Università degli Studi della Campania
Luigi Vanvitelli

Scuola Politecnica delle Scienze di Base

Dipartimento di Ingegneria Industriale e dell'informazione

Tesi di Laurea Triennale in Ingegneria Informatica e Elettronica

ALGORITMI PER LA PIANIFICAZIONE DI PERCORSI OTTIMALI CON UN'APPLICAZIONE IN AMBITO MULTIMODALE

SHORTEST PATH ALGORITHMS WITH A MULTIMODAL CASE STUDY

Relatore:
Prof. Stefano Marrone
Correlatore:
Ing. Alessandro Rozza

Candidato:
Giuseppe Pinto
Matricola:
A13/000059

Anno Accademico 2016/2017

Ringraziamenti

Un primo doveroso ma soprattutto sentito ringraziamento va ai miei genitori, ai miei nonni e a mio fratello Vincenzo, senza il loro costante supporto non sarei mai riuscito a completare questo percorso.

Un ringraziamento particolare al mio relatore, il professor **Stefano Marrone** per il tempo dedicatomi in mezzo ai mille impegni e al mio correlatore **Alessandro Rozza** (Head Of Research di Waynaut) per l'inesauribile flusso di consigli.

Ringrazio tutta l'azienda **Waynaut** per avermi dato la possibilità di raggiungere questo traguardo tanto desiderato e per avermi dato sempre la massima fiducia e il massimo supporto.

Ringrazio i miei colleghi universitari **Fabio** e **Daniele** per l'infinita disponibilità che fino all'ultimo hanno continuato a dimostrare nei miei riguardi.

Un ringraziamento speciale ad **Annarita**, che ha saputo starmi accanto in questo ultimo periodo, supportandomi e sopportandomi.

Infine, ringrazio i miei amici Afragolesi che non hanno mai smesso di darmi la giusta forza e la giusta carica, nonostante i tanti chilometri che ormai ci separano.

Giuseppe Pinto

Indice generale

1. Introduzione

In questo capitolo vengono introdotti il problema ed il contesto applicativo della tesi, ossia la pianificazione ottima di percorsi basata su algoritmi su grafo. Con percorso ottimale intendiamo un cammino minimo che minimizza la funzione costo associata agli archi di un grafo dato in input. Inoltre presenteremo il contesto industriale di *Waynaut*, dove l'approccio utilizzato per la pianificazione dei percorsi ottimali risulta uno dei punti caratterizzanti i prodotti offerti da tale azienda. Infine formalizzeremo gli obiettivi della tesi e l'algoritmica che produce questo tipo di soluzioni di viaggio.

1.1 Contesto applicativo

Con la continua espansione di Internet e dei dispositivi informatici (oltre un terzo della popolazione mondiale possiederà uno smartphone nel 2017), molte aziende hanno fondato il loro business sul "mondo" *online*. Nello specifico *l'online travel* fornisce agli utenti itinerari di viaggio composti, per esempio, dal volo di andata e ritorno più l'hotel per la permanenza nella città desiderata, il tutto prenotabile su un'unica piattaforma. In questo modo si può pianificare un viaggio in pochissimo tempo. Ovviamente la concorrenza diventa sempre più forte (secondo *Euromonitor* le prenotazioni di viaggio online da *mobile* saranno il 25% delle prenotazioni di viaggio in generale entro il 2019), quindi le aziende devono essere in grado di essere sempre più competitive riducendo tempi di ricerca e prezzi proposti e fornendo soluzioni che massimizzano non solo i loro ricavi ma anche le aspettative dei clienti stessi.

Presentare soluzioni di viaggio significa pianificare un percorso ottimale su un grafo. Molte delle aziende che operano in questo mercato hanno come *core* algoritmi di *shortest path* che prendono in ingresso un grafo e calcolano il percorso "migliore" da un città di partenza a una città di destinazione in base alle richieste dell'utente, come ad esempio la durata o il costo complessivo. Nel primo caso, l'obiettivo è arrivare nel minor tempo possibile, mentre nel secondo caso diventa importante il risparmio spesso a discapito del tempo di viaggio. È proprio nel concetto di *risparmio* che si inserisce il *multimodale* soprattutto rivolto alle nuove generazioni, che sempre più frequentemente, decidono di intraprendere un viaggio cercando di minimizzarne il costo. Minimizzare il costo può significare implicitamente accettare delle soluzioni di viaggio composte da mezzi di trasporto

con caratteristiche completamente differenti: bus, car sharing, taxi, carpooling, aereo, shuttle aeroportuale, nave e treno.

L'approccio multimodale al viaggio è implicitamente utilizzato da molte persone. È infatti il multimodale che permette di partire (o di arrivare) in città che non hanno collegamenti diretti (che siano aeroporti, porti o stazioni) come per le metropoli. Basti pensare ad esempio a Matera, una città con un grande patrimonio culturale, ma senza un aeroporto nelle immediate vicinanze. In questo caso una soluzione *unimodale* (con un unico mezzo di trasporto) risulta essere estremamente limitante. Naturalmente il problema esiste anche nel caso contrario, ossia partire da Matera, dove o si sceglie di usare esclusivamente il treno oppure si può decidere di utilizzare una piattaforma multimodale dove è possibile trovare una soluzione di viaggio che combini diversi tipi di mezzo avendo così un insieme di soluzioni, tra le quali scegliere, con una cardinalità molto più ampia :

- una soluzione a basso costo, caratterizzata da un passaggio in *carpooling,* un autobus a lunga percorrenza o anche dalla combinazione dei due;
- una soluzione a bassa durata, composta da un taxi per l'aeroporto più vicino, un volo per l'aeroporto più vicino alla città destinazione e un taxi per raggiungere il punto desiderato della città di destinazione;
- una soluzione che massimizzi sia il risparmio che la durata totale del viaggio, come ad esempio, un treno dalla stazione della città di origine per l'aeroporto più vicino, un volo per un aeroporto vicino alla città di destinazione e un treno da quest'ultimo alla stazione centrale della città di destinazione.

Possono esserci anche casi in cui non esiste una soluzione unimodale, come quando si deve arrivare o partire da un'isola. Prendendo come esempio Capri e la città di Milano, creare un itinerario di viaggio tra i due punti con un unico mezzo di trasporto (soluzione unimodale) è impossibile. Infatti, come minimo, serve un volo o un treno da Milano a Napoli più un traghetto che colleghi la città di Napoli all'isola di Capri. Una piattaforma che mostri solo voli o solo treni (o più in generale che mostri solo soluzioni unimodali) non è in grado di soddisfare una richiesta di questo tipo.

È facile notare che la creazione di un percorso ottimo *multimodale,* debba tenere in considerazione molti fattori, tra i quali, quelli più importanti sono:

- la qualità della soluzione;

- la velocità con la quale la soluzione viene presentata.

Una piattaforma in grado di creare soluzioni di viaggio multimodali accuratissime ma in un tempo molto lungo sarà sicuramente fuori mercato. È quindi fondamentale scegliere un algoritmo in grado di bilanciare l'accuratezza delle soluzioni con la velocità di presentazione delle stesse, e che sappia allo stesso tempo sfruttare tutte le informazioni dei diversi tipi di mezzo di trasposto.

La classe di algoritmi, utile ai fini di creare una soluzione di viaggio multimodale, sfrutta il concetto di *cammino minimo* su un grafo orientato. Trovare il percorso minimo su un grafo consiste nell'identificare l'inseme di nodi e archi con il peso minore (data una funzione costo come ad esempio il prezzo o la durata). Gli algoritmi che appartengono a questa classe sono:

- algoritmo di Dijkstra;
- algoritmo Bellman-Ford;
- algoritmo A*;
- algoritmo Floyd-Warshal;
- algoritmo di Johnson.

La formalizzazione di cammino minimo sarà presentata nel prossimo capitolo mentre nel capitolo 3 verrà formalmente mostrato l'algoritmo A*.

1.2 Contesto industriale: Waynaut

Waynaut è una delle aziende che ha sposato il concetto di *multimodale, r*ivoluzionando l'idea di *viaggio*, essendo la prima azienda europea a creare soluzioni di viaggio composte da diversi tipi di mezzo (da quelli più tradizionali a quelli più innovativi). L'azienda ha come *mission* la progettazione di tecnologie informatiche sempre più moderne per combinare i diversi tipi di mezzi di trasporto creando soluzioni di viaggio interamente prenotabili.

La piattaforma di Waynaut in grado di generare questo tipo di soluzioni si chiama *Wayfinder*. Attualmente è in grado di aggregare i seguenti tipi di mezzo: aereo, treno, shuttle aeroportuali, carpooling, navi, traghetti, bus a lunga percorrenza e ncc (noleggio auto con conducente). Wayfinder è un sistema API *RESTfull/Json* che permette ad ogni sito o app di integrare le soluzioni multimodali e unimodali che il

suo algoritmo è in grado di generare, data una determinata richiesta in input (composta da una chiamata *POST* e da una chiamata *GET*) contenente le informazioni del viaggio desiderato:

- data di partenza ed eventualmente data di ritorno;
- coppia di coordinate (latitudine/longitudine) per il punto di partenza;
- coppia di coordinate (latitudine/longitudine) per il punto di destinazione;
- numero di passeggeri.

A questa richiesta il sistema crea un grafo, andando ad identificare tutti i possibili nodi intermedi. Questi nodi sono entità fisiche raggiungibili attraverso i mezzi di trasporto che il sistema ha a disposizione, come ad esempio: stazioni ferroviarie, città, porti ed aeroporti. Questi nodi poi vengono collegati attraverso archi che identificano i *trips* che congiungono le entità prese in esame. Per esempio, se due nodi del grafo sono delle stazioni ferroviarie allora l'arco che le congiunge è schematizzabile come un viaggio in treno (durata, orario di partenza, orario di arrivo, prezzo e tipo di treno).

Il sistema, una volta che ha completato la creazione del grafo, ha tutte le informazioni per poter calcolare le soluzioni di viaggio tra il nodo di partenza e il nodo di destinazione. Queste ultime si ottengono eseguendo l'algoritmo di ricerca A* sul grafo creato, fino ad ottenere un prefissato numero di percorsi ottimali. Ovviamente la ricerca A*, dato un grafo e i nodi per i quali si vuole calcolare il percorso, restituisce un singolo *shortest path* e non il prefissato numero di path ottenibili. Ottenere un certo numero di cammini minimi vuole dire re-iterare la ricerca A* sul grafo, andando a eliminare alcuni degli archi che sono contenuti negli *shortest path* ottenuti nelle iterazioni precedenti, riuscendo così a generare un *pool* di cammini minimi diversi tra di loro. Wayfinder, quindi, è in grado di pianificare percorsi ottimali in ambito multimodale grazie alla creazione dinamica del grafo e alle ricerche A* effettuate su di esso.

Nella figura seguente viene mostrato una *soluzione multimodale* calcolata da Wayfinder per una richiesta di soluzioni di viaggio da Milano a Matera per il 07/05/2017.

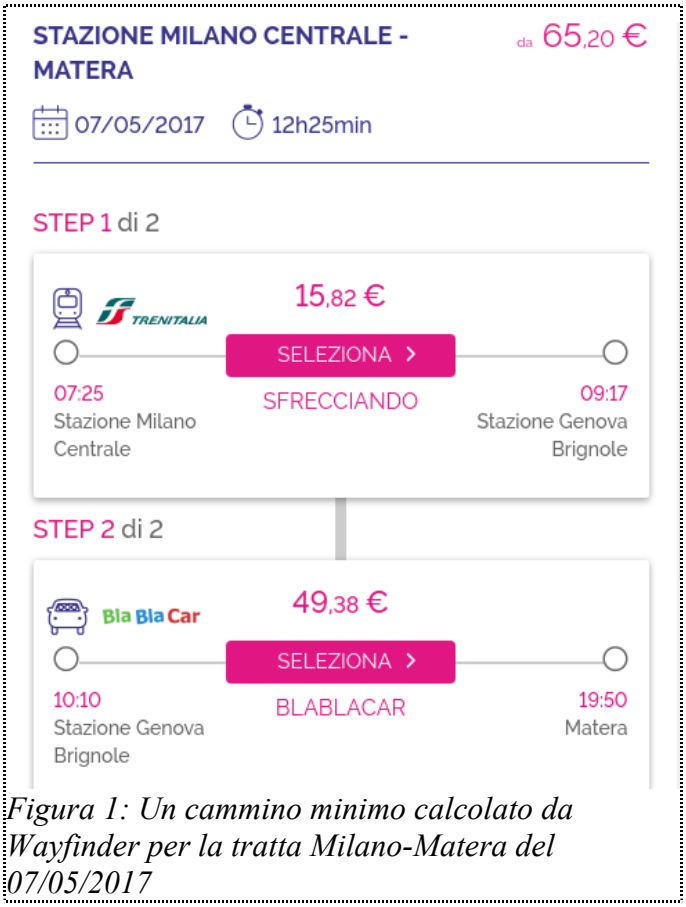

Figura 1: Un cammino minimo calcolato da Wayfinder per la tratta Milano-Matera del 07/05/2017

1.3 Obiettivi della tesi

Il presente lavoro affronta il problema della pianificazione di percorsi ottimali, intesi come soluzioni di viaggio tra due punti geo-localizzati (visti come nodi di un grafo orientato), attraverso gli algoritmi di ricerca di cammini minimi su grafo. A tale scopo, vengono presentati i concetti fondamentali della teoria dei grafi e gli algoritmi più importanti per la ricerca dei cammini su grafo, come l'algoritmo A* (pronunciato come *A star*), l'algoritmo di Dijkstra e l'algoritmo di Bellman-Ford. È importante sottolineare che l'attenzione principale di questa tesi è posta sull'algoritmo A*, presentandone la formalizzazione, un'implementazione dettagliata in pseudo-codice e un risultato calcolato su un grafo orientato di esempio.

