmi, è bene specificare il tipo di strutture dati che utilizziamo per una più facile comprensione del codice[3].

[2] Route planning se applicato a reti stradali.
[3] Tutti gli algoritmi in questa tesi sono implementati in linguaggio Python.

- Rappresentiamo il grafo con un *dizionario*, in Python un dizionario è una struttura dati che mette in memoria coppie del tipo

  {"chiave" : "valore"}

  Questo ci permette di implementare la rappresentazione per liste di adiacenza vista nel capitolo 1.

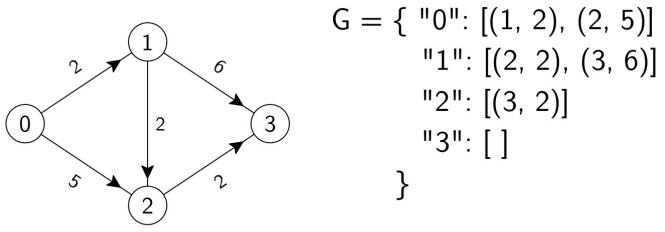


Figura 3.1: Il grafo di Figura 1.2 e relativo dizionario.

  In questa rappresentazione le n "chiavi" del dizionario sono i nodi del grafo, i rispettivi "valori" sono array contenenti tuple del tipo (nodo adiacente, costo);

- utilizziamo due ulteriori dizionari, uno chiamato *distance* per immagazzinare i costi di ogni nodo, l'altro chiamato *prev* per ricostruire il cammino;

- ci serviamo di un array, chiamato *visited*, per tener conto dei nodi già visitati.

3.2.1 Dijkstra, single-source

L'algoritmo di Dijkstra è un algoritmo di visita di un grafo, che permette di trovare il cammino minimo da un dato vertice (sorgente) a tutti gli altri vertici del grafo.
Per ogni nodo i, l'algoritmo tiene traccia di un valore $d(i)$ che indica il costo

da sostenere per raggiungere i.
Ad ogni passo, l'algoritmo divide l'insieme dei vertici in due gruppi, quelli con *distanza permanente* (i nodi già visitati) e quelli con *distanza provvisoria* (i nodi non ancora visitati). Il valore $d(i)$ di un nodo con distanza permanente rappresenta il costo minimo per raggiungere quel nodo, il valore $d(i)$ di un nodo con distanza provvisoria rappresenta un limite superiore del costo per raggiungere quel nodo.
L'idea dell'algoritmo è di visitare il grafo, partendo dalla sorgente, ed assegnare mano a mano una distanza permanente ad ogni nodo. L'ordine di visita dei nodi dipende dal valore $d(i)$ assegnato, vale a dire, diamo priorità ai nodi con distanza provvisoria minore (sarà proprio questa la condizione che garantisce l'ottimalità della soluzione, come dimostrato più avanti dal Teorema 2).
Esso termina quando tutti i vertici vengono visitati, ovvero quando ad ogni nodo viene assegnata una distanza permanente[4].
Inoltre, l'algoritmo tiene traccia di un'etichetta $pred(i)$ che indica il nodo che precede i nel cammino minimino che porta a tale nodo. Così facendo, è possibile ricostruire un *albero*, con radice la sorgente, che ricopre l'intero grafo di partenza e che dà informazioni sul cammino minimo di ogni nodo.

Definiamo ora la funzione `scelta_del_nodo(dict, F)` che prende in input il dizionario *distance* e l'array `F` dei nodi non ancora visitati. Restituisce il nodo (tra quelli non esplorati) con distanza minima.

```
1 def scelta_del_nodo(dict, F):
2     costo_minimo = 999999
3     nodo_con_costo_minimo = None
4
```

[4]Di seguito, utilizzeremo la dicitura "nodo visitato", "nodo espanso" o "nodo esplorato", per sottintendere un nodo a cui è stata assegnata una distanza permanente dall'algoritmo.

```
5       for i in dict:
6           if dict[i] < costo_minimo and i in F:
7               costo_minimo = dict[i]
8               nodo_con_costo_minimo = i
9   return nodo_con_costo_minimo
```

Quindi vediamo in dettaglio un'implementazione dell'algoritmo di Dijkstra, andando ad analizzarla a blocchi.
Nella prima parte inizializziamo le strutture dati che ci serviranno, quindi, il dizionario `distance` in cui, per ogni nodo, assegniamo un valore infinito. Il dizionario `prev` in cui, per ogni nodo, assegniamo None. Poniamo quindi la distanza (permanente) della sorgente uguale a zero e creiamo due array uno per i nodi visitati (inizialmente vuoto), uno per i nodi ancora da visitare (con all'interno gli n nodi del grafo).

```
1   def Dijkstra_ss(graph, source):
2       distance = {}
3       prev = {}
4       for i in nodes:
5           distance[i] = 999999
6           prev[i] = None
7       distance[source] = 0
8       visited = []
9       not_visited = nodes
```

Da qui comincia il ciclo di visita del grafo e termina non appena ogni nodo viene visitato. Prendiamo quindi il nodo con distanza minima con la funzione `scelta_del_nodo(distance, not_visited)`, lo mettiamo tra quelli visitati e lo togliamo da quelli non visitati

```
10  while len(not_visited) > 0:
11      curr_node = scelta_del_nodo(distance, not_visited)
```

```
12      visited.append(curr_node)
13      not_visited.remove(curr_node)
```

Una volta effettuata la scelta del nodo (chiamato `curr_node`), accediamo alla sua lista di adiacenza (riga 14) e per ogni nodo adiacente (nel codice `neighbor`) verifichiamo l'esistenza di un cammino più breve. In caso affermativo (riga 16-17) aggiorniamo il valore `distance[neighbor]` e il nodo `prev[neighbor]`

```
14  for neighbor, cost in graph[curr_node]:
15      if distance[curr_node] + cost < distance[neighbor
           ]:
16          distance[neighbor]= distance[curr_node] + cost
17          prev[neighbor]= curr_node
```

Qui si conclude la visita del nodo `curr_node` e si passa a quello successivo (si ritorna alla riga 10).

Correttezza

Teorema 2. *Le distanze permanenti assegnate dall'algoritmo di Dijkstra ad ogni nodo del grafo sono ottimali.*

Dimostrazione. Sia $G = (N, A)$ un grafo, sia S l'insieme dei nodi visitati fino ad una generica iterazione, sia $\bar{S} = N - S$ l'insieme dei nodi non visitati, e $L = \{(i, j) \in A : i \in S, j \in \bar{S}\}$ l'insieme degli archi uscenti dal "taglio" tra S e $\bar{S}$.
Dimostriamo il teorema per induzione sulla cardinalità dell'insieme dei nodi visitati.
All'inizio, l'insieme S contiene solo il nodo sorgente s, quindi la tesi del teorema risulta banalmente verificata.
Ad una generica iterazione, ipotizziamo induttivamente che per ogni nodo $i \in S$ l'etichetta permanente $d(i)$ sia ottimale. Sia $q \in \bar{S}$ il nodo da espandere

all'iterazione successiva, supponiamo quindi:

$$d(p) + c_{pq} = \min_{(i,j)\in L} \{d(i) + c_{ij}\} \tag{3.1}$$

Abbiamo quindi che $d(p) + c_{pq}$ è l'etichetta permanente assegnata al nodo q dall'algoritmo.
Sia P un qualunque cammino da s a q, per dimostrare l'ottimalità dell'etichetta appena assegnata, basterà far vedere che il suo costo $c(P)$ sia maggiore o uguale a $d(p) + c_{pq}$.

Dato che $s \in S$ e $q \in \bar{S}$ esisterà in P un arco $(i, j) \in L$. Il costo del cammino sarà dunque $c(P) = c(P_1) + c_{ij} + c(P_2)$ con P_1 il sottocammino da s a i, e P_2 il sottocammino da j a q.
Per ipotesi induttiva abbiamo che $c(P_1) \geq d(i)$, e per il vincolo di non negatività degli archi abbiamo che $c(P_2) \geq 0$. Di conseguenza

$$c(P) \geq d(i) + c_{ij}$$

Utilizzando infine l'equazione (3.1) abbiamo che $c(P) \geq d(p) + c_{pq}$. □

In conclusione, dal teorema deriva la correttezza dell'algoritmo di Dijkstra in quanto la distanza permanente assegnata ad ogni nodo corrisponde al costo del cammino minimo.

Tempo di esecuzione

Possiamo ridurre l'analisi dell'algoritmo alle seguenti due operazioni.

- *Scelta del nodo*: ad ogni iterazione, l'algoritmo cerca il nodo di costo minore dall'insieme dei vertici non visitati, impiegando un tempo $O(n)$. Quest'operazione è eseguita per n volte quindi il tempo totale impiegato è dell'ordine di $O(n^2)$.

- *Aggiornamento distanze*: per ogni nodo i l'algoritmo va ad aggiornare (nel peggiore dei casi) la distanza di ogni nodo a lui adiacente per un

totale di $|A(i)|$ volte, con $A(i)$ lista di adiacenza di i. Quest'operazione è eseguita n volte per un totale di $\sum_{i \in N} |A(i)| = m$. Dato che ogni aggiornamento viene eseguito in un tempo costante $O(1)$, il tempo totale impiegato è dell'ordine di $O(m)$.

L'algoritmo impiega quindi un tempo $= O(n^2 + m)$. Considerando la limitazione relativa al numero m di archi fatta nella sezione precedente, possiamo affermare che l'algoritmo di Dijkstra risolve il problema di cammino minimo single-source in un tempo $O(n^2)$.

3.2.2 Dijkstra, single-pair

L'idea che sta alla base degli algoritmi che risolvono il problema single-pair è quella di non andare a visitare tutti i possibili nodi del grafo ma solamente quelli "necessari" per produrre un cammino minimo. La scelta di questi nodi e la *stopping condition* di questi algoritmi costituirà il punto cruciale per determinare la loro efficienza.

Vediamo ora una soluzione "ingenua" per risolvere il single-pair sulla base dell'algoritmo appena esposto.
Innanzitutto (riga 1) la funzione avrà bisogno di un terzo input che indica il nodo destinazione (nel codice `target`). Per il resto, l'inizializzazione delle variabili rimane invariata.

```
1  def Dijkstra_sp(graph, source, target):
2      distance = {}
3      prev = {}
4      for i in nodes:
5          distance[i] = 999999
6          prev[i] = None
7      distance[source] = 0
8      visited = []
```

```
9       not_visited = nodes
```

Quello che cambia rispetto al single-source è la stopping condition (riga 10), vale a dire, l'algoritmo termina non appena il nodo destinazione viene visitato.

```
10  while target not in visited:
11      curr_node = scelta_del_nodo(distance, not_visited
            )
12      visited.append(curr_node)
13      not_visited.remove(curr_node)
14      for neighbor, cost in graph[curr_node]:
15          if distance[curr_node] + cost < distance[
                neighbor]:
16              distance[neighbor]= distance[curr_node] +
                    cost
17              prev[neighbor]= curr_node
```

Qui termina la visita del nodo `curr_node` e si passa al prossimo.
Il seguente codice estrae informazioni dal dizionario *prev* per ricostruire il percorso dal nodo `source` al nodo `target`. Restituisce l'array dei nodi che compongono il cammino trovato.

```
18  rev_path = [target]
19  k = 0
20  while source not in rev_path:
21      rev_path.append(prev[rev_path[k]])
22      k = k + 1
23      path = rev_path[::-1]
24  return path
```

Osservazioni

Chiaramente la correttezza della soluzione di quest'ultimo algoritmo è assicurata dal teorema precedente, in quanto l'algoritmo si interrompe non appena assegniamo una distanza permanente al nodo `target`.
Inoltre è interessante osservare quanti e quali nodi sono stati visitati dall'algoritmo. La Figura 3.2 dà un'idea di come l'algoritmo di Dijkstra si espande attraverso il grafo.

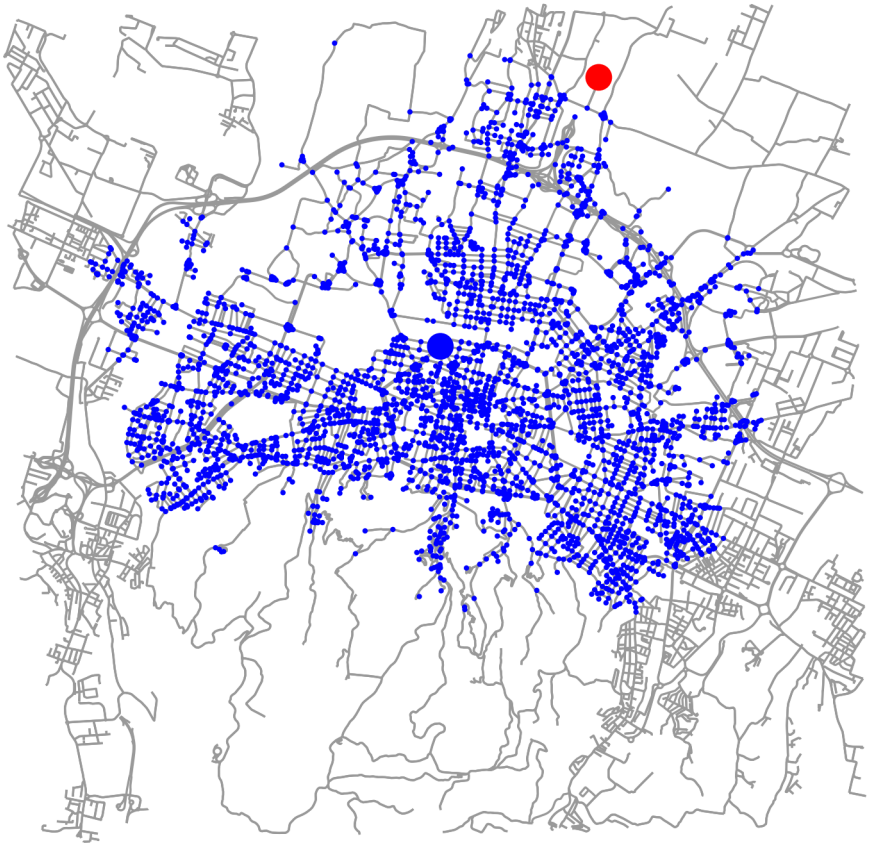


Figura 3.2: Algoritmo di Dijkstra eseguito sulla mappa di Bologna.

I nodi grandi in blu e in rosso sono rispettivamente sorgente e destinazione, i nodi blu in piccolo sono i nodi visitati per produrre un cammino minimo.

Tempo di esecuzione

Per quanto riguarda il tempo di esecuzione l'ordine della funzione di complessità non cambia rispetto al single-source, questo perché c'è la possibilità che il nodo destinazione sia l'ultimo visitato e quindi rimane $O(n^2)$.
Sperimentalmente invece possiamo considerarlo più veloce nel restituire la soluzione perché ci aspettiamo che non sia necessario visitare tutti i nodi.

3.2.3 Heap

La complessità computazionale dell'algoritmo risiede maggiormente nella scelta del nodo con distanza minima. Qui presentiamo una struttura dati che ci permetterà di ridurre la complessità worst-case di questa operazione. Inoltre, data la sua efficienza, la utilizzeremo anche negli algoritmi a seguire.

Una *heap* è una struttura dati ad albero che mantiene ordinati i nodi rispetto a un certo valore che gli viene assegnato mentre vengono aggiunti. Questo comporta due vantaggi:

- la scelta del nodo è ristretta ai soli elementi nella heap, e non su tutto l'array dei nodi non visitati;
- ogni volta che inseriamo un elemento nella heap, questo viene posizionato in base alla priorità assegnata (nel nostro caso in base alla distanza). Tenendo la struttura "ordinata" in questo modo, la scelta dal nodo verrà eseguita in un tempo nettamente minore.

Per costruire questa struttura in Python ci serviamo del pacchetto *heapq*. Vediamo ora l'algoritmo.

L'inizializzazione è sempre la stessa, a parte il fatto che la nuova struttura (chiamata `q` nel codice) rimpiazza l'array `not_visited`.
In particolare, la heap è rappresentata come un array con elementi del tipo

`(distance[i] , i)`. Questi saranno ordinati in base al valore `distance[i]`. Inizializziamo la heap inserendo il nodo sorgente con distanza zero.

```
1  def HeapDijkstra(graph, source, target):
2      distance = {}
3      prev = {}
4      for i in nodes:
5          distance[i] = 999999
6          prev[i] = None
7      distance[source] = 0
8      visited = []
9      q = [(0, source)]
```

Ad ogni iterazione prendiamo dalla heap il nodo con costo minimo con la funzione `heappop()` e lo mettiamo nell'array visited.

```
10 while target not in visited:
11     curr_dist, curr_node = heappop(q)
12     if curr_node not in visited:
13         visited.append(curr_node)
```

Accediamo ora alla lista di adiacenza del nodo scelto e, se necessario, aggiorniamo le distanze. In tal caso, attraverso la funzione `heappush()` aggiungiamo nuovi nodi alla heap mantenendo l'ordine di priorità.

```
14 for neighbor, cost in graph[curr_node]:
15     if curr_dist + cost < distance[neighbor]:
16         distance[neighbor]= curr_dist + cost
17         prev[neighbor]= curr_node
18         heappush(q, (distance[neighbor], neighbor))
```

Il codice che restituisce l'array `path` rimane lo stesso come in `Dijkstra_sp` , quindi lo omettiamo.

Osservazioni

In realtà l'implementazione corretta della struttura heap, prevede (oltre alle funzioni `heappop()` e `heappush()`) la funzione *descease-key* che permette di modificare la priorità di un nodo che è già all'interno della heap (cioè nel caso in cui effettuiamo un aggiornamento su di un nodo nella heap).
L'implementazione appena mostrata non utilizza questa funzione per due motivi:

- non è presente nel pacchetto heapq;
- è possibile aggirare il problema senza che questo vada ad incidere sulla complessità worst-case dell'algoritmo.

In questa implementazione, invece di aggiornare la priorità del nodo nella heap, viene aggiunto un altro nodo, con lo stesso nome ma con maggiore priorità. Alla fine dell'iterazione ci troviamo quindi elementi doppi (o più di due in base a quante volte effettuiamo l'aggiornamento su di un particolare nodo) con diversa priorità.
Per evitare di visitare nuovamente i nodi con i valori "obsoleti" abbiamo aggiunto la riga 12 che a prima vista può apparire superflua, ma in realtà serve proprio per controllare questo problema.

Una cosa importante da notare è che questa struttura va a modificare l'architettura dell'algoritmo ma non il suo comportamento, vale a dire che il numero di nodi visitati e l'ordine con il quale vengono visitati è lo stesso dell'algoritmo `Dijkstra_sp()`. Se andassimo a evidenziare sulla mappa i nodi visitati dall'algoritmo `HeapDijkstra()` il risultato sarebbe identico a quello di Figura 3.2.

Tempo di esecuzione

- heappop(): estrae il nodo con costo minimo in tempo costante, però bisogna ricostruire l'ordine nella struttura, impiegando un tempo $O(\log(n))$. Questa operazione viene eseguita al massimo n volte quindi il tempo totale impiegato è dell'ordine di $O(n \log(n))$.

- heappush(): aggiunge un elemento alla heap posizionandolo in base alla priorità a lui assegnata in un tempo $O(\log(n))$. Questa viene eseguita al massimo m volte, quindi il tempo totale è dell'ordine di $O(m \log(n))$.

In conclusione, dato che per le reti stradali possiamo ipotizzare che $m = O(n)$, l'algoritmo impiega $O(n \log(n))$.

3.2.4 A*

L'algoritmo di Dijkstra precedentemente implementato è detto *non informato* e *ammissibile*. Non informato perché, non conoscendo la posizione del nodo destinazione, effettua una ricerca "alla cieca" espandendosi in tutte le direzioni fino a che questo non viene trovato; ammissibile perché assicura che la soluzione trovata sia effettivamente quella migliore, ovvero di costo minimo.
Sebbene la complessità dell'algoritmo `HeapDijkstra()` sia da considerare "ottimale", questo, come possiamo vedere dalla Figura 3.2, visita una grande quantità di nodi al di fuori di quelli che compongono il cammino di costo minimo. Ciò che possiamo fare per rendere ancora più efficiente l'algoritmo è ridurre questi nodi, e cioè espandere solo (o quasi) i nodi necessari a produrre un cammino ottimale. In breve, miriamo a rendere l'algoritmo di Dijkstra un algoritmo *informato*.

Euristica

Per raggiungere l'obiettivo prefissato, dobbiamo quindi modificare in qualche modo l'ordine di visita dei nodi rendendo più "appetibili" quelli in dire-

zione della destinazione.
Chiamiamo *funzione di valutazione* una particolare funzione che assegna ad ogni nodo un valore, che indica la priorità all'interno della heap. In particolare, la funzione di valutazione utilizzata in Dijkstra è $d(i)$.
Una funzione di valutazione f è del tipo:

$$f(i) = d(i) + h(i)$$

dove $d(i)$ è il costo minimo per raggiungere i partendo dalla sorgente. mentre $h(i)$ è quella che viene chiamata *euristica*, ovvero una *stima* della distanza che separa il nodo i dal nodo destinazione.

Eseguendo l'algoritmo di Dijkstra, associando però la priorità ai nodi in base alla funzione di valutazione del tipo di cui sopra, "informiamo" l'algoritmo della posizione del nodo di arrivo e quello, di conseguenza, va ad espandersi in quella direzione.
L'algoritmo di Dijkstra con tale modifica è chiamato $A*$ (A star). Vediamo ora una sua implementazione.

La struttura dell'algoritmo è molto simile a quello di Dijkstra, in particolare l'inizializzazione delle strutture dati e gran parte del ciclo while sono identiche. Tuttavia avremo bisogno di un parametro di input in più in cui specifichiamo la funzione euristica utilizzata.

```
1 def Astar(graph, source, target, heuristic):
2 distance = {}
3 prev = {}
4 for i in nodes:
5     distance[i] = 999999
6     prev[i] = None
7 distance[source] = 0
8 visited = []
9 q = [(0, source)]
```

```
10 while target not in visited:
11     curr_dist, curr_node = heappop(q)
12     if curr_node not in visited:
13         visited.append(curr_node)
```

L'unica differenza risiede all'interno del ciclo **for** in particolare alle righe 18 e 19, in cui assegniamo la priorità ai nodi da inserire nella heap in base alla funzione di valutazione `distance[neighbor] + h`.

```
14 for neighbor, cost in graph[curr_node]:
15     if distance[curr_node] + cost < distance[neighbor
       ]:
16         distance[neighbor]= distance[curr_node] +
           cost
17         prev[neighbor]= curr_node
18         h = heuristic(neighbor, target)
19         heappush(q, (distance[neighbor] + h, neighbor
           ))
```

Osservazioni

Consideriamo ora il *grado di informazione* dell'algoritmo dato dall'euristica. Se l'informazione è nulla, cioè $h(i) = 0, \quad \forall i \in N$, abbiamo che $f(i) = d(i)$ quindi la visita si riduce ad esplorare tanti nodi quanti Dijkstra. Se l'informazione è "perfetta" cioè $h(i) = c(i)$ con $c(i)$ distanza minima effettiva tra i e la destinazione, l'algoritmo visita **solo** i nodi che compongono il cammino minimo, ma ci troveremmo a dover far girare un Dijkstra single-target solo per calcolare $h(i)$ ottenendo l'effetto opposto a quello desiderato. Bisogna quindi trovare un equilibrio tra accuratezza e costo del calcolo dell'euristica in modo da trarre vantaggio dalle iterazioni risparmiate per visitare nodi non utili a produrre un cammino minimo.

Vediamo ora come si comporta l'algoritmo[5] A* sulla stessa mappa di Figura 3.2.

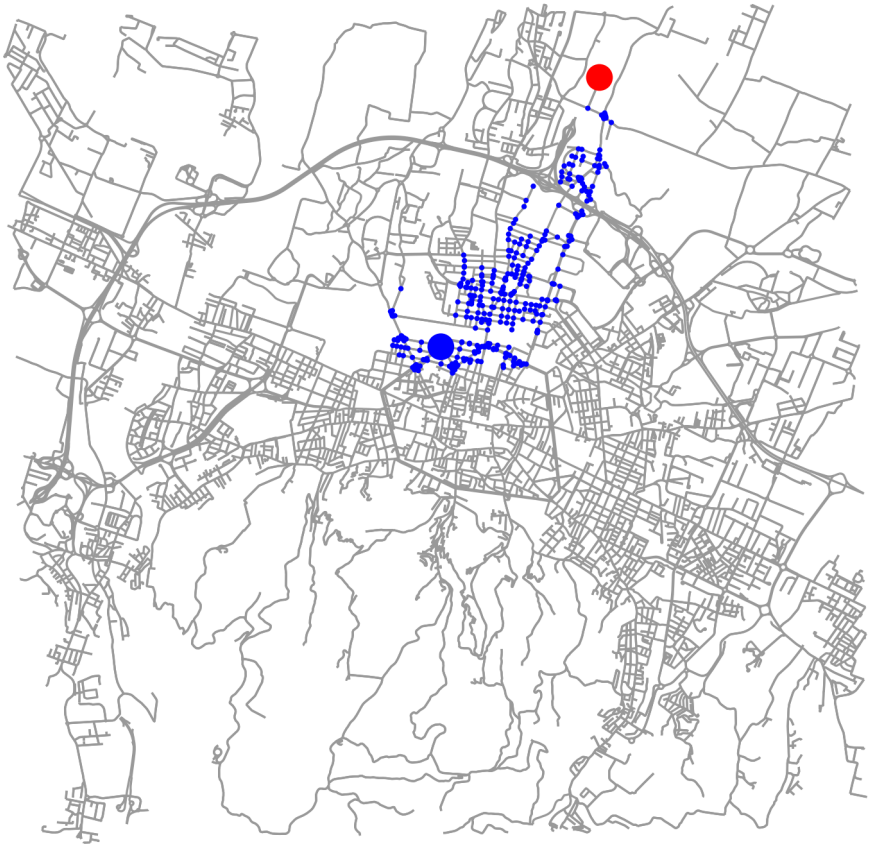


Figura 3.3: Algoritmo A* euclideo eseguito sulla mappa di Bologna.

Come possiamo vedere dalla Figura 3.3, l'espansione di A* è in direzione del target e visita molti meno nodi dell'algoritmo di Dijkstra.