Infine, come caso di studio reale, presenteremo la logica della piattaforma Wayfinder di Waynaut che sfrutta l'algoritmo A* per la pianificazione delle soluzioni di viaggio multimodali, focalizzando la nostra attenzione su:

- la descrizione del grafo di input per l'algoritmo (visto come una rete di trasporto) e quindi di come vengono rappresentati i nodi e gli archi che li congiungono;
- descrizione della funzione di costo per gli archi che legano i nodi del grafo di input;
- descrizione della funzione euristica che permette la creazione dei percorsi ottimali;
- la presentazione dei risultati dell'algoritmo trattando una richiesta di viaggio.

2. Pre-requisiti

Questo capitolo comprende i fondamenti della teoria dei grafi, quindi, tutte le nozioni e le definizioni fondamentali per poter capire concetti come il cammino minimo o l'*optimal path*. Di conseguenza, la presentazione di alcuni algoritmi che risolvono problemi di *shortest path* sui grafi, lasciando al terzo capitolo la presentazione dettagliata dell'algoritmo A*.

2.1 Fondamenti di teoria dei grafi

Un grafo orientato G è una coppia (V, E) dove V è un insieme non vuoto ed E una relazione binaria su V , $E \subseteq V \times V$, ossia un insieme di coppie ordinate di elementi di V . Gli elementi di V sono chiamati *vertici* o *nodi*, gli elementi di E sono chiamati *archi* (indicati con una coppia di lettere tra parentesi tonde). Nel caso in cui invece il grafo non è orientato gli elementi di E sono detti *spigoli* (indicati con una coppia di lettere tra parentesi graffe) . Se l'insieme V è finito, il grafo viene detto *finito*. Nella *Figura 2* un esempio di grafo finito e orientato.

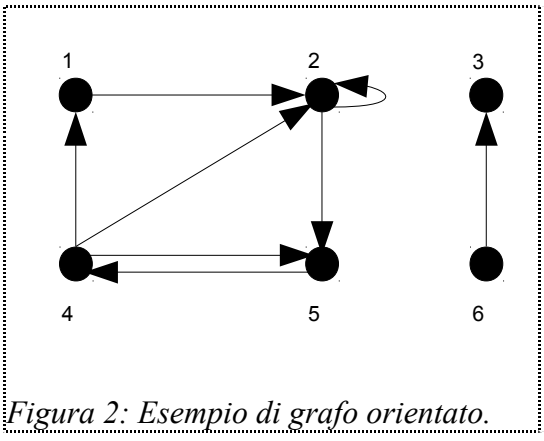

Figura 2: Esempio di grafo orientato.

Un multigrafo orientato è un grafo orientato $G=(V, E)$ che ha archi multipli, cioè due nodi sono estremi di più archi. In tal caso si dice che E è un multiinsieme, in altre parole, i suoi elementi hanno una *molteplicità*.

Se (*u*,*v*) è un arco di un grafo orientato, si dice che (*u*,*v*) è *incidente da u* o che *lascia u* ed è *incidente a v* o che *entra* nel vertice *v*. Inoltre è anche ammesso dire che il vertice *v* è *adiacente* al vertice *u*. Ovviamente se gli archi non hanno relazione di incidenza il grafo *non è orientato* e viene indicato semplicemente con la terminologia "grafo".

Se (*u*,*v)* è un arco di un grafo, si dice che (*u*,*v)* è *incidente nei vertici u* e *v*. Inoltre è ammesso dire che i vertici *u* e *v* sono *adiacenti* o *confinanti*.

Sia $G=(V,E)$ un grafo o un multigrafo. Il grado di un vertice $x\in V$ in G è il numero, $d_G(x)$, di archi incidenti con esso. Se $d_G(x)=0$, x si dice isolato; se $d_G(x)=1$, x è detto vertice pendente. Con queste informazioni possiamo enunciare il seguente teorema:

Primo teorema della teoria dei grafi:

In un grafo o multigrafo finito si ha:

1. $2|E|=\sum_{x\in V} d_G(x)$.
2. Il numero di vertici di grado dispari è pari.

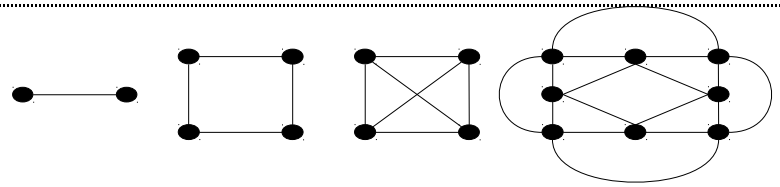

Figura 3: I grafi di questa figura sono regolari di grado rispettivamente 1, 2, 3, 4.

Un grafo o multigrafo i cui vertici hanno tutti lo stesso grado *d* si dice *regolare di grado d* (*Figura 3*). Un grafo finito regolare di grado *d* con *n* vertici ha $\frac{nd}{2}$ archi.

Un grafo si dice *completo* se ha tutti gli archi possibili, quindi un grafo completo con *n* vertici è regolare di grado *n*-1 e viene indicato con K_n .

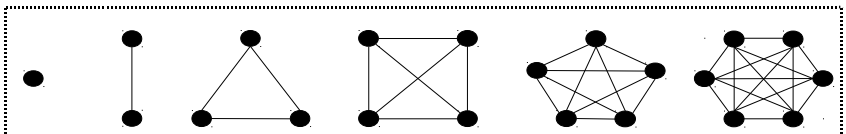

Figura 4: Sei grafi completi.

Un grafo completo con *n* vertici ha esattamente $\binom{n}{2}=\frac{n(n-1)}{2}$ archi.

Un multigrafo, invece, si dice *completo* e di indice λ se ha tutti gli archi possibili, ognuno ripetuto λ volte. Un multigrafo completo con *n* vertici ed indice λ è regolare di grado $\lambda(n-1)$ e viene indicato con λK_n . Un multigrafo completo con *n* vertici ed indice λ ha esattamente $\lambda\binom{n}{2}=\frac{\lambda n(n-1)}{2}$ archi.

Considerando il grafo $G=(V,E)$ si dice *sottografo* il grafo $H=(W,F)$ tale che $W\subseteq V$ e $F\subseteq E$. Nella *Figura 5* un esempio di sottografo.

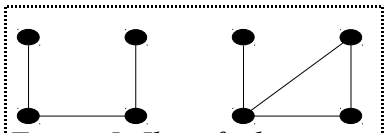

Figura 5: Il grafo di sinistra è sottografo di quello di destra.

Considerando il grafo $G=(V,E)$, si chiama cammino (*walk*) di lunghezza *t* e di estremi u_1 e u_{t+1} (o che congiunge i vertici u_1 e u_{t+1}) una sequenza:

$\langle u_1, e_1, u_2, e_2, \ldots, u_t, e_t, u_{t+1}\rangle$,

dove $t\geq 0$, u_i è un vertice di V ed ogni e_i è l'arco $\{u_i, u_{i+1}\}$. Un cammino si dice chiuso se $u_1=u_{t+1}$, cioè il primo e l'ultimo vertice coincidono.

Per $t=0$ il cammino è costituito da un unico vertice $\langle u_1\rangle$, ed è un cammino di lunghezza zero. Spesso elencare gli archi di un cammino è superfluo, così si può anche adottare anche la seguente forma: $\langle u_1, u_2, \ldots, u_t, u_{t+1}\rangle$.

Inoltre un cammino può avere sia archi che vertici ripetuti.

Al cammino sono assegnate definizioni in base alle sue proprietà:

- un cammino si dice *semplice* se non contiene archi ripetuti;
- un cammino si dice *elementare* se non contiene vertici ripetuti con l'eventuale eccezione del primo e dell'ultimo vertice (che potrebbero essere uguali);
- un cammino elementare che ha tutti i vertici distinti (cioè non chiuso) si dice *path*;
- un cammino elementare chiuso si dice *ciclo*, *d*ove la *lunghezza* di un ciclo è il numero dei suoi archi. Un grafo senza cicli è detto *aciclico*.

Con il concetto di cammino si possono estendere ulteriori proprietà per il grafo.

Un grafo $G=(V,E)$ si dice *connesso* se, comunque presi due vertici $x, y\in V$, esiste almeno un cammino che li congiunge (cioè di estremi *x* e *y*). Un grafo che non è connesso viene detto *sconnesso*. Su un grafo connesso possiamo definire la

distanza di due vertici *x* e *y* come la lunghezza minima dei cammini di estremi *x* e *y*. Se $a \in V$ l'insieme C_a , formato da tutti i vertici $x \in V$ per i quali esiste un cammino da *a* a *x*, si dice *componente connessa* di *a*. Di conseguenza il *numero di connessione* di un grafo è il numero delle sue componenti connesse.

Considerando $G=(V,E)$ come grafo connesso, allora per ogni coppia $u,v \in V, u \neq v$, esiste un *path* di estremi *u* e *v*.

Il concetto di *sottografo* può essere esteso anche a grafi orientati in questo modo:

Sia $G=(V,E)$ un grafo orientato. Il grafo orientato $J=(W,B)$ si chiama *sottografo orientato* di G se $W \subseteq V$ e $B \subseteq E$. Se B coincide con tutti gli archi di E aventi entrambi i vertici (quello iniziale e finale) in W, possiamo dire che $J=(W,B)$ è un *sottografo orientato indotto*.

Sia $G=(V,E)$ un multigrafo orientato. Se si sostituisce ogni arco $(a,b) \in E$ con lo spigolo $\{a,b\}$ si ottiene un multigrafo $|G|$ chiamato *underlying multigrafo* di G. Se, inoltre, in $|G|$ si sostituiscono gli eventuali spigoli paralleli con un solo spigolo si ottiene un grafo (G) chiamato *underlying grafo*.

Considerando $G=(V,E)$ come multigrafo, si chiama *orientazione* di G un qualsiasi multigrafo orientato H tale che $|H|=G$. Se ogni spigolo non orientato $\{a,b\} \in E$ si sostituisce con i due archi orientati (*a*,*b*) e (*b*,*a*) si ottiene il multigrafo orientato G, detto l'*orientazione completa* di G.

Nei grafi orientati la nozione di *cammino* può avere differenti interpretazioni a seconda se si vuole che gli archi siano tutti orientati nelle stessa direzione o no. Nel primo caso si parla di *cammini orientati*. Le varie definizioni di cammino date per un grafo possono ripetersi in modo del tutto analogo per un multigrafo. Quindi sia $G=(V,E)$ un grafo orientato, la sequenza : $\langle u_1, e_1, u_2, e_2, \ldots, u_t, e_t, u_{t+1} \rangle$, dove con e_i si indica o l'arco (u_i, u_{i+1}) oppure (u_{i+1}, u_i) , si chiama *cammino*, *cammino semplice*, *cammino elementare*, *path* o *ciclo* se la sequenza ottenuta sostituendo nella sequenza mostrata appena sopra $\{u_i, u_{i+1}\}$ al posto di e_i è rispettivamente un *cammino*, *cammino semplice*, *cammino elementare*, *path* o *ciclo* nell'*underlying multigrafo* $|G|$.