Correttezza

Perché l'algoritmo, con la suddetta funzione di valutazione, risulti ammissibile l'euristica utilizzata deve soddisfare le seguenti due proprietà.

[5]In Figura 3.3 abbiamo utilizzato l'euristica euclidea. Nel prossimo capitolo vedremo nel dettaglio come costruire un'euristica utilizzando una metrica.

Definizione 3. Un'euristica h si dice *ammissibile* se:

$$h(i) \leq c(i) \quad \forall i \in N$$

con $c(i)$ il costo effettivo per arrivare a destinazione. A parole, è ammissibile se il costo effettivo viene sottostimato.

Definizione 4. Un'euristica h si dice *consistente* se:

$$h(i) - h(j) \leq c_{ij} \quad \forall (i,j) \in A$$

. A parole, è consistente se il triangolo di lati: $h(i)$, $h(j)$, c_{ij} soddisfa la disuguaglianza triangolare, per ogni $(i,j) \in A$

Si può dimostrare che la seconda condizione è più forte della prima, in particolare, la consistenza di un'euristica implica la sua ammissibilità.

Prima di enunciare il teorema sulla correttezza dell'algoritmo abbiamo bisogno del seguente lemma.

Lemma 1. *Sia G un grafo orientato con costi c_{ij} per ogni arco (i,j). Sia G' un grafo come G ma con costi ridotti $c'_{ij} = c_{ij} - h(i) + h(j)$, con h un'euristica. Sia $c_{u \to v}$ il costo di un cammino in G da u a v, e $c'_{u \to v}$ il costo dello stesso cammino in G'. Allora*

$$c'_{u \to v} = c_{u \to v} - h(u) + h(v) \quad \forall u, v \in N$$

Dimostrazione. Supponiamo che il cammino da u a v sia della forma $(u, i_1, i_2, \ldots, i_n, v)$. Allora, per definizione di costo di un cammino, abbiamo che

$$c'_{u \to v} = c'_{ui_1} + c'_{i_1 i_2} + \cdots + c'_{i_n v} =$$

Ora, per definizione di costo ridotto abbiamo

$$= \underbrace{c_{ui_1} - h(u) + h(i_1)}_{c'_{ui_1}} + \underbrace{c_{i_1 i_2} - h(i_1) + h(i_2)}_{c'_{i_1 i_2}} + \cdots + \underbrace{c_{i_n v} - h(i_n) + h(v)}_{c'_{i_n v}} =$$

ciò che rimane eliminando i termini opposti è

$$= c_{ui_1} + c_{i_1 i_2} + \cdots + c_{i_n v} - h(u) + h(v) = c_{u \to v} - h(u) + h(v)$$

□

Teorema 5. *L'algoritmo A* è ammissibile se l'euristica utilizzata $h(i)$ è consistente*

Dimostrazione. La dimostrazione si appoggia alla correttezza dell'algoritmo di Dijkstra. L'idea è quella di far vedere che l'algoritmo A* con un'euristica consistente eseguito su di un grafo G è equivalente all'algoritmo di Dijkstra eseguito sul grafo G' con i costi modificati, vale a dire, i due algoritmi hanno stesso ordine di visita e stessa soluzione. In questo modo essendo Dijkstra ammissibile, lo sarà anche A*.

Per quanto riguarda l'ordine di visita dei nodi abbiamo che:

- Dijkstra sceglie in G' i nodi in ordine crescente dai valori

$$c'_{s \to i} = c_{s \to i} - h(s) + h(i)$$

.

- A* sceglie in G i nodi in ordine crescente dai valori

$$f(i) = c_{s \to i} + h(i)$$

Il valore $h(s)$ è una costante quindi l'ordine di visita rimane invariato.

Inoltre, dal lemma, scende che il costo di ogni cammino dalla sorgente s alla destinazione t in G differisce dal costo dello stesso cammino in G' per una costante $h(t) - h(s)$. Perciò un cammino minimo in G' è minimo anche in G. Infine, dato che l'euristica è consistente per ipotesi, i costi ridotti in G' risultano tutti positivi, di conseguenza la soluzione prodotta dall'algoritmo di Dijkstra è corretta, e quindi lo è anche quella dell'algoritmo A* in G. □

In conclusione, se ci serviamo di un'euristica non consistente non è detto che il cammino trovato sia minimo. Va specificato però che un'euristica

accurata e allo stesso tempo consistente è spesso costosa in termini di computazione, quindi, per applicazioni reali (ad esempio per trovare il cammino che attraversi un'intera nazione) può essere preferibile utilizzare un'euristica non consistente in modo da produrre un "buon" cammino in tempi più brevi[6].

3.2.5 Bi-directional Dijkstra

Qui presentiamo un algoritmo che rientra sotto la categoria *ricerca bidirezionale*.
L'idea di base degli algoritmi bidirezionali è quella di eseguire una ricerca "in avanti" dalla sorgente e una ricerca "all'indietro" dalla destinazione fino a che queste non si incontrino.
In particolare, implementeremo un Dijkstra bidirezionale composto da un Dijkstra single-source insieme ad un Dijkstra single-target.

Importante è la stopping condition dell'algoritmo per garantire la sua ammissibilità.
Una possibile idea (sbagliata) potrebbe essere quella di eseguire le due visite finché non abbiamo un'intersezione non vuota tra i due array visited[7], e cioè quando una delle due visite incontra un nodo già esplorato dall'altra. In realtà, quel nodo in comune può non appartenere al cammino di costo minimo.
Per garantire l'ammissibilità è necessario invece tenere in memoria una variabile μ (posta inizialmente a $+\infty$) che rappresenta il costo del cammino minimo finora trovato tra s e t ed aggiornare tale variabile nel caso trovassimo un cammino migliore. L'algoritmo si interrompe quando il valore che indica la priorità del nodo in cima ad una delle due heap (quindi quello da visitare all'iterazione successiva) sommato al valore che indica la priorità del

[6]Chiaramente la bontà del cammino e il tempo di esecuzione dell'algoritmo dipenderanno dall'euristica utilizzata.

[7]Dato che eseguiremo due visite, avremo bisogno di due diversi array visited, uno per la visita in avanti e uno per la visita all'indietro.

nodo in cima all'altra heap risulterà maggiore o uguale a μ. Vedremo poi perchè questa condizione assicura l'ammissibilità.

Prima di vedere una sua implementazione, mostriamo una funzione che sarà utile per "stabilizzare" l'algoritmo.

```
1 def min_heap(heap, visited):
2     bool = True
3     while bool:
4         curr_dist, curr_node = heappop(heap)
5         if curr_node not in visited:
6             bool = False
7     return curr_dist, curr_node
```

Questa funzione restituisce semplicemente il nodo in cima alla heap con relativa priorità. Però, abbiamo dovuto "personalizzarla" per il problema riscontrato nell'algoritmo `HeapDijkstra()`, cioè quello in cui vi sono nodi multipli con stesso nome ma diversa priorità nella heap.
Come vedremo, l'algoritmo esegue sistematicamente un'iterazione in avanti seguita da un'iterazione all'indietro. Senza la suddetta funzione è possibile che l'algoritmo salti una delle due iterazioni perché ha trovato un nodo già visitato, quindi un doppione con priorità obsoleta. Tale funzione serve appunto per distribuire uniformemente il lavoro svolto in entrambe le visite.

Dato che si tratta di una ricerca bidirezionale, dobbiamo raddoppiare tutte le strutture dati. In particolare, abbiamo bisogno di due grafi, `graph_1` che rappresenta la mappa originale su cui effettuiamo la visita in avanti e `graph_2` lo stesso grafo ma con gli archi in direzione opposta sul quale effettuiamo la visita all'indietro[8].
Inoltre inizializziamo le variabili `curr_dist_1` e `curr_dist_2` per entrare nel

[8]Tutte le strutture seguite dal numero 1 hanno a che fare con la visita in avanti, quelle seguite dal numero 2, con la visita all'indietro.

ciclo **while**, una volta dentro, a queste variabili assegniamo i reali valori che poi ci permetteranno di terminare l'algoritmo.

```
1  def Bidirectional_Dijkstra(graph_1, graph_2, source,
       target):
2
3  distance_1 = {}
4  distance_2 = {}
5  prev_1 = {}
6  prev_2 = {}
7  visited_1 = []
8  visited_2 = []
9
10 for i in nodes:
11     distance_1[i] = 99999
12     prev_1[i] = None
13     distance_2[i] = 99999
14     prev_2[i] = None
15
16 distance_1[source] = 0
17 distance_2[target] = 0
18 q_1 = [(0, source)]
19 q_2 = [(0, target)]
20 mu = 9999999
21 curr_dist_1 = 0
22 curr_dist_2 = 0
```

Qui inizia il ciclo **while**, in cui vi è un'iterazione in avanti seguita da un iterazione all'indietro. Analizziamo ora l'iterazione per la ricerca in avanti, quella all'indietro sarà analoga.

Come vediamo l'iterazione è identica a quella di `HeapDijkstra()` (utilizzando però la funzione `min_heap()`) tuttavia, al termine di essa dobbiamo

aggiornare il valore μ se la condizione di riga 31 viene soddisfatta.
Inoltre, teniamo in memoria due variabili `node_in_path` e `neigh_in_path` che ci serviranno per ricostruire il cammino al termine dell'algoritmo.

```
23 while (curr_dist_1 + curr_dist_2 < mu):
24 curr_dist_1, curr_node_1 = min_heap(q_1, visited_1)
25 visited_1.append(curr_node_1)
26 for neighbor, cost in graph_1[curr_node_1]:
27     if curr_dist_1 + cost < distance_1[neighbor]:
28         distance_1[neighbor]= curr_dist_1 + cost
29         padre_1[neighbor]= curr_node_1
30         heappush(q_1, (distance_1[neighbor], neighbor
               ))
31         if neighbor in visited_2 and curr_dist_1 +
               cost + distance_2[neighbor] < mu:
32             mu = curr_dist_1 + cost + distance_2[
                   neighbor]
33             node_in_path = curr_node_1
34             neigh_in_path = neighbor
```

L'iterazione nella visita all'indietro è analoga alla precedente, tranne che per l'assegnazione alle variabili `node_in_path` e `neigh_in_path` che risultano invertite.

```
35 curr_dist_2, curr_node_2 = min_heap(q_2, visited_2)
36 visited_2.append(curr_node_2)
37 for neighbor, cost in graph_2[curr_node_2]:
38     if curr_dist_2 + cost < distance_2[neighbor]:
39         distance_2[neighbor]= curr_dist_2 + cost
40         padre_2[neighbor]= curr_node_2
41         heappush(q_2, (distance_2[neighbor], neighbor
```

```
            ))
42          if neighbor in visited_1 and distance_2[
               curr_node_2] + cost + distance_1[neighbor]
                < mu:
43              mu = distance_2[curr_node_2] + cost +
                   distance_1[neighbor]
44              node_in_path = neighbor
45              neigh_in_path = curr_node_2
```

Queste ultime righe di codice mostrano come costruire l'array dei nodi che compongono il cammino minimo dai dizionari `prev_1` e `prev_2`. Il motivo per il quale alla seconda iterazione abbiamo invertito i valori da assegnare alle variabili `node_in_path` e `neigh_in_path` è che in questo modo l'algoritmo genera il giusto cammino sia nel caso in cui l'ultimo aggiornamento della variabile μ viene effettuato alla prima iterazione o alla seconda.

In particolare creiamo due array `path_1` e `path_2` provenienti rispettivamente dalla visita in avanti e dalla visita all'indietro (l'array `path_1` ha bisogno di essere invertito mentre il `path_2` no). Uniamo infine i due array.

```
46  rev_path = [node_in_path]
47  k = 0
48  while source not in rev_path:
49      rev_path.append(prev_1[rev_path[k]])
50      k = k + 1
51  path_1 = rev_path[::-1]
52  path_2 = [neigh_in_path]
53  j = 0
54  while target not in path_2:
55      path_2.append(prev_2[path_2[j]])
56      j = j + 1
57  path = path_1 + path_2
```

```
58  return path
```

Osservazioni

Anche se possiamo considerarlo un algoritmo non informato si vedrà che performa meglio del semplice Dijkstra. Vediamo innanzitutto come questo si comporta nella mappa di Figura 3.2.