Inoltre per un grafo orientato possiamo esprimere il concetto di connessione, e quindi un grafo orientato si dice connesso se per ogni paio di vertici *u* e *v* esiste (almeno) un cammino di estremi *u* e *v*.

Nel caso in cui si considera un grafo orientato con cammini dotati di archi orientati nella stessa direzione, si ottengono le stesse definizioni ottenute nel caso di grafi non orientati aggiungendo ovviamente la parola "orientato" (es. *cammino chiuso orientato*, *cammino semplice orientato*, *cammino elementare orientato* ecc.).

Un grafo orientato si dice *fortemente connesso* se per ogni paio di vertici *u* e *v* esiste un cammino orientato che va da *u* a *v* e un cammino orientato che va da *v* ad *u*, cioè se *v* è raggiungibile da *u* e viceversa. Per un grafo orientato fortemente connesso G si chiama *distanza dal vertice x al vertice y*, indicata con $d_G(x, y)$, la lunghezza minima dei cammini orientati che vanno da *x* a *y*.

I grafi connessi e aciclici appartengono ad una classe di grafi detta *alberi*, molto importante nei problemi informatici. Di seguito un teorema che esplicita le caratteristiche di un albero.

Teorema: Sia $G=(V,E)$ è un grafo, le seguenti affermazioni sono equivalenti:

1. *G è un albero;*
2. *G è aciclico, ma se si aggiunge un qualsiasi arco si forma un ciclo;*
3. *Comunque presi* $u, v \in V$ *, esiste un unico path di estremi u e v;*
4. *G è connesso, ma eliminando un qualsiasi arco di G si perde la connessione.*

2.2 Algoritmi per la ricerca di cammini minimi

Il linguaggio dei grafi permette di rappresentare in modo semplice la struttura di molti problemi applicativi, consentendo, in molti casi di grande importanza, di costruire metodi razionali di soluzione dei problemi stessi. Fra i problemi più importanti, più semplici e più antichi, per la cui soluzione sono utilizzate rappresentazioni basate su grafi, vi sono i problemi di ricerca di cammini minimi.

Per *shortest path* si intende il problema del cammino minimo tra due vertici, ossia il percorso che collega due nodi dati e che minimizza la somma dei costi associati all'attraversamento di ciascuno arco. Più formalmente:

Sia $G=(V,E)$ un grafo orientato con funzione costo $w: E \rightarrow \mathbb{R}$ che associa ad ogni arco un peso a valore nei reali.

Il peso di un cammino $p=\langle v_0, v_1, ..., v_k \rangle$ è la somma dei pesi degli archi che lo costituiscono: $w(p)=\sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Il peso di cammino minimo da *u* a *v* è definito come:

$$\begin{cases} \delta(u, v)=min\{w(p)\} , & \text{se esiste almeno un cammino tra i nodi} \\ \delta(u, v)=\infty & \text{altrimenti} \end{cases}$$

Un cammino minimo dal vertice *u* al vertice *v* è definito quindi come un qualunque cammino *p* con peso $w(p)=\delta(u, v)$.

Gli algoritmi di ricerca di cammini minimi sfruttano la proprietà, detta di *sottostruttura,* che un cammino minimo tra due nodi contiene al suo interno altri cammini minimi. Questa proprietà può essere formalmente definita con un lemma e un corollario:

Lemma: Dato un grafo $G=(V, E)$ orientato e pesato con funzione peso $w: E \rightarrow \mathbb{R}$, sia $p=\langle v_1, v_2, ..., v_k \rangle$ un cammino minimo dal nodo v_1 al nodo v_k , e per ogni *i* e *j* tali che $1 \leq i \leq j \leq k$, sia $p_{ij}=\langle v_i, v_{i+1}, ..., v_j \rangle$ il sotto-cammino di *p* dal nodo v_i al nodo v_j . Allora p_{ij} è un cammino minimo da v_i a v_j .

Corollario: Sia $G=(V, E)$ un grafo orientato e pesato con funzione peso $w: E \rightarrow \mathbb{R}$. Si supponga che un cammino minimo *p* da un nodo sorgente *s* ad un nodo *v* possa essere decomposto in un cammino *p** con nodo sorgente *s* e un qualunque nodo destinazione *u* più un altro cammino dal nodo *u* al nodo *v*. Allora il peso del cammino minimo da *s* a *v* è $\delta(s, v)=\delta(s, u)+w(u, v)$.

Dati due vertici *s* e *v*, si dice che *v* è *raggiungibile* da *s* se esiste almeno un cammino minimo da *s* a *v*. Bisogna precisare però che la raggiungibilità non assicura l'esistenza di un cammino minimo tra due nodi. Il problema dei cammini minimi può non avere soluzioni ammissibili , nel caso in cui non esiste un cammino orientato che vada dal nodo sorgente al nodo destinazione. La presenza in *G* di archi negativi può indurre in *G* la presenza di cicli con peso negativo (ad esempio cammini chiusi, dove la somma dei pesi degli archi del cammino ha valore negativo). Quindi dato un grafo $G=(V, E)$ diretto e pesato, se dal nodo *s* sono raggiungibili nodi appartenenti a cicli di peso negativo, allora esiste un nodo *v* per cui esistono cammini da *s* a *v* di peso arbitrariamente piccolo. In altri termini se

esiste un ciclo orientato C, tale che $p(C)<0$ (peso negativo), il problema è illimitato inferiormente. Questo è dovuto al fatto che più volte si decide di percorrere un ciclo con peso negativo più il cammino risultante risulta avere un peso minore. Basti guardare la figura seguente, nella quale il nodo sorgente può raggiungere nodi che si trovano in un ciclo con peso negativo.

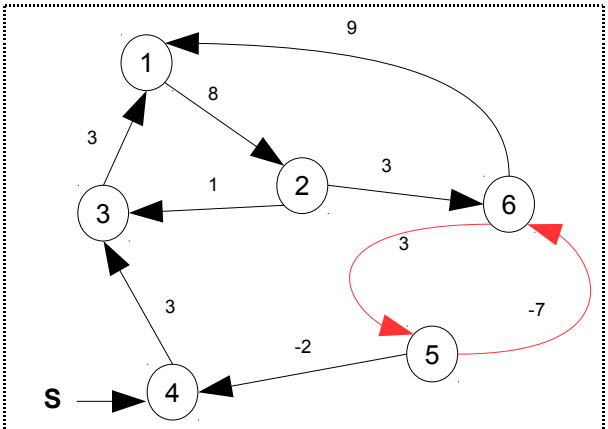

Figura 6: Grafo con ciclo negativo.

Al contrario, se in un grafo nessun vertice appartenente a cicli di peso negativo è raggiungibile dal vertice s, il fenomeno descritto non si verifica e la raggiungibilità di v da parte di s è condizione necessaria e sufficiente perché il cammino esista: dato un grafo G diretto e pesato, se dal nodo s non sono raggiungibili vertici appartenenti a cicli di peso negativo, allora i nodi v del grafo raggiungibili da s hanno cammini minimi di lunghezza inferiore a $|V|$. Il tutto visibile nella figura che segue:

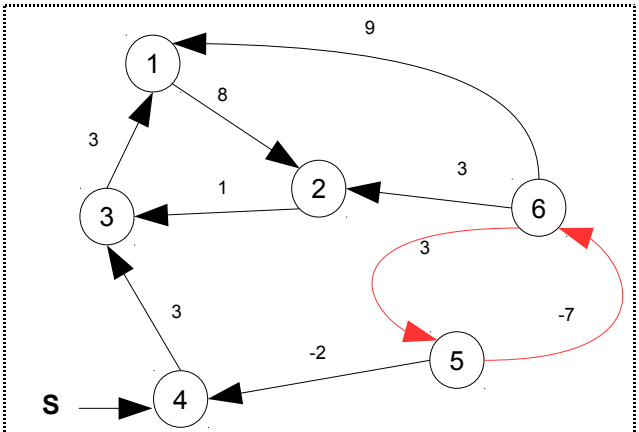

Figura 7: Esempio di grafo che garantisce la raggiungibilità.

Ci sono, inoltre, tante varianti varianti al problema dei cammini minimi:

- *cammino minimo tra due nodi*: dati due nodi *s* e *t* trovare il cammino minimo da *s* a *t*, anche indicato con la dicitura inglese *single-source single-destination shortest path problem*;
- *cammino minimo con una sorgente*: dato un nodo sorgente *s*, trovare per ogni nodo del grafo *t*, un cammino minimo da *s* a *t*, anche indicato con la dicitura inglese *single-source shortest path problem*;
- *cammino minimo con una destinazione*: data una destinazione *t*, trovare, per ogni nodo del grafo *s*, un cammino minimo da *s* a *t*, anche indicato con la dicitura inglese *single-destination shortest path problem*;
- *cammino minimo per tutte le coppie di nodi*: per ogni coppia di nodi *s*,*t* del grafo trovare il cammino minimo da *s* a *t*, anche indicato con la dicitura inglese *all-pairs shortest path problem*.

Il peso degli archi è una variabile molto importante in questi problemi perché può essere modellata in tanti modi, ponendo quindi l'interesse sulla grandezza che si vuole minimizzare. Per esempio il peso può essere inteso come una distanza, come un tempo o come un costo. In letteratura ci sono tanti algoritmi che risolvono il problema dei cammini minimi, dove ognuno ha le sue peculiarità e i suoi limiti. Gli algoritmi più importanti sono sicuramente i seguenti:

- algoritmo di Dijkstra di tipo *single-source*;
- algoritmo di Bellman-Ford di tipo *single-source*;
- algoritmo A* (trattato in dettaglio al capitolo 3) di tipo *single-source single-destination.*

Molti algoritmi affrontano il problema dello shortest path non determinando uno dei cammini minimi da un nodo sorgente a un nodo destinazione, ma presentando in output l'albero dei cammini minimi, cioè un sottografo del grafo di partenza *G,* che è un albero i cui nodi includono il nodo sorgente e tutti i nodi da esso raggiungibili con un cammino orientato e tale che l'unico cammino dalla sorgente ad ogni altro nodo dell'albero sia un cammino minimo tra i due. La *Figura 8* mostra un grafo con una sorgente e i relativi cammini minimi per i suoi nodi, mentre la *Figura 9* mostra l'albero dei cammini minimi del grafo considerato.

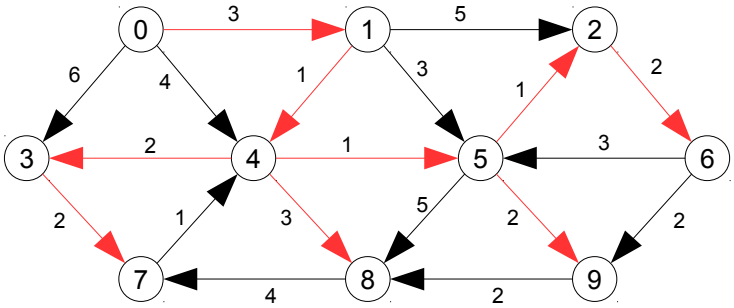

Figura 8: Grafo con sorgente in 0 e relativi cammini minimi.

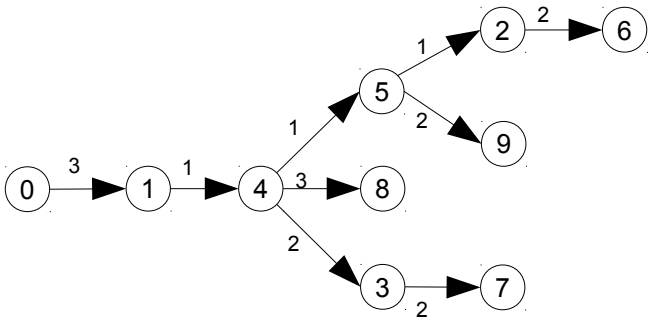

Figura 9: Albero dei cammini minimi per il grafo di Figura 8.

L'algoritmo di Dijkstra e quello di Bellman-Ford utilizzano questo tipo approccio.