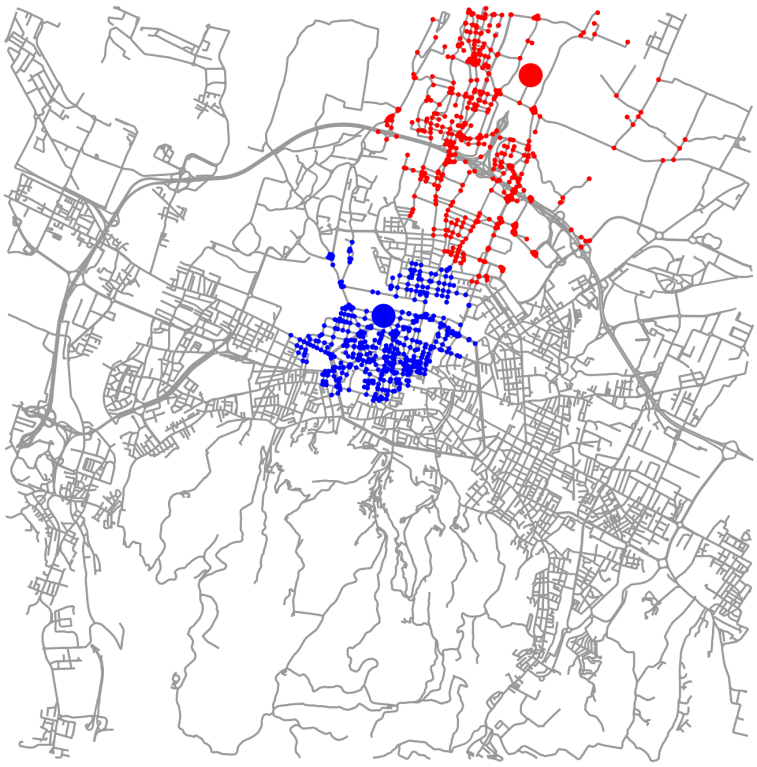


Figura 3.4: Algoritmo di Dijkstra bidirezionale eseguito sulla mappa di Bologna.

In Figura 3.4, in blu abbiamo i nodi esplorati dalla visita in avanti, mentre in rosso i nodi esplorati dalla visita all'indietro.

Intuitivamente possiamo pensare alla quantità di spazio che i due algoritmi espandono. Lo spazio visitato da Dijkstra può essere approssimato da una circonferenza di raggio $d(i)$ e di centro il nodo sorgente s, avente quindi

area uguale a $A_d = \pi d(i)^2$ (vedi Figura 3.2). Mentre l'algoritmo bidirezionale ne visita due di raggio $\frac{d(i)}{2}$ quindi l'area totale sarà $A_b = 2\pi(\frac{d(i)}{2})^2 = \frac{1}{2}\pi d(i)^2$ e quindi minore di A_d (vedi Figura 3.4).
Chiaramente questa è solo un'idea ma si riflette sul numero di nodi visitati dall'algoritmo.

Correttezza

Chiamiamo S e $\tilde{S}$, rispettivamente, l'insieme dei nodi esplorati dalla visita in avanti e l'insieme dei suoi nodi di frontiera (e cioè i nodi all'interno della relativa heap) ad una generica iterazione. Siano T e $\tilde{T}$ gli analoghi insiemi per la visita all'indietro. Siano t_a la distanza del nodo da espandere all'iterazione successiva della visita in avanti (quindi quello con distanza provvisoria minima) e t_b l'equivalente nella visita all'indietro.

L'algoritmo si interrompe dunque non appena la seguente condizione risulta soddisfatta:

$$t_a + t_b \geq \mu \tag{3.2}$$

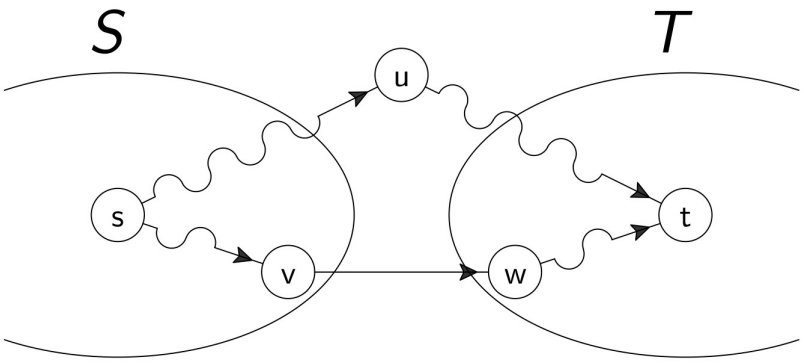


Figura 3.5: Illustrazione della correttezza dell'algoritmo di Dijkstra bidirezionale.

Teorema 6. *L'algoritmo di Dijkstra bidirezionale con la stopping condition (3.2) è corretto.*

Dimostrazione. Per dimostrare la correttezza dell'algoritmo, facciamo un ragionamento sul suo stadio finale, cioè quando esso termina.
Consideriamo quindi il generico cammino P che va da s a t e mostriamo che $c(P) \geq \mu$. Si presentano due casi:

- P passa per un nodo $u \notin S \cup T$ e cioè un nodo che non è stato esplorato da nessuna delle due visite. Abbiamo quindi che $c(P) = c(P_1) + c(P_2)$ con P_1 un cammino che va dalla sorgente s a u, P_2 un cammino che va da u alla destinazione t.
  Dato che $u \notin S$, dal Teorema 2 scende che:

  $$c(P_1) \geq t_a$$

  Analogamente:

  $$c(P_2) \geq t_b$$

  Di conseguenza abbiamo che:

  $$c(P) \geq t_a + t_b \geq \mu$$

  .

- P passa per un arco (v, w) tale che $v \in S$ e $w \in T$. Abbiamo quindi che $c(P) = c(P_3) + c_{vw} + c(P_4)$ con P_3 un cammino da s a v e P_4 un cammino che va da w a t.
  Senza perdita di generalità possiamo considerare solo i cammini tali per cui P_3 e P_4 siano di costo minimo (gli altri sarebbero inevitabilmente più costosi). I cammini di questo tipo corrispondono esattamente a quelli trovati dall'algoritmo durante la sua visita. Il fatto che i cammini trovati dall'algoritmo passino per un'arco (v, w) tale che $v \in S$ e $w \in T$ è assicurato dalla regola di aggiornamento di μ (riga 31 e 42 nel codice), mentre il fatto che i suoi sottocammini P_3 e P_4 siano di costo minimo

è assicurato dal Teorema 2. Di conseguenza, nel calcolo nella variabile μ è stato tenuto conto della lunghezza di ogni cammino di questo tipo. Avremo quindi che:

$$c(P) \geq \mu$$

□

Osserviamo che la dimostrazione dell'ammissibilità non fa riferimento a come l'algoritmo alterna l'espansione delle due visite (anche detta *regola decisionale*). Questo fatto fa sorgere la questione di quale regola decisionale utilizzare.
Possiamo ad esempio scegliere di alternare in maniera casuale l'espansione della visita in avanti e della visita all'indietro, oppure scegliere di espandere una sola visita (e ritornare però all'algoritmo di Dijkstra). Dunque, come ultimo, mostriamo l'algoritmo bidirezionale presentato in [6], la cui regola decisionale si basa sul concetto di densità del grafo.

Il calcolo a priori della densità del grafo può risultare complesso e costoso per ogni diversa istanza del problema, tuttavia, la cardinalità dell'insieme dei nodi di frontiera della visita in avanti $\tilde{S}$ e della visita all'indietro $\tilde{T}$ è a noi conosciuta ad ogni passo dell'algoritmo. Queste quantità riflettono la densità locale delle regioni adiacenti alle due visite S e T senza dover effettuare ulteriori calcoli per stimare la densità.
Perciò, utilizziamo la regola decisionale che stabilisce: se $|\tilde{S}| < |\tilde{T}|$ allora espandiamo la visita in avanti S, altrimenti espandiamo T. In questo modo l'algoritmo espande maggiormente la visita situata nella regione più sparsa del grafo.

Omettiamo di mostrare l'implementazione perché differisce dalla precedente per sole due righe di codice: `if len(q_1)< len(q_2)` posto tra la riga 23 e la riga 24 nell'implementazione precedente, ed `Else` tra la riga 34 e la

riga 35.

Bisogna dire che, ancora una volta, ci imbattiamo nel problema riscontrato nell'algoritmo `HeapDijkstra()`, perché esso influisce sul numero di nodi di frontiera. Tuttavia, questo va ad incrementare la cardinalità di **entrambi** gli insiemi e non modifica il comportamento dell'algoritmo. Infatti, i test effettuati nel prossimo capitolo confermano che l'algoritmo espande maggiormente la visita della zona di grafo considerata sparsa, che è proprio il risultato ricercato.

Capitolo 4

Esperimenti

In questo capitolo andremo, finalmente, a testare gli algoritmi su vere mappe stradali di alcune città europee scelte in base a diversi criteri. In particolare suddividiamo il capitolo in tre sezioni.

- **Analisi:** qui specifichiamo quali algoritmi, tra quelli proposti nel terzo capitolo, andremo a testare e quale parametro prendiamo in considrazione per giudicare/misurare la loro performance;
- **Instance set:** in questa sezione mostreremo le mappe scelte ed il criterio di scelta, insieme a come verrà condotto il test;
- **Risultati:** in cui riportiamo i risultati ottenuti e le conclusioni.

4.1 Analisi

Gli algoritmi che andremo ad analizzare saranno quattro:

- A* con euristica euclidea
- A* con euristica di Chebyshev
- Dijkstra bidirezionale
- una variante del Dijkstra bidirezionale

Per stimare la distanza tra due nodi in una mappa, e quindi per costruire l'euristica h, ci serviamo delle coordinate geografiche UTM, cioè una coppia di valori (x, y) che permette di individuare un punto nel globo terrestre, proprio come le coordinate latitudine e longitudine. Però, a differenza di queste ultime, le coordinate UTM hanno come unità di misura il metro e ci permettono quindi di calcolare la funzione di valutazione f[1].
Quindi l'algoritmo A* con euristica euclidea (chiamato d'ora in poi A_1) stima la distanza tra due nodi n_1 e n_2 con coordinate rispettivamente (x_1, y_1) e (x_2, y_2) tramite la funzione

$$dist(n_1, n_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

L'algoritmo A* con l'euristica di Chebyshev (chiamato d'ora in poi A_2) stima la distanza tra i nodi n_1 e n_2 tramite la funzione

$$dist(n_1, n_2) = max\{|x_1 - x_2|, |y_1 - y_2|\}$$

Il Dijkstra bidirezionale, che chiameremo BD_1, si riferisce a quello descritto nel capitolo 3, in cui viene effettuata sistematicamente un'iterazione in avanti seguita da un'iterazione all'indietro, mentre la variante chiamata BD_2 si riferisce all'algoritmo presentato in [6] in cui si alterna la visita in avanti e la visita all'indietro in base alla densità del grafo.

I criteri utilizzati per misurare gli algoritmi sono i seguenti:

- Tempo di esecuzione
- Quantità di nodi visitati

[1] La funzione di valutazione f è definita come la somma delle funzioni d ed h, utilizzando le coordinate UTM entrambe le funzioni esprimono quantità nella stessa unità di misura.

4.2 Instance set

Il test verrà strutturato nel seguente modo.

Sia $G = (N, A)$ il grafo associato ad una mappa stradale:

1. prendiamo due sottoinsiemi S e T di N, che chiamiamo rispettivamente *insieme sorgente* e *insieme destinazione*;

2. da questi generiamo un array I (chiamato l'insieme delle istanze o *instance set*) contenente coppie di nodi del tipo $(sorgente, destinazione)$ presi in modo casuale dai rispettivi insiemi;

3. Infine facciamo girare gli algoritmi su ogni istanza di I (contenente circa 100 istanze) prendendo quindi una media dei nodi visitati e del tempo di esecuzione per ogni algoritmo.

La scelta delle mappe su cui verranno eseguiti gli algoritmi di ricerca verrà effettuata in base a due caratteristiche morfologiche:

- **Densità variabile:** le mappe scelte in base a questa caratteristica presentano una regione più densa e una più sparsa (ad esempio il centro e la periferia di una città rispettivamente). In queste mappe analizziamo come si comportano gli algoritmi prendendo come insieme sorgente alcuni nodi nella regione densa e come insieme destinazione alcuni nodi della regione sparsa.

- **Presenza di ostacoli:** le mappe scelte in base a questa caratteristica presentano "ostacoli" come fiumi, laghi, vicinanza alla costa, ecc, di conseguenza analizziamo come l'algoritmo cerca di evitare l'ostacolo quindi cercando ponti, aggirando laghi, ecc. In queste mappe l'algoritmo non "gira liberamente" come nelle mappe della tipologia precedente ma è vincolato dalla morfologia di tale mappa.

4.2.1 Mappe a densità variabile

Per testare gli algoritmi su questa tipologia di mappe, i due insiemi S e T vengono costruiti rispettivamente nella regione più densa/sparsa della mappa, in particolare:

- L'insieme S (situato geograficamente in centro città) sarà composto da quei nodi all'interno di una circonferenza di raggio opportuno, con centro una particolare attrazione come una piazza o un monumento principale della città;
- L'insieme T (situato geograficamente in periferia) sarà composto da quei nodi all'interno di una corona circolare (o parte di essa) concentrica alla circonferenza precedente.

In questo modo, l'insieme delle istanze I generato dagli insiemi S e T avranno come soluzione un cammino di lunghezza (quasi) costante per ogni coppia (s, t) in I.

In realtà, come vedremo, prenderemo più insiemi destinazione $T_1, T_2, \ldots T_n$, cioè n corone circolari di raggio crescente (e quindi n instance set $I_1, I_2, \ldots I_n$) per vedere se la lunghezza della soluzione influisce sulle performance dei vari algoritmi.
Vediamo nel dettaglio le mappe di questa tipologia prese in considerazione.

Kilkenny

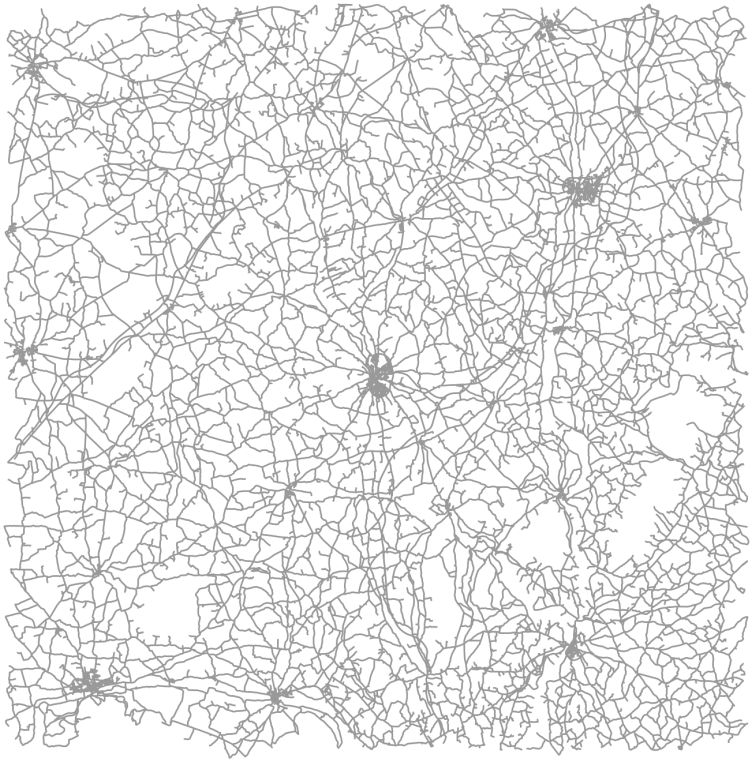


Figura 4.1: Kilkenny (al centro) e dintorni: 6'400 km^2, 16'393 nodi, 38'362 archi.

In questa mappa abbiamo selezionato 5 insiemi: L'insieme sorgente S e gli insiemi destinazione T_1, T_2, T_3, T_4. Tutti sono costruiti in base ad un nodo i scelto opportunamente nella parte interna alla città. Vediamo in dettaglio.

Sia $G = (N, A)$ il grafo associato alla mappa di Figura 4.1. Sia $d(j)$ la lunghezza (in metri) del cammino minimo dal nodo i (precedentemente scelto) al nodo j.[2]

$$S = \{j : j \in N, d(j) < 1500\}$$

[2]Per ottenere il valore di $d(j)$ per ogni $j \in N$ abbiamo eseguito l'algoritmo di Dijkstra single source con sorgente i.

$$T_1 = \{j : j \in N, 10000 < d(j) < 14000\}$$

$$T_2 = \{j : j \in N, 15000 < d(j) < 18000\}$$

$$T_3 = \{j : j \in N, 21000 < d(j) < 24000\}$$

$$T_4 = \{j : j \in N, 25000 < d(j) < 28000\}$$

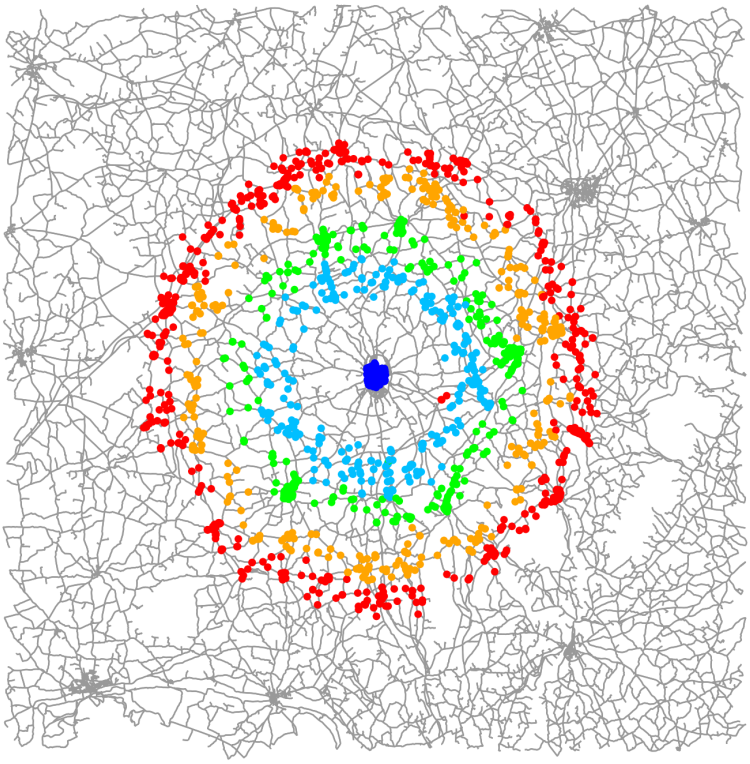


Figura 4.2: Insiemi test per Kilkenny.

La Figura 4.2 mostra la disposizione di tali insiemi sulla mappa, in blu l'insieme S, in azzurro l'insieme T_1, in verde l'insieme T_2, in arancio l'insieme T_3, in rosso l'insieme T_4.

Bologna

In questa mappa abbiamo selezionato 4 insiemi (S, T_1, T_2, T_3) costruiti come prima, tuttavia in questo caso non consideriamo tutta la corona circolare, ma solamente una parte proprio per evitare che, scegliendo in maniera

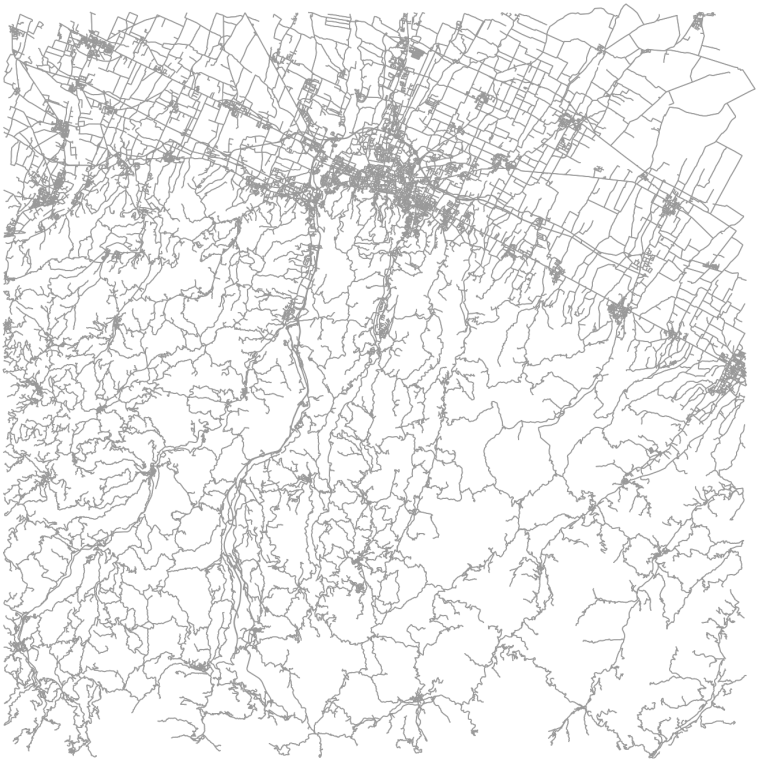


Figura 4.3: Zona sud di Bologna: 3'600 km^2, 28'844 nodi, 63'550 archi.

casuale i nodi dagli insiemi T_i, possa capitare di pescare nodi nelle zone nord/ovest/est di Bologna la cui differenza di densità rispetto al centro non è poi così rilevante.
Siano dunque $x(j)$ e $y(j)$ le coordinate geografiche UTM del nodo j. Chiamiamo:

$$A = \{j : j \in N, \quad y(j) < \alpha, \quad \beta < x(j) < \gamma\}$$

Per α, β e γ opportunamente scelti. Definiamo quindi l'insieme sorgente e gli insiemi destinazione come:

$$S = \{j : j \in N, d(j) < 1600\}$$

$$T_1 = \{j : j \in N, 17000 < d(j) < 22000\} \cap A$$

$$T_2 = \{j : j \in N, 24000 < d(j) < 28000\} \cap A$$

$$T_3 = \{j : j \in N, 29000 < d(j) < 32000\} \cap A$$

Ancora, in Figura 4.4 visualizziamo tali insiemi sulla mappa, in blu l'insieme S, in verde l'insieme T_1, in arancio l'insieme T_2, in rosso l'insieme T_3.

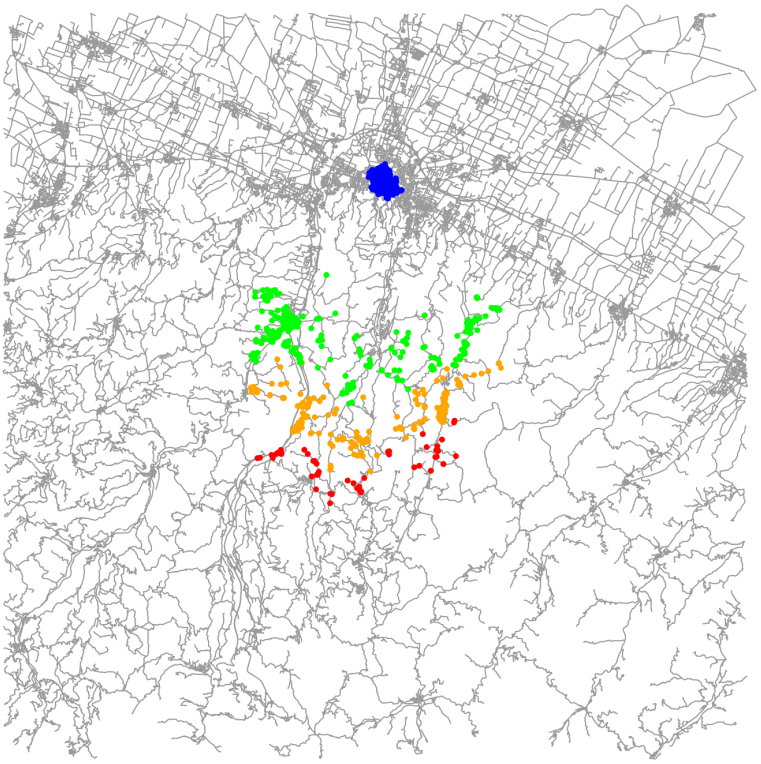


Figura 4.4: Insiemi test per Bologna.

4.2.2 Mappe con presenza di ostacoli

Le mappe di questa tipologia prese in considerazione sono caratterizzate dalla presenza di fiumi, corsi d'acqua che "ostacolano" la ricerca.
Per testare gli algoritmi su queste mappe abbiamo scelto gli insiemi S, $T_1, T_2, \ldots, T_n$ tutti della stessa forma, vale a dire $n+1$ circonferenze di raggio e centro opportuno. In particolare, fissiamo $n+1$ nodi su cui costruiamo altrettante circonferenze di egual raggio.

Ancora una volta, (se il raggio di ogni insieme è relativamente piccolo) gli elementi del generico insieme delle istanze del tipo

$$I_i = \{(s,t) : s \in S, t \in T_i\}$$

avranno come soluzione un cammino di lunghezza (quasi) costante per ogni coppia (s,t) in I_i.
Vediamo nel dettaglio le mappe di questa tipologia prese in esame.

Stoccolma

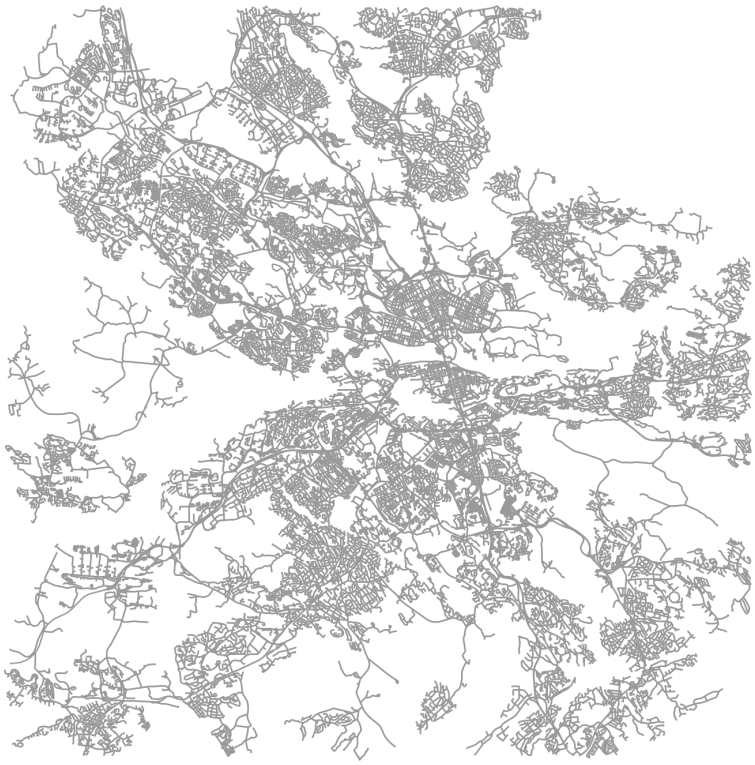


Figura 4.5: Stoccolma: 900 km^2, 30'830 nodi, 71'736 archi.