L'algoritmo di Dijkstra è stato proposto nel 1959 da Edsger W. Dijkstra, l'algoritmo risolve problemi di cammini minimi con singola sorgente su di un grafo orientato e pesato nel caso particolare in cui tutti i pesi degli archi siano non negativi, costruendo l'albero dei cammini minimi, quindi tutti gli *shortest path* che congiungono il nodo di origine con tutti i nodi del grafo. L'algoritmo mantiene un'etichetta *dist[i]* per i nodi del grafo, che vengono quindi partizionati in due insiemi: quello dei nodi etichettati *permanentemente* e quello dei nodi etichettati *temporaneamente*. L'etichetta dei nodi permanenti rappresenta la distanza del cammino minimo dalla sorgente a tali nodi, mentre quella dei nodi temporanei presenta un valore che può essere maggiore o uguale al valore del cammino minimo. L'algoritmo parte dalla sorgente e etichetta i nodi successori. La prima fase è una fase di inizializzazione dove la distanza del nodo sorgente viene posta a zero e le altre distanze, relative agli altri nodi, vengono settate a un valore arbitrariamente alto. L'algoritmo seleziona il nodo che ha il valore dell'etichetta più basso tra quelli

etichettati temporaneamente e setta l'etichetta del nodo preso in esame da temporanea a permanente, aggiornando poi tutte le etichette dei nodi a lui adiacenti. Quindi le fasi dell'algoritmo sono principalmente tre: *inizializzazione*, *selezione del nodo* e *aggiornamento delle distanze*. Di seguito una procedura in pseudo-codice dell'algoritmo che ha come input un grafo e un nodo che è la sorgente.

```
Function Dijkstra(grafo, source){

      for each nodo v contenuto in grafo :
            dist[v] ← ∞ ;
            precedente[v] ← non definita;
      end for

      dist[source] ← 0;
      precedente[source] ← 0
      Q ← tutti i nodi del grafo;

      while Q != Ø :
            u ← nodo di Q con minior valore in dist[];
            rimuovi u da Q;

            for each nodo adiacente v di u :
                  alt ← dist[u] + distanza(u, v);
                  if alt < dist[v] :
                        dist[v] ← alt;
                        precedente[v] ← u;
                        riordina v in Q;
                  end if
            end for
      end while

      return dist; //etichette per ogni nodo del grafo
}
```

L' algoritmo di Bellman-Ford fornisce la soluzione al problema dei cammini minimi su di un grafo $G=(V,E)$ orientato e pesato, senza la limitazione di peso positivo per gli archi, dal nodo origine verso tutti gli altri nodi o individua un ciclo di costo negativo e quindi la non esistenza dell'albero dei cammini minimi. L'assunzione dell'algoritmo di Bellman-Ford è che un cammino dalla sorgente s ad un nodo di destinazione t deve contenere al più $n=|V|$ nodi e $n-1$ archi. Analogamente nel caso in cui ci siano circuiti negativi, ci sarà sicuramente un cammino che congiunge i due vertici passando per un numero di nodi al più pari a n e n archi. L'algoritmo determina in modo ricorsivo i cammini minimi con origine in s imponendo che questi percorsi siano, all'inizio, composti al più da un solo nodo (zero archi), permettendo così ad ogni passo l'aggiunta di un nodo e quindi di un

arco. Di conseguenza non va oltre l'*n-esima* iterazione, se il cammino diminuisce ulteriormente per l'esecuzione di un passo successivo allora siamo in presenza di un percorso che ha un loop negativo. Di seguito una funzione in pseudo-codice che implementa l'algoritmo. La funzione ritorna *FALSE* nel caso in cui il grafo contiene un ciclo con peso negativo, in caso contrario termina con *TRUE* e gli elementi contenuti nella struttura *p[]* permettono di ricostruire l'albero dei cammini minimi.

```
Function BellmanFord(grafo, source){

      //fase di inizializzazione
      for each nodo v in grafo :
            d[v] ← ∞ ;
            p[v] ← non definita;
      end for

      d[source] ← 0;

      //fase di rilassamento degli archi
      for i ← 1 to |grafo|−1 :
            for each arco (u,v) in grafo:
                  if d[v] > d[u] + distanza(u,v) :
                        d[v] ← d[u] + distanza(u,v);
                        p[v] ← u;
                  end if
            end for
      end for

      for each arco (u,v) in grafo:
            if d[v] > d[u] + distanza(u,v) :
                  return FALSE;
            end if
      end for

      return TRUE;
}
```

3. Algoritmo A*

Questo capitolo tratta il dettaglio dell'algoritmo A* dando enfasi ai concetti teorici e quindi alla formalizzazione dello stesso, utilizzando i concetti presentati nel capitolo precedente. Fornendo, inoltre, un esempio di implementazione in pseudo-codice e considerazioni tecnologiche, lasciando al capitolo successivo l'applicazione dell'algoritmo al problema Waynaut.

3.1 Formalizzazione

L'A* è l'algoritmo più popolare per la ricerca dei cammini minimi su grafi orientati pesati o non pesati, $G=(V,E)$ dove è sempre noto il costo che separa due nodi adiacenti, per la sua flessibilità e per la sua completezza (trovando sempre una soluzione al problema, se esiste) e sopratutto per il fatto che può essere utilizzato in molti contesti applicativi (per esempio su un grafo dove i vertici rappresentano gli aeroporti e gli archi sono i voli per raggiungerli, riuscendo quindi a trovare uno *shortest trip* tra due nodi). Risulta essere così famoso perché utilizza un approccio simile all'algoritmo di Dijkstra favorendo i nodi vicini al punto di partenza o sorgente e, allo stesso tempo un approccio *best-first-search* favorendo i nodi che sono vicini al goal o nodo di destinazione. Inoltre utilizza il concetto di euristica che "guida" la ricerca dei cammini minimi.

Quando si parla di A*, con il termine $g(n)$ viene indicato il *costo esatto* del cammino dalla sorgente al nodo n del grafo (quindi la somma di tutti i costi degli archi che legano i due nodi) e con il termine $h(n)$ il *costo della stima euristica* dal nodo n al nodo goal (quindi la somma dei costi stimati tra i due nodi), il nodo n viene anche chiamato nodo *corrente*. L'algoritmo è in grado di bilanciare queste due grandezze (calcolate entrambe con la stessa unità di misura) mentre si sposta per arrivare al nodo goal e ogni volta che deve decidere quale nodo n esaminare, fa una scelta in base al valore più basso della funzione $f(n)=g(n)+h(n)$.

Di conseguenza ogni nodo n del grafo considerato è caratterizzato dalle seguenti proprietà:

- un valore di posizionamento all'interno del grafo (per esempio una coppia di coordinate come la latitudine e la longitudine);
- una referenza al nodo precedente che si indica con *n.parent*;

- il valore della funzione $g(n)$;
- il valore della funzione $h(n)$;
- il valore della funzione $f(n)$.

Il valore della funzione di costo $g(n)$ è semplicemente il costo incrementale del movimento, e quindi degli archi, dal nodo di partenza al nodo corrente n , calcolato attraverso la seguente equazione:

$$g(n)=g(n.parent)+cost(n.parent,n)$$

dove la funzione $cost(n_1, n_2)$ restituisce il costo e quindi il peso dell'arco che congiunge i due nodi di input. Ovviamente per il nodo sorgente il valore della funzione $g(n)$ è pari a zero.

Il valore della funzione euristica $h(n)$ rappresenta la stima della distanza tra ogni nodo n del grafo e il nodo goal. È fondamentale scegliere una buona funzione euristica vista la sua importanza nel controllare le scelte dell'algoritmo e anche perché essa determina il tempo complessivo dell'esecuzione, come dimostrano le ipotesi seguenti:

- nel caso in cui la funzione $h(n)$ è una funzione identicamente nulla, solo la $g(n)$ ha un ruolo nel guidare l'A*, facendolo quindi comportare come l'algoritmo di Dijkstra;
- nel caso in cui $h(n)$ ha sempre valori più bassi del costo del *path* tra il nodo n e il nodo goal, allora l'algoritmo deve *espandere* tanti nodi per arrivare a calcolare lo *shortest path*, arrivando quindi a soluzione lentamente;
- nel caso in cui $h(n)$ è identica al costo del *path* dal nodo n al nodo goal, allora l'algoritmo esaminerà solo i nodi che saranno del *best path* senza espandere ulteriori nodi. Purtroppo non è possibile trovarsi sotto questa ipotesi in ogni occasione;
- nel caso in cui $h(n)$ alcune volte ha valori maggiori del costo del *path* dal generico nodo n al nodo goal, allora l'A* non garantisce lo *shortest path*, favorendo però la velocità di esecuzione;
- nel caso in cui la funzione $h(n)$ presenta valori elevati per ogni nodo, allora essa stessa è l'unica ad avere un ruolo nel guidare l'algoritmo, facendolo quindi ricadere nella categoria *greedy best-first-search* (dove l'algoritmo

viene "ingolosito" dai nodi con l'alto potenziale dovuto agli alti valori dell'euristica).

L'euristica determina di conseguenza la qualità della soluzione finale. Con un euristica *ammissibile* l'algoritmo è in grado di determinare la soluzione ottima o l'*optimal path* (percorso con il minor costo possibile). Un'euristica si dice ammissibile quando l'errore di stima non è mai in eccesso. In termini matematici una funzione euristica *h* è ammissibile se:

$$\forall x \in V : h(goal, x) \leqslant g(goal, x)$$

dove *V* è l'insieme dei nodi del grafo e la funzione *g*, come enunciato prima, rappresenta il costo esatto tra i nodi in input.

Per semplificare ulteriormente la struttura dell'algoritmo e per avere una condizione ancora più forte dell'ammissibilità, per le ricerche su grafo si considera una condizione di *consistenza* euristica che è una condizione di forte ottimalità. Una funzione euristica si dice consistente quando è monotona e quindi quando:

$$\forall (x, y) \in E : h(goal, x) \leqslant g(x, y) + h(goal, y)$$

inoltre una funzione euristica consistente è anche ammissibile e non è vero sempre il contrario.

Ovviamente l'implementazione della funzione euristica dipende molto dalle proprietà del grafo sul quale viene lanciato l'A*. Di seguito quelle più utilizzate sui grafi di tipo *grid*:

- *Manhattan distance*;
- *Diagonal distance;*
- *Euclidean distance.*

La Manhattan distance è l'euristica più semplice e viene utilizzata quando si lavora su grafi (schematizzabili come griglie) dove per passare da un nodo all'altro è permesso solo fare quattro tipi di movimento: su, giù, sinistra e destra. Guardando la funzione costo scelta, con la lettera *d* si intende il minimo tra tutti i costi dei movimenti tra due nodi adiacenti, l'euristica su un grafo a griglia può essere calcolata come *d* volte la *Manhattan distance*:

$$h(n) = d * (|n.x - goal.x| + |n.y - goal.y|)$$

Di seguito una figura dove viene eseguito l'A* utilizzando l'euristica appena enunciata.

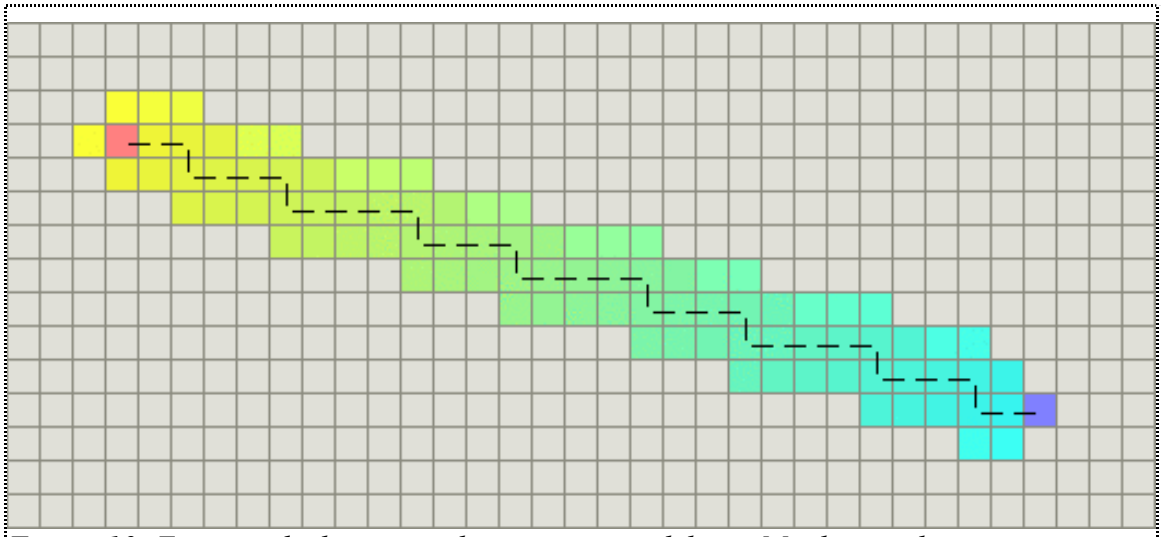

Figura 10: Esempio di shortest path con euristica del tipo Manhattan distance.

La Diagonal Distance si differenzia dalla *Manhattan* per il fatto che essa può essere utilizzata sui grafi dove sono ammissibili movimenti in diagonale. Anche in questo caso viene utilizzato il costo *d* già enunciato per la Manhattan distance e in aggiunta si indica con d_{diag} il costo del movimento in diagonale da ogni nodo del grafo. Questo tipo di euristica viene calcolata con le seguenti equazioni:

$$dx = |n.x - goal.x|$$

$$dy = |n.y - goal.y|$$

$$h(n) = d * (dx + dy) + (d_{diag} - 2 * d) * min(dx, dy)$$

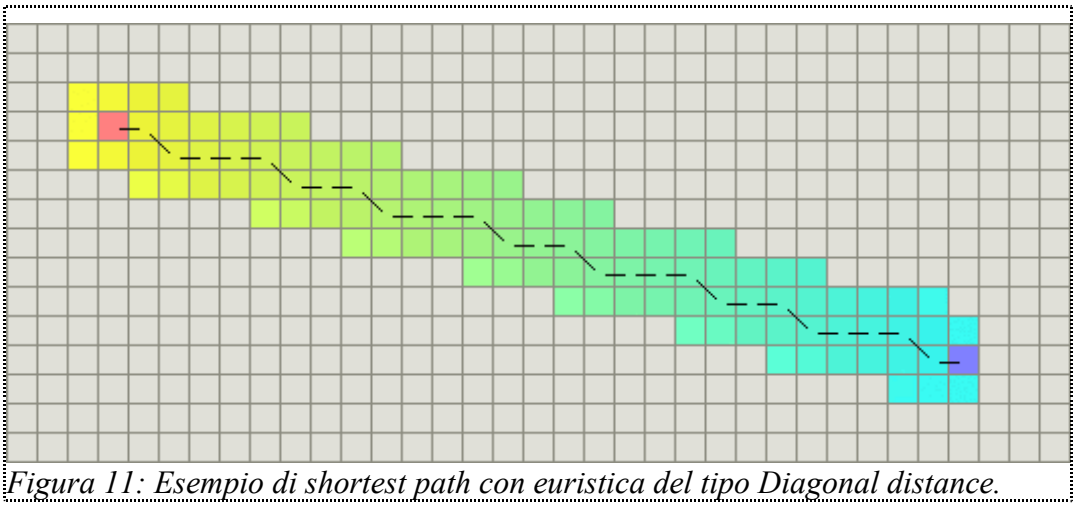

Figura 11: Esempio di shortest path con euristica del tipo Diagonal distance.

Nella *Figura 11* viene eseguito l'A* utilizzando l'euristica di tipo *diagonal*.

L'*Euclidean distance* viene utilizzata quando l'algoritmo può decidere di muoversi in ogni direzione del grafo e viene calcolata con le seguenti equazioni:

$$dx = |n.x - goal.x|$$

$$dy = |n.y - goal.y|$$

$$h(n) = d * \sqrt{dx^2 + dy^2}$$

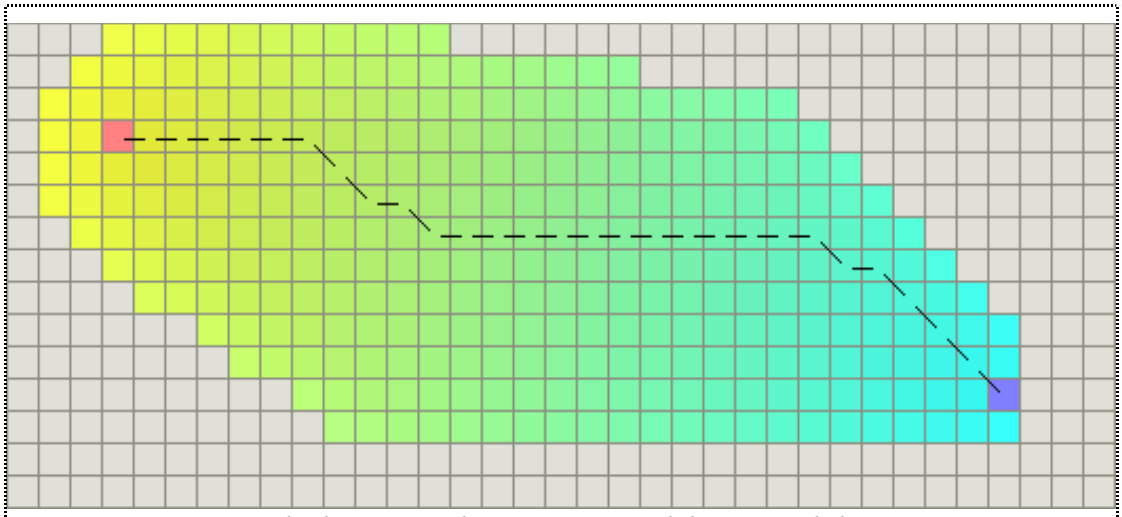

Figura 12: Esempio di shortest path con euristica del tipo Euclidean Distance.

Siccome l'*Euclidean distance* ha valori più bassi della *Manhattan* e della *Diagonal*, applicando questo tipo di euristica si avranno sicuramente gli *shortest path*, ma l'algoritmo avrà un'esecuzione abbastanza lenta perché esaminerà più nodi. Come si può vedere dalla *Figura 12*, notando il numero di nodi colorati.

Per quanto riguarda invece la ricerca di soluzioni di viaggio si utilizzano multi-grafi orientati dove i nodi sono delle *locations* e quindi entità fisiche raggiungibili (per esempio: città, stazioni ferroviarie, aeroporti, porti) identificati da una coppia di coordinate spaziali come per esempio la latitudine e la longitudine, mentre gli archi possono essere modellati con le informazioni legate al mezzo di trasporto che collega due nodi adiacenti.

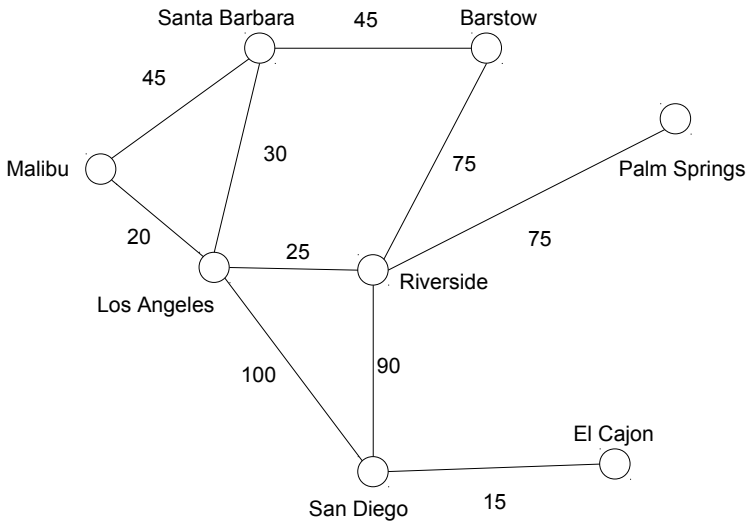

Figura 13: Esempio di grafo con locations Californiane e archi pesati.

Per questo tipo di problemi le funzioni per le stime euristiche sono molto *personalizzabili*, e quindi per idearle si utilizzano concetti guidati dall'esperienza di viaggio. Cercando quindi di simulare una stima di distanza proprio come farebbe un essere umano nello scegliere un itinerario di viaggio. Quelle utilizzate, per esempio, dall'algoritmo *Wayfinder* di Waynaut sono le seguenti:

- *cost*;
- *duration*;
- *agony*.

La metrica di *cost* è il concetto più semplice e significativo che si può utilizzare nell'ambito di ricerche di soluzioni di viaggio, in quanto essa rappresenta il costo proprio in termini di moneta e quindi il *prezzo* per arrivare dal nodo A al nodo B del grafo. Di conseguenza, utilizzando questo tipo di metrica, il cammino minimo risultante dall'esecuzione dell'A* sarà la soluzione di viaggio che *costa di meno* tra il nodo sorgente e il nodo goal, riuscendo magari a combinare tra di loro archi che congiungono i nodi attraverso i mezzi di trasporto più economici per l'input dato.

La metrica di *duration* rappresenta il costo in termini di durata temporale del segmento di viaggio che congiunge due nodi del grafo. Il cammino minimo ottenuto utilizzando questo tipo di metrica sarà quindi la soluzione di viaggio che impiega il minor tempo possibile per arrivare al nodo goal. Utilizzando quindi archi, magari

molto costosi a livello monetario, ma che massimizzano appunto il concetto di durata.

La metrica di *agony* rappresenta il concetto di agonia e quindi un costo che tiene conto della combinazione dei seguenti fattori: la durata di un segmento del viaggio, i tempi di attesa o tempi "morti" che si incontrano nel percorrere il segmento, il numero di cambi che si effettuano (per esempio il numero di scali per raggiungere un determinato aeroporto o il numero di treni da cambiare) e il prezzo in termini monetari del segmento. Con questa metrica come output dell'esecuzione dell'A* si ha una soluzione ottima che massimizza la combinazione di tutti i costi che si possono trovare nel calcolo di una soluzione di viaggio.

Questi concetti verranno trattati nel capitolo 4.

3.2 Pseudo-codice

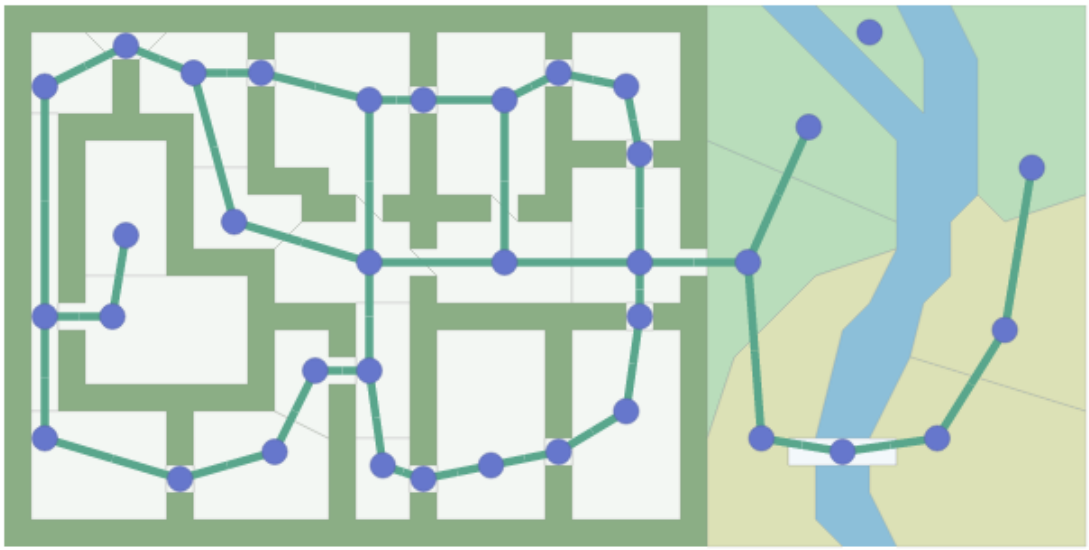

Figura 14: Esempio di grafo di input per l'algoritmo A.*

La prima cosa da capire quando si studia un algoritmo è il *dato*. Bisogna capire cosa l'algoritmo prende in input e cosa restituisce come output. Gli algoritmi di ricerca su grafi, come appunto l'A*, accettano in ingresso un grafo, come quello illustrato nella *Figura 14*.

L'A* non considera niente altro che il grafo, nel senso che vede solo nodi e archi. Non è in grado di capire se ci sono per esempio diverse zone morfologiche, se ci

troviamo in una grande o piccola area, se ci troviamo in ambiente chiuso o all'aperto, se ci troviamo in una zona bagnata dal mare o meno.

L'output dell'algoritmo è un *path* o *cammino* formato da nodi e archi del grafo. Quindi ci fornisce l'informazione sul *percorso* da seguire per muoversi da un punto a un altro, ma non ci dirà il *come* muoverci lungo questo cammino, proprio perché l'A* conosce solo il grafo. L'informazione del come muoversi la si può iniettare nei pesi associati agli archi.