In questa mappa abbiamo scelto 4 insiemi (S, T_1, T_2, T_3) costruiti nel seguente modo:

Sia $G = (N, A)$ il grafo associato alla mappa di Figura 4.5, fissiamo quattro nodi i_0, i_1, i_2, i_3. Sia $d_j(k)$ la distanza del cammino minimo tra il nodo i_j e k per $j = 0, 1, 2, 3$. Allora

$$S = \{k : k \in N, d_0(k) < 400\}$$

$$T_1 = \{k : k \in N, d_1(k) < 400\}$$

$$T_2 = \{k : k \in N, d_2(k) < 400\}$$

$$T_3 = \{k : k \in N, d_3(k) < 400\}$$

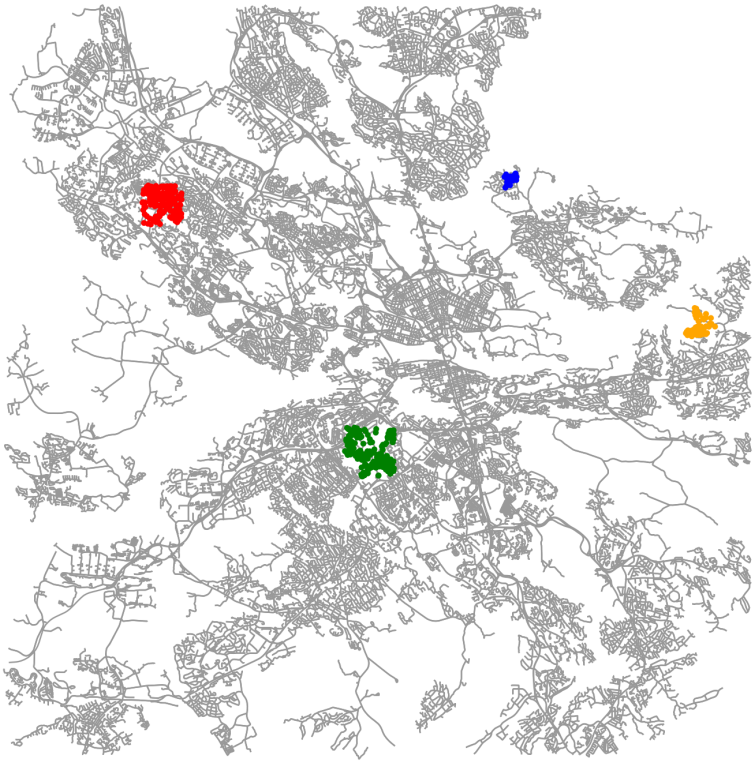


Figura 4.6: Insiemi test per Stoccolma.

In Figura 4.6 visualizziamo tali insiemi sulla mappa, in blu l'insieme S, in giallo l'insieme T_1, in verde l'insieme T_2, in rosso l'insieme T_3.

Rotterdam

Figura 4.7: Zona sud di Rotterdam: 2'500 km^2 20'920 nodi, 52'565 archi.

Qui abbiamo scelto 3 insiemi (S, T_1, T_2) costruiti come prima, quindi:

Sia $G = (N, A)$ il grafo associato alla mappa di Figura 4.7, fissiamo tre nodi i_0, i_1, i_2. Sia $d_j(k)$ la distanza del cammino minimo tra il nodo i_j e k per $j = 0, 1, 2$. Allora

$$S = \{k : k \in N, d_0(k) < 1500\}$$

$$T_1 = \{k : k \in N, d_1(k) < 2900\}$$

$$T_2 = \{k : k \in N, d_2(k) < 1500\}$$

in Figura 4.8 visualizziamo tali insiemi sulla mappa, in blu l'insieme S, in rosso l'insieme T_1, in verde l'insieme T_2.

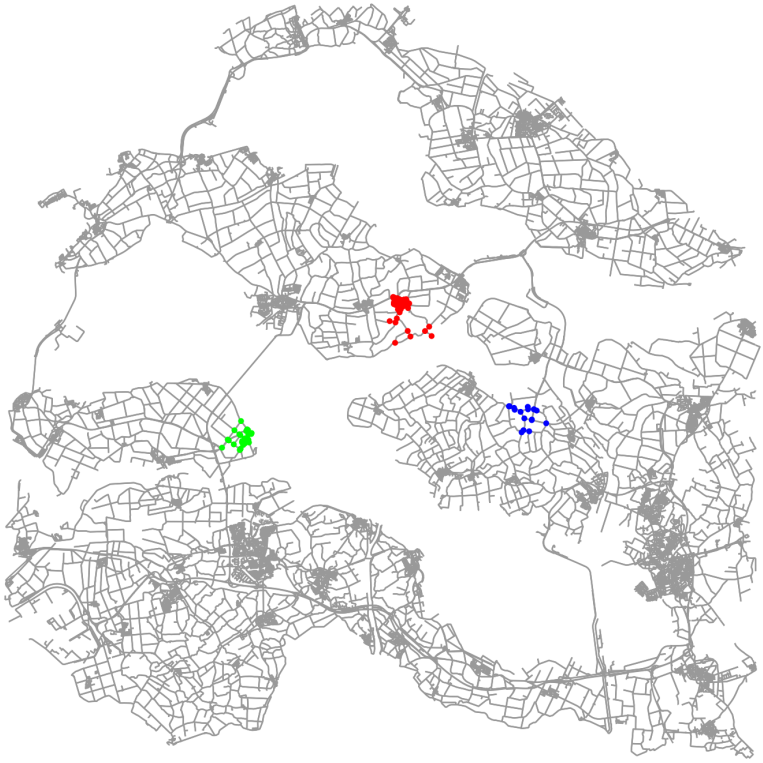


Figura 4.8: Insiemi test per Rotterdam.

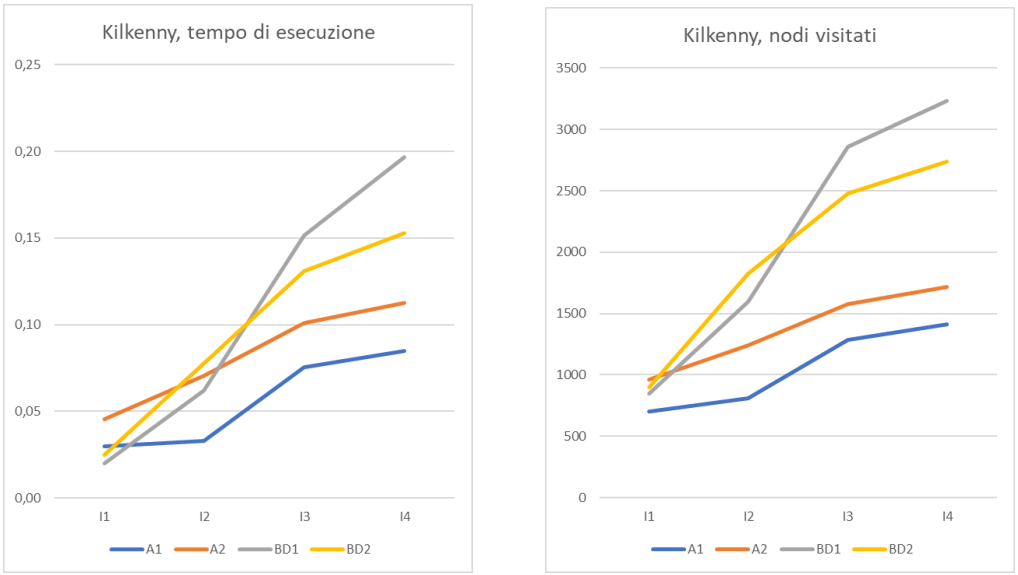
4.3 Risultati

Prima di cominciare con l'analisi vera e propria è bene fare una piccola premessa. Abbiamo detto che l'analisi consiste nell'esaminare la quantità di nodi visitati dall'algoritmo per produrre una soluzione e il tempo che esso impiega. Tuttavia l'analisi per tempo di esecuzione è da considerare *più rilevante* soprattutto se tali algoritmi sono progettati a scopo applicativo come in questo caso.
Nonostante l'analisi dei nodi passi quindi in secondo piano, questa, sarà utile a spiegare e giustificare come e perché un dato algoritmo è risultato più o meno performante di un altro.

Analisi Kilkenny

Iniziamo quindi con l'analizzare le due mappe di densità variabile, in particolare cominciamo con Kilkenny.
La caratteristica di questa mappa è che la differenza di densità tra centro città e periferia è molto "lieve" (vale a dire che la zona densa è, in proporzione, molto piccola se confrontata con la zona sparsa), di conseguenza i nodi appartenenti alla generica soluzione giacciono in gran parte nella zona sparsa della città.

Siccome i tre instance set I_1, I_2, I_3 sono costruiti in base alla loro distanza (crescente) dall'insieme S abbiamo riportato i risultati ottenuti in un grafico a linee per avere quindi un'idea di come si comportano gli algoritmi al crescere della lunghezza della soluzione.



(a) per tempo di esecuzione

(b) per numero di nodi

Figura 4.9: Analisi su Kilkenny.

A prima vista, quello che risalta dai grafici di Figura 4.9 è che l'andamento di ogni algoritmo risulta (come ci aspettavamo) crescente, di conseguenza una soluzione più lunga implica un numero maggiore di nodi da visitare e un

tempo di esecuzione maggiore.

Concentrandoci ora sull'analisi per tempo di esecuzione notiamo che nell'instance set I_1 i due Dijkstra bidirezionali (in giallo e grigio) mostrano performance migliori rispetto ai due A* (in blu e rosso), ma questo risultato si inverte al crescere della lunghezza della soluzione.
Prendendo ora in esame i soli Dijkstra bidirezionali, possiamo osservare che BD_1 performa meglio in I_1 e I_2 mentre in zone più distanti come I_3 e I_4 osserviamo che è BD_2 a prevalere, e questo è dovuto dal fatto che tale algoritmo è stato costruito appositamente per questa situazione, in particolare ci dà la conferma che espandere maggiormente la zona più sparsa porta risultati migliori.

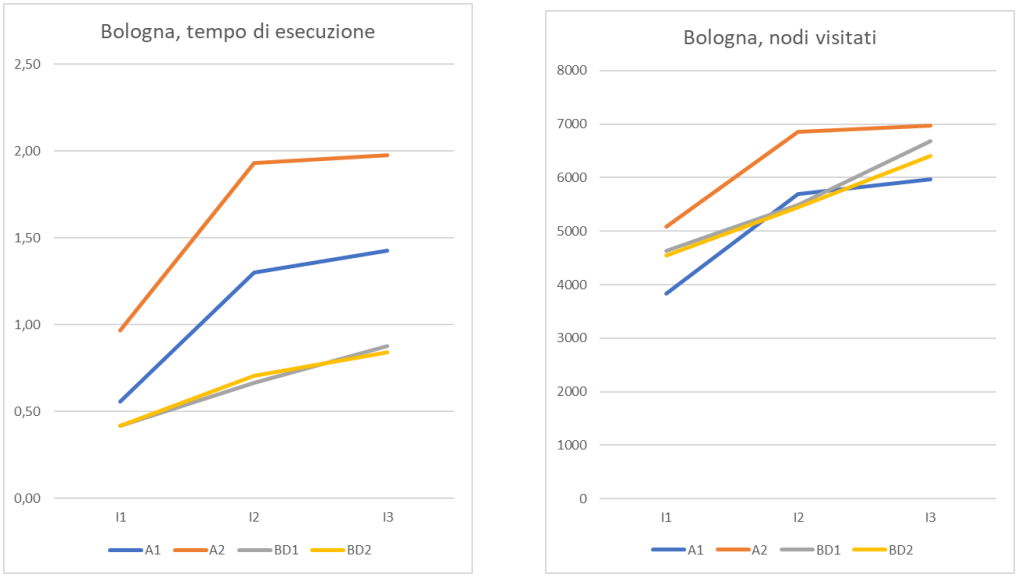
Guardando l'analisi per numero di nodi visitati, vediamo innanzitutto che l'andamento degli algoritmi risulta molto simile all'analisi per tempo di esecuzione. Un fatto interessante è che nell'instance set I_1, quindi per cammini brevi, è l'algoritmo A_1 (euristica euclidea) a visitare il minor numero di nodi, nonostante ciò questo impiega più tempo rispetto a BD_1 e BD_2 per trovare la soluzione. Questo fatto mette in mostra che:

- la generica iterazione dell'algoritmo A_1 per la visita di un singolo nodo, seppur quasi identica all'iterazione di BD_1 o BD_2 (come visto nel nel terzo capitolo), è più costosa proprio a causa del calcolo dell'euristica;

- l'importanza della scelta dell'euristica, in particolare la stima euclidea è risultata molto efficiente in termini di accuratezza infatti grazie ad essa l'algoritmo si è aggiudicato il primo posto di questa analisi, però è risultata pesante in termini di costo computazionale, mentre l'euristica di Cebysev, seppur molto leggera computazionalmente, non è risultata così accurata da rendere l'algoritmo competitivo.

Analisi Bologna

La mappa di Bologna differisce dalla precedente per il fatto che la differenza di densità tra centro città e zona sud è "più accentuata", di conseguenza avremo che l'algoritmo impiega gran parte del tempo nella zona densa, inoltre i nodi contenuti nella soluzione saranno distribuiti equamente tra zona densa e sparsa.