Per implementare l'algoritmo si usano due tipi di insiemi (nel linguaggio informatico indicati anche con la parola *set* o *lista*) chiamati *OPEN* e *CLOSED*. La lista OPEN contiene tutti quei nodi che sono i *candidati* per essere esaminati. La lista CLOSED contiene, invece, i nodi che sono già stati esaminati e quindi inizialmente vuota. Ad ogni nodo l'algoritmo associa i valori della funzione *g* e *h* prima di aggiungerli alla lista OPEN, questa operazione viene chiamata anche "costo di entrata". Da questa lista ad ogni ciclo viene selezionato il nodo con il valore di *f* più basso. Se la lista è vuota, non ci saranno percorsi dal nodo di partenza al nodo goal e l'algoritmo termina. Se, invece, il nodo prelevato dalla lista è il nodo goal, l'A* ricostruisce e pone in output il percorso ottenuto e si arresta. Se il nodo prelevato non è il nodo goal, allora vengono presi in esame i nodi vicini o anche chiamati *neighbors*. Per ciascuno di questi vicini, indicati con n' , A* calcola il costo di entrata e lo salva con il nodo. Questo costo è calcolato dalla somma cumulativa dei pesi immagazzinati nei *parent nodes*, più il costo dell'arco per raggiungere il nodo n' : $g(n') = g(n) + cost(n, n')$.

Se un nuovo nodo che deve essere esaminato è già presente nella lista CLOSED con un costo uguale o più basso, non ci sarà nessuna indagine futura su questo nodo o col percorso ad esso associato. Se un nodo della lista di CLOSED è uguale ad un nuovo nodo da esaminare, ma è stato immagazzinato con un costo più elevato, allora sarà rimosso dalla lista CLOSED, e il processo continuerà a partire dal nuovo nodo.

Successivamente, la stima della distanza dal nodo vicino al nodo goal è aggiunta al costo calcolato in precedenza per formare la funzione *f*. Tale nodo viene aggiunto alla lista OPEN, a meno che non esista un nodo identico con il valore di *f* minore o uguale. Le operazioni vengono ripetute per ogni vicino del nodo corrente, fatto ciò il nodo originale viene posto nella lista di CLOSED. Il prossimo quindi sarà ottenuto dalla lista OPEN e con esso viene ripetuto il processo. Di seguito un esempio di pseudo-codice:

```
function AStar(start, goal){
	// Il set dei nodi già valutati.
	closedSet ← {};
	// Il set dei nodi esplorati non ancora valutati.
	// Inizialmente solo il nodo di partenza viene inserito nel set.
	openSet ← {start};
	/*
	Per ogni nodo raggiunto con un path efficiente utilizziamo una mappa
	per la ricostruzione dei nodi precedenti nel path.
	/*
	cameFrom ← the empty map;
	// Per ogni nodo, il costo per arrivarci partendo dal nodo start.
	gScore ← map with default value of Infinity;
	// Il costo per andare da start a start è zero.
	gScore[start] ← 0 ;
	/*
	Per ogni nodo, il costo totale per andare dal nodo start
	al goal passando per questo nodo.
	/*
	fScore ← map with default value of Infinity;
	// Per il primo nodo il valore è solo euristico.
	fScore[start] ← heuristicCostEstimate(start, goal);
	while openSet is not empty :
		current ← the node in openSet having the lowest fScore[] value
		if current = goal :
			return ricostruisciPath(cameFrom, current);
		end if
		openSet.Remove(current);
		closedSet.Add(current);
		for each neighbor of current :
			if neighbor in closedSet :
				// Ignora il vicino già valutato.
				continue;
			end if
			// La distanza dal nodo start al nodo vicino
			tentativeGScore ← gScore[current] +
					cost(current,neighbor);
			if neighbor not in openSet :
				// Esplorato un nuovo nodo
				openSet.Add(neighbor);
			else if tentativeGScore >= gScore[neighbor] :
				continue; // Non è il path migliore.
			end if
			// Miglior path trovato finora.
			cameFrom[neighbor] ← current
			gScore[neighbor] ← tentativeGScore
			fScore[neighbor] ← gScore[neighbor] +
					heuristicCostEstimate(neighbor,goal);
		end for
	end while
	return failure;
}
function ricostruisciPath(cameFrom, current){
	total_path ← [current];
	while current in cameFrom.Keys :
		current ← cameFrom[current];
		total_path.append(current);
	end while
	return total_path;
}
```

Vedendo il tipo di implementazione, e quindi il funzionamento reale dell'algoritmo, si capisce subito che le liste OPEN e CLOSED giocano un ruolo fondamentale per la buona riuscita dell'A*. Di conseguenza è importante capire che tipo di struttura dati bisogna utilizzare per modellare le due liste. Sulla lista OPEN, per esempio, vengono fatte tre tipi di operazioni: nel ciclo principale ripetutamente si prende dalla lista il nodo con il valore di *f* migliore e quindi più basso; per i nodi vicini bisogna fare un controllo sul fatto che essi siano presenti o meno nella lista; i nodi vicini devono poi essere inseriti nella lista OPEN. Le operazioni di *insert* e *remove-best* sono operazioni tipiche della struttura dati chiamata coda di priorità o *priority queue* che è una struttura dati simile ad una *pila*, ma diversa per via del fatto che ogni elemento inserito ha una sua "priorità". Ogni elemento avente una priorità più alta viene inserito prima di un altro avente priorità più bassa. In particolare, l'elemento con priorità alta è posto in cima alla coda, mentre l'elemento con la più bassa priorità si troverà in coda. Questo tipo di struttura viene implementata in tanti modi, quello che fornisce una maggiore *performance* è l'*heap binario*.

L'*heap binario* è un albero binario finito dove i vertici vengono etichettati da elementi di un insieme linearmente ordinato detto "chiavi", che soddisfa le seguenti regole:

- se v è figlio di v' allora $chiave(v) \leq chiave(v')$;
- l'albero è completo tranne al più l'ultimo livello, che è riempito con le foglie a partire da sinistra.

Un heap binario può essere realizzato come un vettore H di dimensione pari alla cardinalità dell'albero e tale che: $H[1]$ sia la radice dell'albero; $H[2i]$ e $H[2i+1]$ siano rispettivamente il figlio sinistro e quello destro di $H[i]$.

Nell'operazione di *insert* il nuovo elemento viene aggiunto come foglia dell'albero, facendola quindi risalire lungo il ramo cui è stata aggiunta finché poi non viene ricostruito l'heap.

L'operazione di *extract*, invece, avviene dalla radice attraverso due fasi: l'elemento più a destra dell'ultimo livello rimpiazza la radice; l'elemento ora in radice viene fatto discendere lungo l'albero finché non sia maggiore di entrambi i figli; nel discendere si sceglie sempre il figlio col valore massimo della chiave.

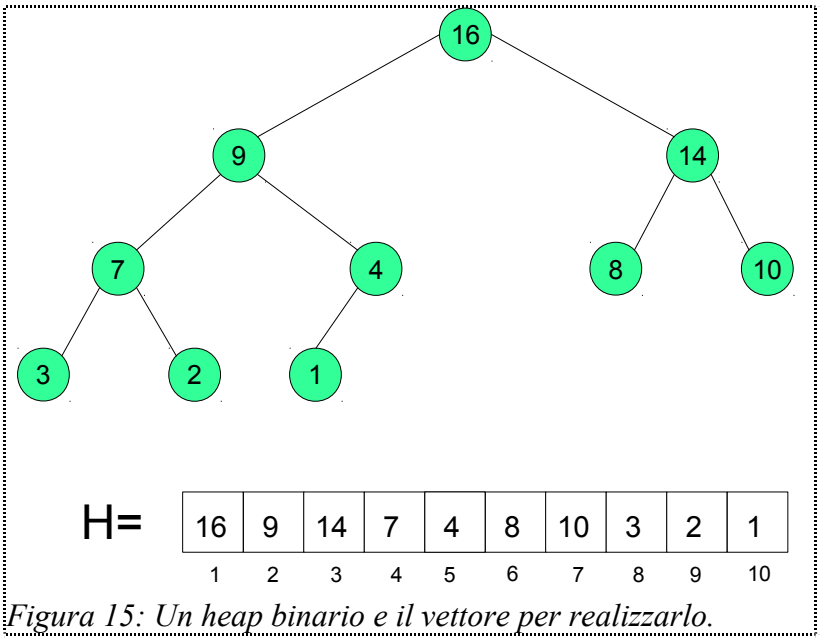

Figura 15: Un heap binario e il vettore per realizzarlo.

A questo punto vediamo un applicazione dell'algoritmo su un grafo di esempio (*Figura 16*) assumendo che la funzione euristica utilizzata è la distanza in linea d'aria tra i nodi del grafo.

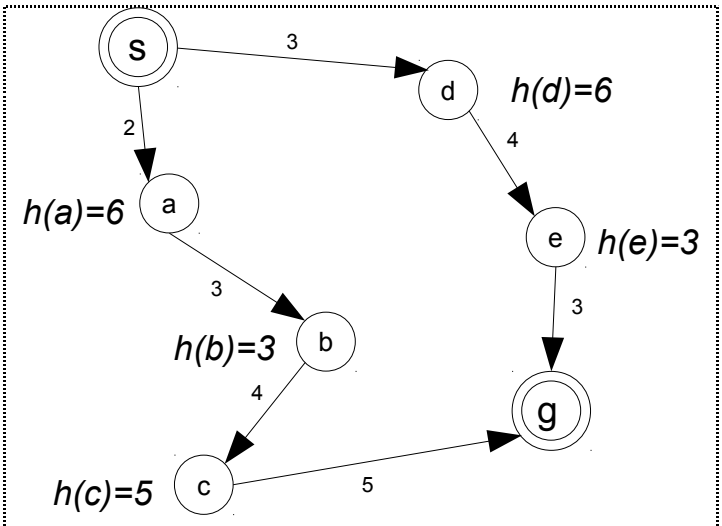

Figura 16: Grafo di esempio.

Come si può vedere anche dalla figura, il nodo sorgente è il nodo *s* mentre il nodo destinazione o nodo *goal* è il nodo *g*. L'algoritmo effettua un numero passi pari a 5. Al primo passo vengono analizzati i vicini del nodo sorgente (*a*, *d*) e vengono

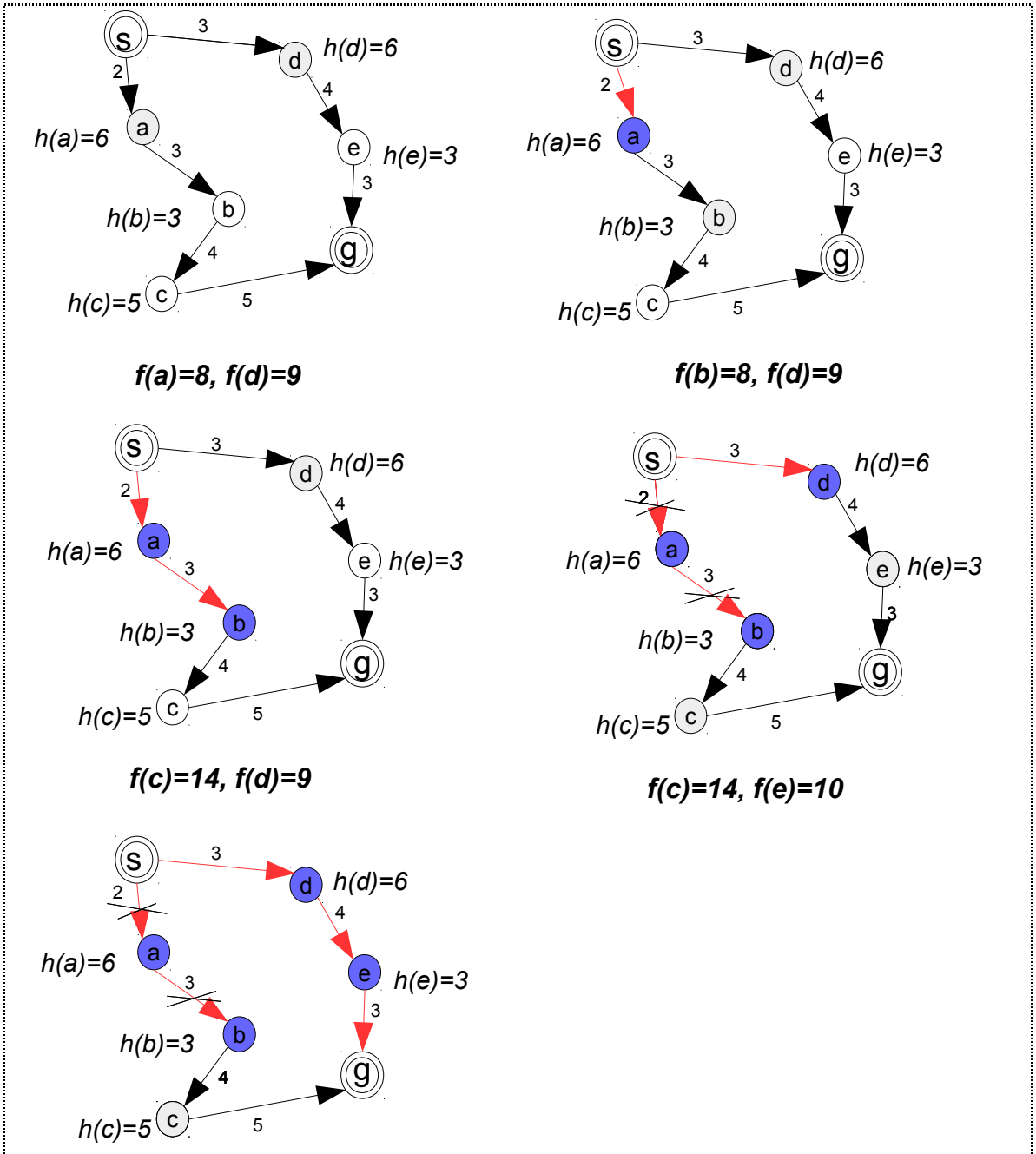

Figura 17: Sequenza di passi che effettua l'A sul grafo d'esame.*

quindi calcolati i valori della funzione *f*. In particolare: $f(a)=2+6=8$ (costo dell'arco che congiunge *s* ad *a* più il valore dell'euristica del nodo *a* e quindi la distanza in linea d'aria dal nodo *a* al nodo goal) e $f(d)=3+6=9$. A questo punto il primo passo è completato e l'algoritmo sceglie il nodo che ha il valore *f* più basso, passando quindi al nodo *a*. A questo punto vengono analizzati i vicini di *a* (il nodo *b*) e si calcolano i valori della funzione *f* aggiungendoli a quelli già calcolati. Per il

nodo *b* la funzione vale: $f(b)=2+3+3=8$, (costo degli archi che congiungo il nodo sorgente *s* al nodo *b* più l'euristica del nodo *b).* A questo punto l'algoritmo sceglie di seguire il nodo *b* e analizzare i suoi nodi vicini (il nodo *c*) e calcolando $f(c)=2+3+4+5=14$. Dato l'alto valore della funzione in *c* l'algoritmo sceglie di andare a esplorare la strada che passa per il nodo *d*. Al nodo *d* viene analizzato l'unico vicino "*e"* e calcolata la funzione $f(e)=4+3+3=10$, risultando quindi essere la strada vantaggiosa. A questo punto viene analizzato *e*, il suo vicino è proprio il nodo destinazione, completando così la ricerca. I passi appena spiegati sono riportati nella *Figura 17*.

Il cammino minimo risulta essere la seguente sequenza di nodi: *s*, *d*, *e*, g, avente un peso totale pari a 10. Mentre l'altro cammino, ossia quello che segue i nodi *s*, *a*, *b*, *c*, *g*, ha un peso totale pari a 14. Il passo interessante è il numero 3, l'esplorazione arriva al nodo *b* (siccome quella strada sembrava essere molto promettente) ma poi l'A*, grazie alla stima euristica, decide di abbandonare quella strada e di provare a esplorare altri nodi che sono caratterizzati da un valore di *f* più basso. Facendo questo tipo di scelta si risparmia l'esplorazione del nodo *c* e quindi l'esame dei sui vicini e il calcolo della funzione *f* per essi.

3.3 Vantaggio dell'algoritmo A*

In questo paragrafo viene evidenziato il punto di forza dell'algoritmo A*, prendendo come riferimento l'algoritmo di Dijkstra, essendo anch'esso un algoritmo di ricerca dello *shortest path* su grafo e visto che i due algoritmi hanno dei punti in comune. Infatti, come è stato già presentato nel paragrafo precedente, se all'algoritmo A* viene fornita una funzione euristica pari alla funzione identicamente nulla, esso si comporta proprio come l'algoritmo di Dijkstra. Questo è un concetto fondamentale perché ci suggerisce che l'algoritmo di Dijkstra compie una ricerca "cieca", in assenza della "guida" dell'euristica, che lo porta quindi ad esplorare molti più nodi in ogni direzione per arrivare a destinazione, mentre l'A* esamina solo l'area che è *nella direzione del nodo goal*. Di conseguenza l'area esplorata dall'A* è sicuramente più contenuta rispetto a quella esplorata da Dijkstra, rendendo quindi l'A* molto veloce. Di seguito due figure che mostrando i nodi esplorati dai due algoritmi.

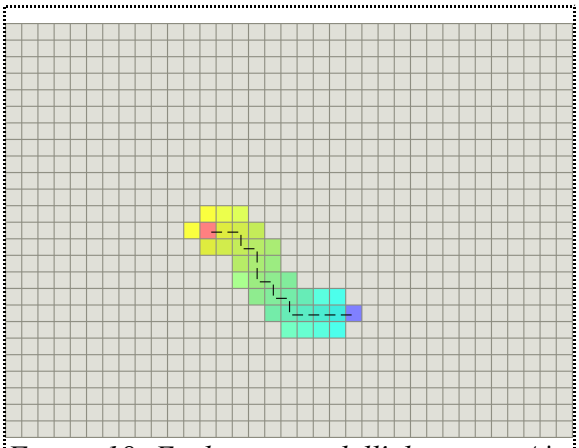

Figura 18: Esplorazione dell'algoritmo A.*

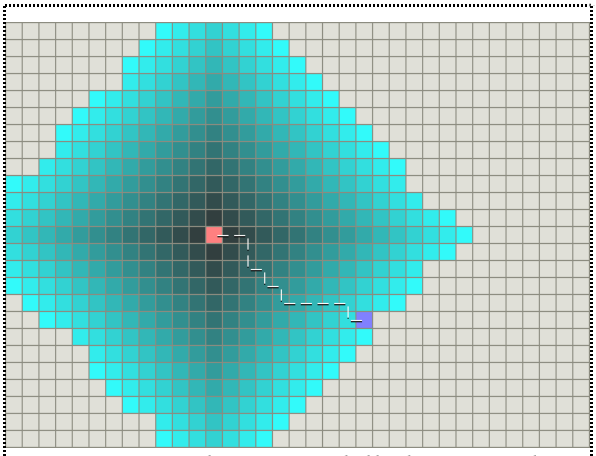

Figura 19: Esplorazione dell'algoritmo di Dijkstra.

Come si può vedere nella *Figura 19* Dijkstra esplora un numero elevato di nodi vicini al nodo sorgente (i quadratini colorati di blu), cosa che non succede nell'esplorazione A* di *Figura 18* dove in questo caso i colori dei quadratini sono diversi per via dell'euristica che decresce in base alla vicinanza al nodo goal.

Nel caso Waynaut, per esempio, il concetto di velocità è un concetto imprescindibile. Perché il sistema che implementa l'algoritmo che deve fornire soluzioni di viaggio (e quindi *shortest path*) agli utenti, deve essere *responsive* e allo stesso tempo accurato. Nessun utente userebbe un sistema che per dare una risposta, sia essa una soluzione di viaggio o qualsiasi altro tipo di servizio, impiegherebbe un tempo molto lungo.

La funzione euristica è quindi il vero punto di forza di questo algoritmo. In base a come essa viene modellata possiamo avere il giusto bilancio tra velocità e bontà della soluzione, come già espresso nel capitolo. La cosa importante è riuscire a capire cosa si vuole come output dall'algoritmo e successivamente implementare la funzione euristica più consona. Proprio perché in alcuni casi magari è preferibile avere un percorso "buono" rispetto al percorso "perfetto" guadagnandone in tempi di esecuzione oppure, al contrario, decidere di avere un percorso "perfetto" aumentando il tempo di esecuzione.

4. Caso di studio: Wayfinder

In questo capitolo presenteremo il caso di studio *Wayfinder* (progetto dell'azienda Waynaut) che ha come obiettivo la creazione di soluzioni di viaggio multimodali. In particolare ci focalizzeremo su un sotto-problema del multimodale, ossia: come l'algoritmo A* calcoli lo *shortest path* a partire da un multigrafo orientato e i nodi sorgente e destinazione. Descrivendo il grafo, la funzione costo e l'euristica utilizzata. Mostreremo, poi, i risultati reali dell'algoritmo prendendo in esame una tratta di viaggio.

4.1 Descrizione del grafo

Uno dei sotto-problemi affrontati dal progetto Wayfinder è il calcolo del percorso ottimale (*shortest path*) su di un multigrafo orientato $G=(V,E)$ utilizzando l'algoritmo A*. Come tutti gli algoritmi per i cammini minimi, A* prende in input un grafo e i nodi per i quali si vuole calcolare lo shortest path. Per poter capire il tipo di soluzioni che il sistema è in grado di generare, bisogna focalizzarsi, in prima istanza, sul grafo utilizzato e quindi sulle caratteristiche dello stesso. Evidenziando come sono fatti i nodi e gli archi che li collegano.

I nodi utilizzati dall'algoritmo (gli elementi dell'insieme V) rappresentano entità fisiche raggiungibili attraverso i diversi tipi di mezzo che Wayfinder è in grado di utilizzare. Queste ultime vengono raggruppate sotto un'unica classe *astratta* indicata con il termine *location*.

Ogni location ha un certo numero di attributi che possono essere rappresentati con il seguente elenco:

- un nome;
- una coppia di coordinate, come la latitudine e la longitudine;
- un insieme di informazioni legate alla nazione e alla regione di appartenenza;
- la timezone.

Essendo *location* astratta (indicata anche con il termine *classe padre* o *classe base*), esistono classi che la estendono (indicate con *classi derivate* o *classi figlie*). Queste classi ereditano i campi della classe padre aggiungendo ulteriori concetti colmando le "lacune" della classe base. Le classi figlie utilizzate in Wayfinder sono le seguenti:

- stazione ferroviaria o *Station*;
- aeroporto o *Airport*;
- porto o *Port*;
- città o *City*.

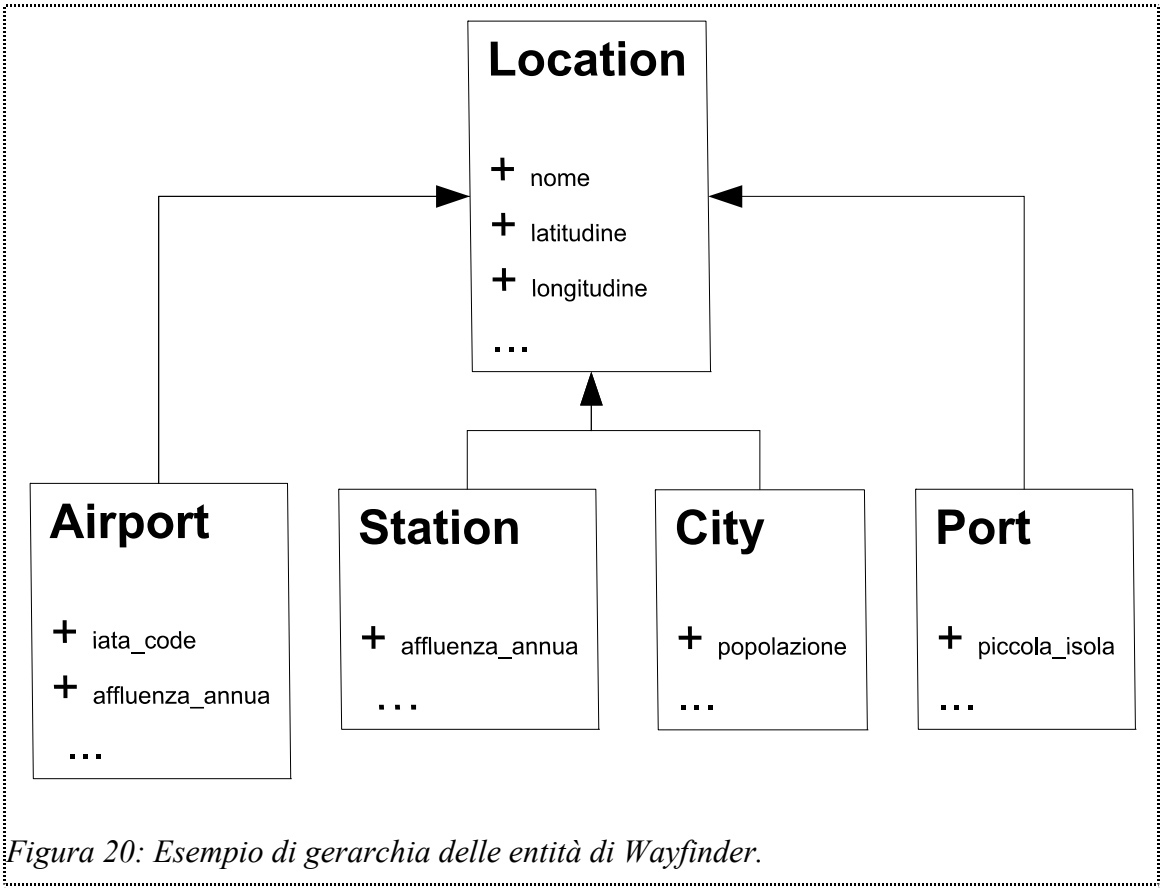

Figura 20: Esempio di gerarchia delle entità di Wayfinder.