(a) per tempo di esecuzione (b) per numero di nodi visitati

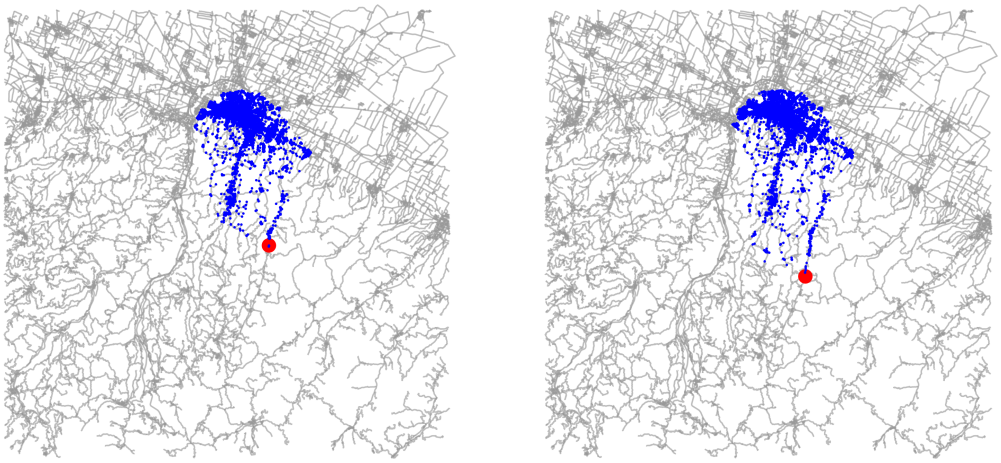
Figura 4.10: Analisi su Bologna.

Anche qui abbiamo rappresentato i dati ottenuti in un grafico a linee per lo stesso motivo di prima.
Considerando dapprima l'analisi per tempo di esecuzione si vede subito come i due algoritmi BD_1 e BD_2 siano più performanti rispetto ad A_1 e A_2.
Confrontando i due grafici, come già osservato dall'analisi precedente, si può notare la pesantezza del calcolo dell'euristica in I_1, in questo caso il risultato è presente anche nelle regioni più distanti dal centro (in I_3).

Un fatto interessante nell'analisi per numero di nodi è l'andamento delle due curve spezzate che rappresentano i due algoritmi A_1 e A_2 che risultano

quasi orizzontali nell'intervallo I_2, I_3 nonostante la diversa distanza (crescente, anche se di poco) dall' insieme S.

A dimostrazione di questo fatto le Figure 4.11 (a) e (b) mostrano il comportamento di A_1 rispettivamente su di un esempio preso da I_2 e uno da I_3 in cui, il nodo sorgente è lo stesso per entrambi gli esempi, mentre il nodo destinazione di I_3 risulta posto poco più a sud di quello di I_2, diciamo "sulla stessa strada".



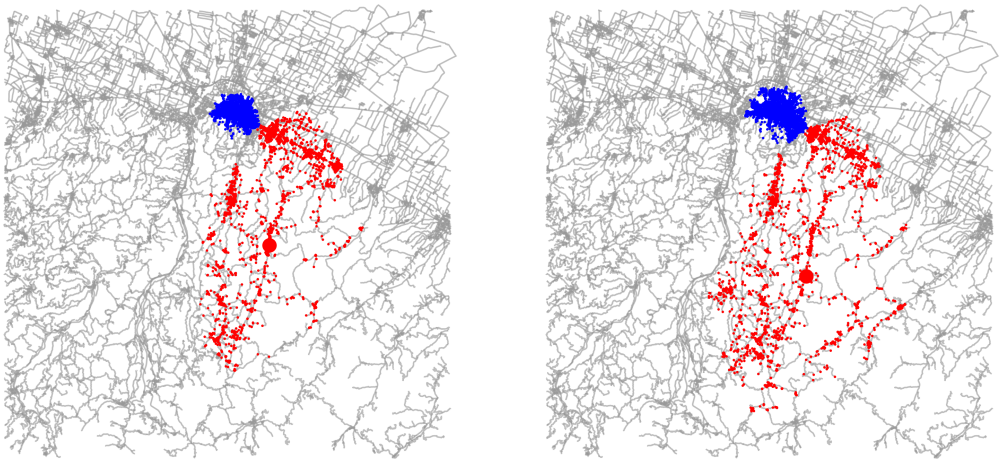
(a) A_1 su I_2 (b) A_1 su I_3

Figura 4.11: Algoritmo A_1 su Bologna.

L'algoritmo A_1 sull'esempio di I_2 ha visitato 5499 nodi, mentre sull'esempio di I_3 ha visitato 5876 nodi (circa il 6% in più). Tutto sommato, la differenza del numero di nodi visitati nei due esempi risulta quasi irrilevante se comparata con la totalità dei nodi visitati.

Considerando invece l'algoritmo BD_2 sugli stessi esempi (Figura 4.12) abbiamo che il primo visita 5102 nodi mentre il secondo ben 6526, quindi circa il 28% di nodi in più.

Le performance dei due algoritmi A_1 e BD_2 sugli instance set I_2 e I_3 lasciano intuire il loro comportamento per cammini molto più lunghi, infatti,



(a) BD_2 su I_2 (b) BD_2 su I_3

Figura 4.12: Algoritmo BD_2 su Bologna.

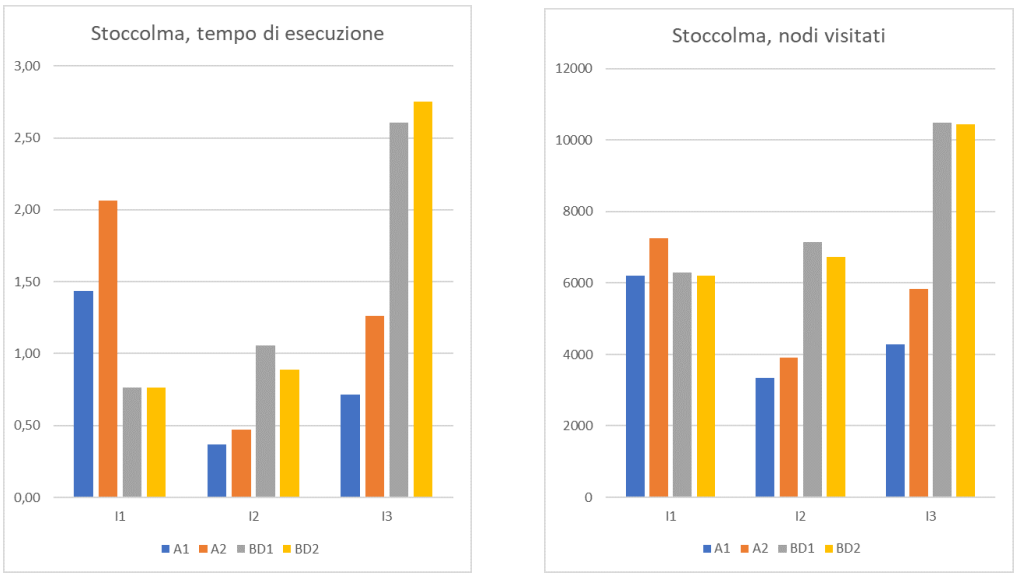
sebbene non ci sia un quarto instance set a confermarlo, possiamo notare che il tasso di crescita dei nodi visitati da BD_2 al variare della lunghezza del cammino è nettamente superiore rispetto tasso di crescita dei nodi visitati da A_1.

Analisi Stoccolma

Prendiamo ora in analisi le mappe con la presenza di ostacoli, cominciamo con la città di Stoccolma.

La mappa di Stoccolma è caratterizzata dalla presenza di molte isole, il che la rende perfetta per questo tipo di analisi.
Qui non riportiamo più i risultati in un grafico a linee ma in un istogramma per il semplice fatto che non vi è più distinzione tra la lunghezza delle soluzioni, ma gli instance set sono classificati in base alla diversità degli ostacoli.

L'instance set I_1 (ovvero quello generato dall'insieme sorgente in blu e dall'insieme destinazione in giallo in Figura 4.6) è stato creato appositamente per mettere in difficoltà gli algoritmi A_1 e A_2, e così è stato come si può vedere dai risultati su I_1 in Figura 4.13(a).



(a) per tempo di esecuzione

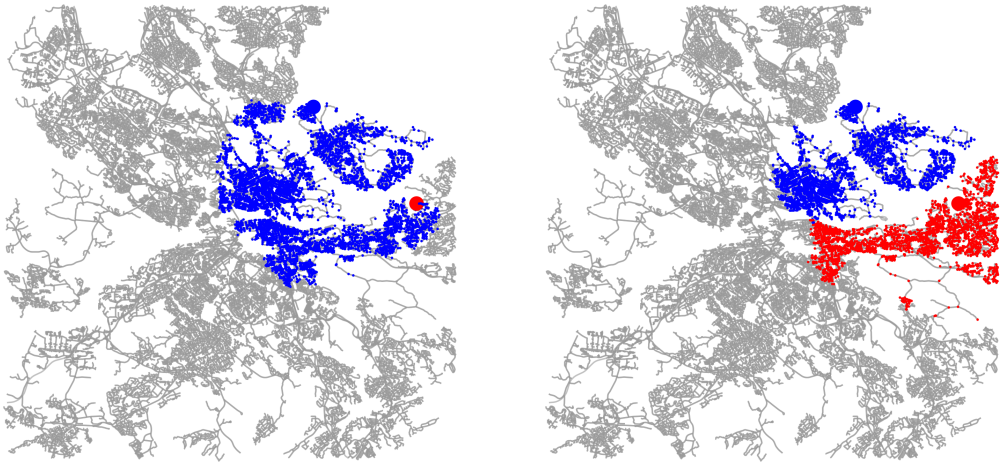
(b) per numero di nodi visitati

Figura 4.13: Analisi su Stoccolma.

In particolare vediamo dalla Figura 4.14(a) come l'algoritmo A_1 cerca di utilizzare l'informazione riguardante la posizione del nodo destinazione datagli dall'euristica per espandersi in direzione di esso che però, vista la conformazione della mappa, risulta fuorviante.

Il numero di nodi visitati in I_1 risulta comunque molto simile in ogni algoritmo (Figura 4.13(b)), però questa volta la differenza del tempo di esecuzione risulta molto più accentuata (BD_2 impiega circa la metà del tempo rispetto ad A_1) diversamente a quanto è accaduto a Kilkenny la cui differenza era minima.
Il fattore che causa questo fatto è la grande quantità di nodi visitati in questo esempio (circa 6000 contro i circa 1000 nella mappa di Kilkenny), dunque, possiamo affermare che a parità di nodi visitati, saranno gli algoritmi BD a dominare sugli A, e la differenza di tempo impiegata dai due è tanto più accentuata quanto più è elevato il numero di nodi visitati.

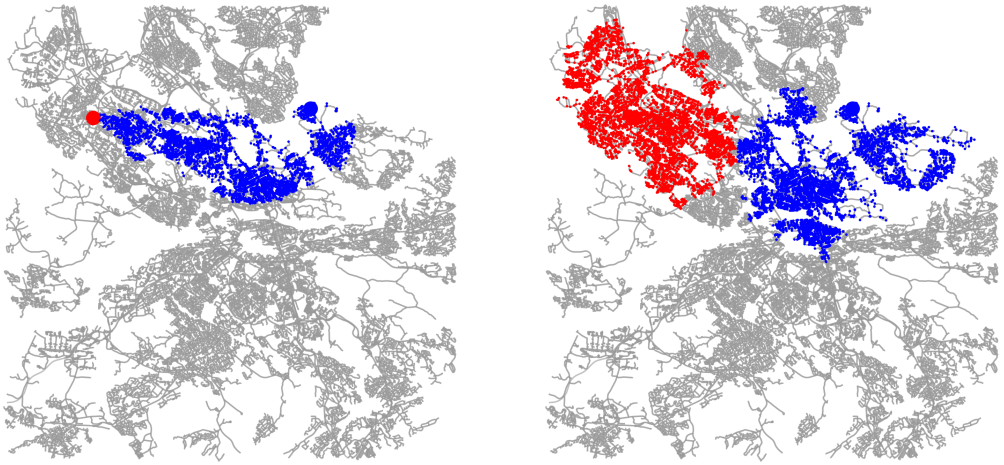
Per quanto riguarda gli instance set I_2 e I_3 possiamo notare come i ri-



(a) Nodi visitati da A_1 (b) Nodi visitati da BD_2

Figura 4.14: A_1 vs BD_2 nell'instance set I_1.

sultati siano molto simili. Prendiamo in esame quindi I_3 dove le seguenti osservazioni risultano più enfatizzate, l'analisi su I_2 sarà analoga.
In questo caso sono gli algoritmi A a prevalere sui BD. L'analisi di Figura 4.15 ci suggerisce il motivo.

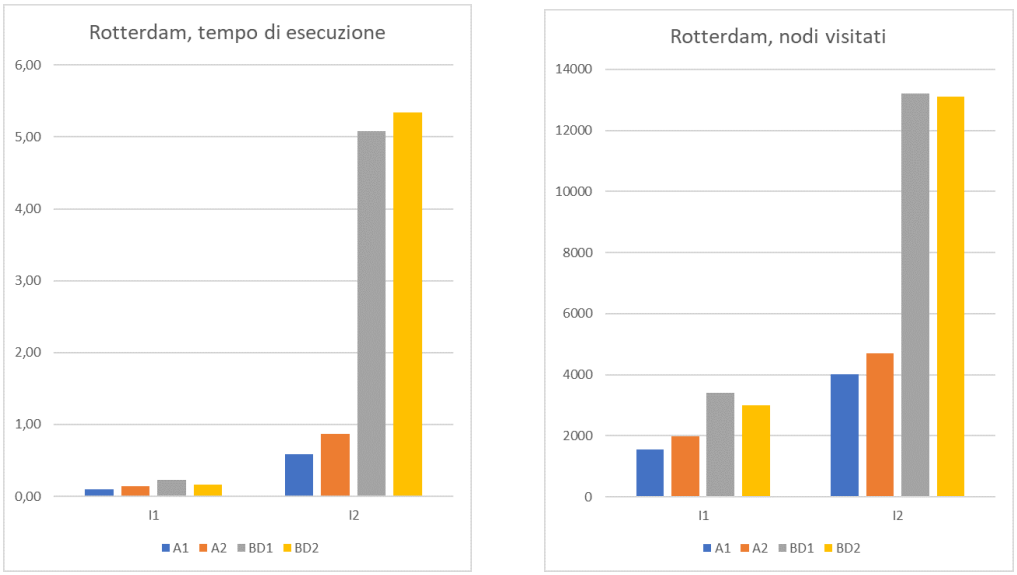


(a) Nodi visitati da A_1 (b) Nodi visitati da BD_2

Figura 4.15: A_1 vs BD_2 nell'instance set I_3.

Come vediamo, l'algoritmo A_1 non è stato "ingannato" dalla propria euristica così come lo è stato in I_1, difatti i ponti attraversati per raggiungere il nodo destinazione risultano (circa) in direzione di tale nodo. Al contrario, l'algoritmo BD_2 (i cui risultati sono quasi identici a BD_1 a causa della densità costante della mappa) sullo stesso esempio espande una quantità spropositata di nodi per trovare la soluzione (circa 10000) a dimostrazione del fatto che soluzioni molto lunghe possono penalizzare questi ultimi.

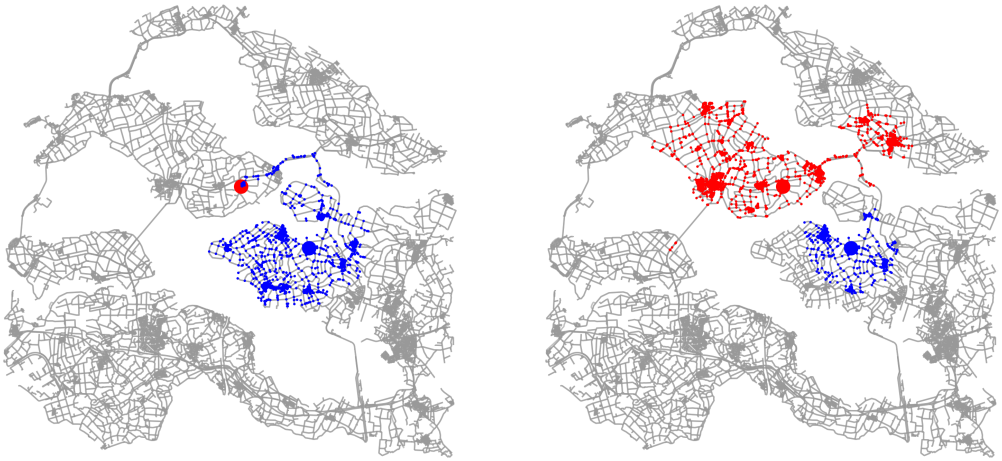
Analisi Rotterdam



(a) per tempo di esecuzione (b) per numero di nodi visitati

Figura 4.16: Analisi su Rotterdam.

Nonostante l'instance set I_1 sia composto di elementi con soluzione "corta" e nonostante la conformazione della mappa sia molto simile a quella di Stoccolma (I_1 in particolare) i risultati non sono come ci saremmo aspettati. Come vediamo dalla Figura 4.17 (a) l'algoritmo A_1 inizialmente si ritrova "incastrato" nella parte di costa geograficamente vicina al nodo destinazione (come suggerito dall'euristica) però una volta trovato il ponte, questo non

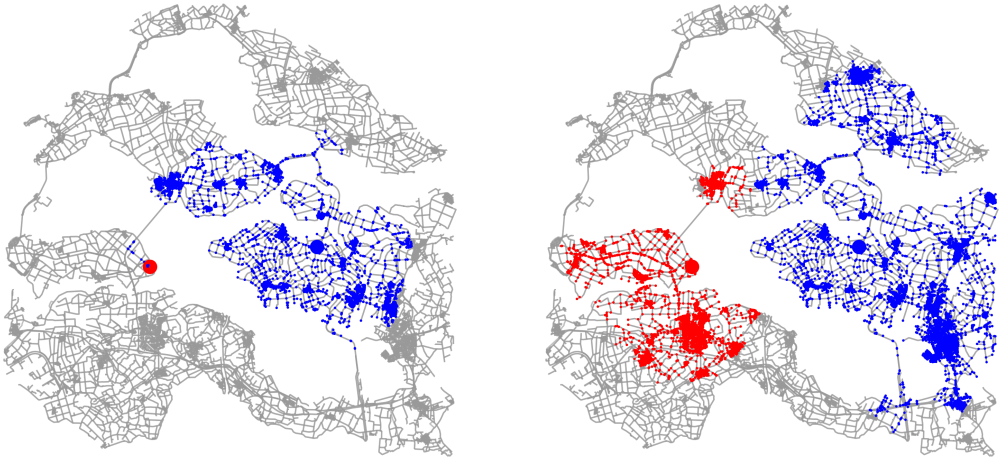


(a) Nodi visitati da A_1 (b) Nodi visitati da BD_2

Figura 4.17: A_1 vs BD_2 nell'instance set I_1.

visita **alcun** nodo al di fuori di quelli che compongono il cammino facendo risparmiare dunque molte iterazioni inutili.

Per l'instance set I_2 la situazione peggiora notevolmente come vediamo in Figura 4.18.



(a) Nodi visitati da A_1 (b) Nodi visitati da BD_2

Figura 4.18: A_1 vs BD_2 nell'instance set I_2.

La situazione è molto simile all'esempio di Figura 4.15 su Stoccolma. An-

che qui il BD_2 si ritrova a visitare quasi l'intera mappa prima di restituire la soluzione visitando fino a 12000 nodi, mentre l'algoritmo A_1, una volta uscito dall'isola di partenza (quindi una volta che l'euristica si fa più affidabile), trova facilmente la soluzione visitando una quantità limitata di nodi.

Oltretutto, in questo particolare esempio l'algoritmo BD_2 è favorito dagli "effetti di bordo" dovuti ai limiti della mappa. In particolare, possiamo vedere come la visita in avanti (in blu) dell'algoritmo BD_2 si "spalmi" sulla parte est della mappa, e una volta visitati tutti quei nodi, la velocità con cui tale visita si espande nella parte opposta aumenta[3], ma nonostante ciò il numero di nodi da esso visitati risultano troppi da poter competere con la performance di A_1.

[3]Se avessimo considerato una mappa molto più grande, l'algoritmo BD avrebbe "sprecato" le sue iterazioni visitando i nodi a est non considerati in questa mappa, impiegando molto più tempo. L'algoritmo A_1 invece non avrebbe subito alterazioni con una mappa più grande.

Conclusioni

Prima di fare qualche osservazione conclusiva è opportuno fare una piccola premessa.
I risultati ottenuti dall'analisi condotta si basano su molteplici variabili come ad esempio la potenza del pc, il linguaggio di programmazione utilizzato, l'implementazione dell'algoritmo ecc, quindi differenti combinazioni di questi parametri possono portare a risultati leggermente diversi.

Ad ogni modo, quello che emerge da questa analisi è che l'algoritmo di Dijkstra bidirezionale risulta "dispersivo" se applicato a mappe molto vaste in cui i nodi sorgente e destinazione sono posti molto distanti tra loro, nel senso che, per sua natura, "si perde" a visitare nodi anche molto lontani da quelli che compongono la soluzione. Basta pensare al problema di trovare un cammino che attraversi un'intera nazione, il Dijkstra bidirezionale si ritroverà (inevitabilmente) a visitare quasi la totalità dei nodi della mappa e oltre.
A livello cittadino invece, è bene valutare la morfologia di tale mappa andando a guardare in particolar modo la densità del grafo e la presenza di ostacoli perché, come abbiamo visto, in determinati casi è l'algoritmo bidirezionale a dominare.

Appendice A

Teoria dei grafi

Definizione 7. Un *grafo orientato* G è una coppia (N, A) di insiemi tale che $A \subseteq N \times N$.

Gli elementi di N vengono chiamati *nodi* (o *vertici*) del grafo G, mentre A è un sottoinsieme di coppie ordinate di N dette *archi* (o *lati*). Diciamo che il nodo j è adiacente al nodo i se esiste l'arco $(i, j) \in A$.
Come esempio possiamo considerare il seguente grafo G di vertici

$$N = \{0, 1, 2\}$$

e di archi

$$A = \{(0, 1), (0, 2), (1, 0), (1, 2)\}$$

Dato un grafo G, possiamo descriverlo più intuitivamente attraverso una sua *rappresentazione* (o *disegno*) sul piano. Vale a dire, segniamo n punti sul piano che stanno a rappresentare i nodi di G, disegniamo quindi, per ogni arco $(i, j) \in A$, una freccia che va da i a j .

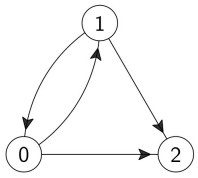


Figura A.1: Rappresentazione del grafo G.

Osserviamo che la disposizione dei vertici è del tutto irrilevante: ciò che importa è conoscere quali nodi sono in relazione e quali no.

Definizione 8. Un *grafo orientato* e *pesato* è un grafo orientato G in cui ad ogni arco è associato un numero reale tramite una funzione

$$c : A \longrightarrow \mathbb{R}$$

Il *peso* (o *costo*) dell'arco $(i, j) \in A$ è indicato con $c_{i,j}$ oppure $c(i, j)$.

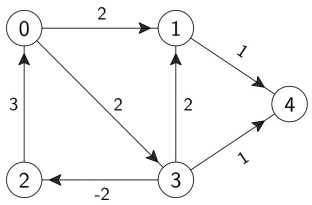


Figura A.2: Rappresentazione di un grafo orientato e pesato.

Definizione 9. Un cammino P che parte dal nodo i_0 e termina in i_n in un grafo orientato G è una sequenza ordinata di nodi $P = (i_0, i_1, \ldots, i_n)$ tali che:

- i_0 e i_n siano rispettivamente il primo e l'ultimo nodo della sequenza;
- i nodi i_k e i_{k+1} risultino adiacenti cioè

$$(i_k, i_{k+1}) \in A \quad \forall k = 0, 1, \ldots, n-1$$

Dato un cammino $P = (i_0, i_1, \ldots, i_n)$ in un grafo orientato e pesato, il suo costo c(P) è dato dalla somma dei pesi di ogni arco presente in tale cammino, ovvero

$$c(P) = \sum_{k=0}^{n-1} c(i_k, i_{k+1})$$

Alcuni cammini nel grafo G di Figura A.2 con relativo peso:

$$P_1 = [0, 3, 1, 4] \qquad c(P_1) = 5$$

$$P_2 = [0, 3, 2] \qquad c(P_2) = 0$$

Bibliografia

[1] Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin, *Network flows. Theory, Algorithms and Applications*, Prentice Hall, Upper Saddle River, New Jersey, 1993.

[2] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. *A formal basis for the heuristic determination of minimum cost paths*. IEEE Transaction of Systems Science and Cybernetics, 4(2):100–107, 1968.

[3] Andrew V. Goldberg, Chris Harrelson, Haim Kaplan, Renato F. Werneck, *Efficient Point-to-Point Shortest Path Algorithm*, 2009.

[4] M. S. Bazaraa, Hanif D. Sherali, John J. Jarvis, *Linear Programming and Network Flows*,John Wiley and Sons Inc, 2009.

[5] J. Lerner, D. Wagner, and K.A. Zweig (Eds.): *Algorithmics of Large and Complex Networks*, LNCS 5515, pp. 117–139, 2009.

[6] Ira Pohl, *Bi-Directional And Heuristic Search In Path Problems*, SLAC Report 104, Stanford Linear Accelerator Center, 1969.

[7] Geoff Boeing, *A Multi-Scale Analysis of 27,000 Urban Street Networks: Every US City, Town, Urbanized Area, and Zillow Neighborhood*, Environment and Planning B: Urban Analytics and City Science, 2018.

Ringraziamenti

Ringrazio innanzitutto la mia famiglia per aver creduto in me, e per il loro costante supporto e incoraggiamento, in particolar modo mio fratello Giacomo per l'aiuto datomi nella creazione di tutti i disegni.
Ringrazio il mio correlatore, il professor Boschetti, che mi ha aiutato nella parte implementativa, nel reperire i dati e nella stesura della tesi.
Ringrazio infinitamente Elisa, per il tempo che mi hai dedicato, perché mi sei sempre stata accanto lungo tutto il mio percorso di studi.