La specializzazione viene utilizzata, quindi, per estendere il concetto di location (nella *Figura 20* è presentata la gerarchia utilizzata nella piattaforma di Wayfinder). Infatti per un'entità *Airport* non bastano i concetti o gli attributi ereditati dall'entità padre (coppia di coordinate, nome, nazione ecc), ma a questi si aggiungono attributi che modellano l'informazione propria della classe figlia come, ad esempio, lo *iata code* (codice identificativo di un aeroporto) o il numero di passeggeri per l'anno corrente (affluenza annua), quest'ultimo presente anche per l'entità *Station*. Per quanto riguarda la *City* si considera in aggiunta anche il valore della popolazione della città, sempre dell'anno corrente, mentre per l'entità *Port*, per alcuni scopi

dell'algoritmo, è importante sapere se il porto selezionato risulta essere un porto di piccola isola o meno.

L'insieme di entità che specializzano location rappresentano i nodi del multigrafo che l'algoritmica di Wayfinder accetta in input, definendo così l'insieme *V*.

Gli archi del grafo, invece, rappresentano la soluzione di viaggio che il sistema Wayfinder mette a disposizione per il collegamento di due nodi adiacenti del grafo, nel nostro caso tra due particolari tipi di location. Queste soluzioni di viaggio (che sono quindi gli elementi dell'insieme *E*) sono raggruppate sotto un'unica classe indicata con il termine *Trip*.

Ogni istanza di Trip contiene le seguenti informazioni:

- location di origine;
- location di destinazione;
- data di partenza;
- data di destinazione;
- il prezzo e la moneta utilizzata;
- la durata;
- il tipo di mezzo utilizzato;
- la lista di *Transfer*;

Per tipo di mezzo si intende proprio il mezzo fisico che bisogna utilizzare per compiere il viaggio tra i due nodi adiacenti (ad esempio: treno, volo, bus, traghetto o passaggio in auto).

Il concetto della lista di Transfer può essere spiegato in questo modo: se un viaggio tra due nodi adiacenti è effettuato con lo stesso tipo di mezzo ma con uno o più cambi di stazione o uno o più scali, l'algoritmica di Wayfinder lo considera come un'unica istanza di Trip, riportando i segmenti separati nella lista di Transfer. Ad esempio, un viaggio in treno tra i nodi adiacenti dalla Stazione di Milano Centrale e la Stazione di Napoli Centrale con un cambio a Roma Termini è un unico Trip, visto che il mezzo che congiunge i due segmenti separati Milano-Roma e Roma-Napoli è sempre un treno. Proprio questi segmenti separati vengono inseriti nella lista di Transfer.

Come vederemo poi nei paragrafi successivi, il prezzo e la durata di un Trip vengono utilizzati per ottenere i valori della funzione costo e della funzione euristica, fondamentali per il calcolo dello shortest path sul grafo.

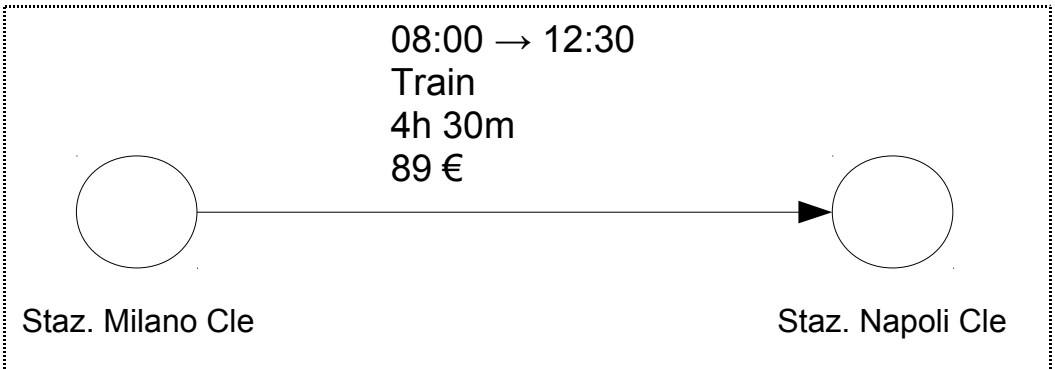

Figura 21: Esempio di sottografo composto da due Location e un Trip.

Un insieme di Trip formati come descritto (nella *Figura 21* è raffigurato un esempio) rappresenta l'insieme degli archi che congiungono i nodi del grafo che l'algoritmo di Wayfinder accetta in input, definendo così l'insieme *E*.

La piattaforma di Wayfinder, ricevuta una richiesta di input, crea il multigrafo G con le caratteristiche appena enunciate, andando a considerare un insieme di nodi e un insieme di archi che li congiungono. Il come venga creato questo grafo, purtroppo, è un argomento che va oltre gli obiettivi di questa tesi.

4.2 Descrizione della funzione costo e della funzione euristica

Una volta creato il grafo, *Wayfinder* fa partire la ricerca A* per la generazioni di shortest path e in questo paragrafo descriviamo le funzioni costo e le euristiche che la piattaforma utilizza per generarli. L'implementazione di queste funzioni utilizzano i seguenti concetti per il singolo *Trip*:

- costo;
- durata;
- agony.

Costo: per questo tipo di metrica la funzione costo (quindi la funzione *g* utilizzata dall'algoritmo) è formata dal prezzo del segmento di viaggio che congiunge due

punti del grafo. Mentre la funzione euristica (i valori della funzione *h* utilizzata dall'A*) rappresenta un prezzo stimato del viaggio (dal nodo corrente al nodo goal), utilizzando una distanza tra i due punti moltiplicata per una costante (una stima del prezzo di viaggio per l'unità di misura di lunghezza scelta). Con questa implementazione l'algoritmica fornisce il cammino minimo che minimizza il prezzo totale del viaggio, dando priorità all'esplorazione dei nodi che sono raggiunti da archi che hanno prezzo basso.

Durata: per questa metrica la funzione costo è formata dalla durata del segmento di viaggio che congiunge i due punti più il tempo di attesa che esiste tra la data di partenza del trip corrente e la data di arrivo del trip appena precedente. Mentre la funzione euristica rappresenta una durata di viaggio stimata tra il nodo corrente e il nodo *goal*, utilizzando quindi la velocità media del mezzo considerato e la distanza tra i due punti. Con questa implementazione viene presentato in output lo shortest path che minimizza la durata totale del viaggio, dando priorità all'esplorazione dei nodi che sono raggiunti da archi che impiegano il minor tempo possibile.

Agony: per questa metrica la funzione costo è formata da una combinazione lineare tra il costo in termini di durata del viaggio, il costo in termini di prezzo e il costo in termini di tempi di attesa tra i transfer del segmento di viaggio. Mentre l'euristica rappresenta una combinazione lineare tra la stima della durata e del prezzo del viaggio per raggiungere il nodo *goal*. Con questa implementazione l'algoritmo presenta in output la soluzione che minimizza la combinazione lineare tra il prezzo, la durata del viaggio e la durata dei tempi di attesa per il cambio, dando priorità all'esplorazione dei nodi che presentano il valore minore della combinazione di questi fattori.

4.3 Risultati dell'algoritmo

In questo paragrafo vengono presentati due risultati dell'A* su una tratta di viaggio di esempio, utilizzando prima la metrica di costo e poi la metrica di durata. La tratta che viene esaminata è un stazione Milano Centrale – stazione di Aversa per il giorno 12/05/2017 effettuata il giorno 08/05/2017. Da tenere presente che utilizzeremo un grafo semplificato rispetto a quello che utilizza realmente *Wayfinder*, andando a considerare solo un sotto-insieme di nodi e un sotto-insieme di archi (quindi solo un sotto-insieme di segmenti di viaggio). Dando per scontato alcune logiche del sistema e non trattando i temi per la dipendenza temporale del grafo, non essendo obiettivo di tesi.

Per la richiesta di esempio la piattaforma *Wayfinder* genera un grafo (quello di *Figura 22* ne è un sottografo*)*. In questo grafo si possono notare quindi i singoli segmenti di viaggio, ad esempio: l'aeroporto di Linate è raggiungibile dalla stazione di Milano Centrale attraverso un Trip di tipo ncc (noleggio auto con conducente) al costo di 38,82 € e con una durata di 23 minuti. Linate è collegato all'aeroporto di Capodichino attraverso un trip di tipo volo che parte alle 11:00 e arriva alle 12:15 con un prezzo di 188,29 € e un durata di 75 minuti. Capodichino è collegato con la stazione di Napoli Centrale attraverso uno shuttle aeroportuale che costa 10€ e con una durata di 20 minuti (partendo alle ore 13:00 e arrivando in stazione alle 13:20). A questo punto dalla stazione di Napoli si arriva alla stazione di Aversa con un treno che parte alle ore 13:42 e arriva alle ore 14:05.

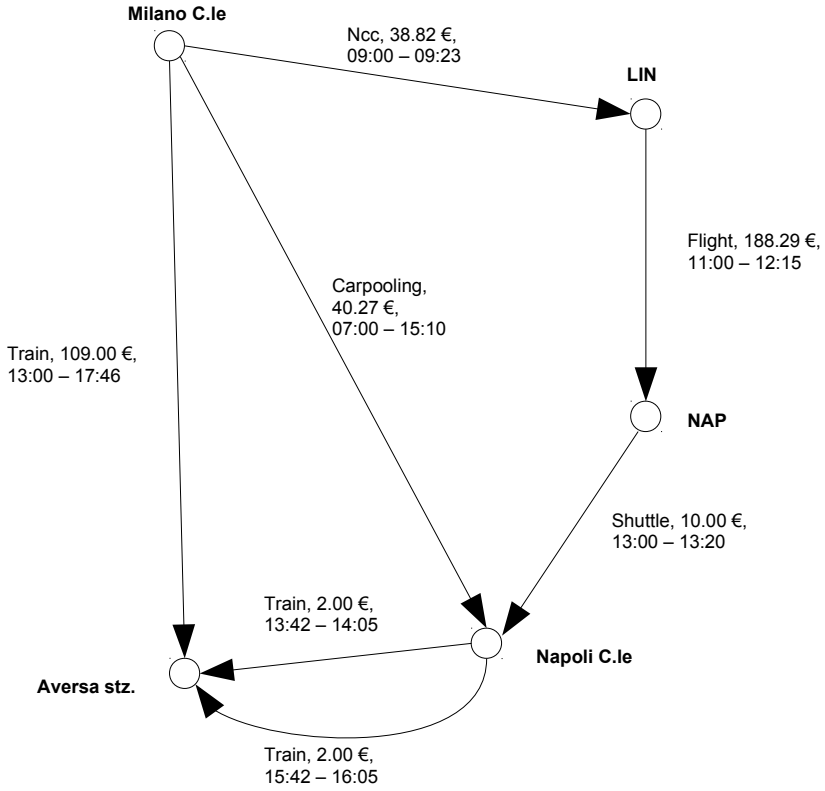

Figura 22: Esempio di sottografo creato dalla piattaforma Wayfinder per la pianificazione di percorsi ottimali sulla richiesta di viaggio Milano Cle – Aversa stz.

Inoltre la stazione di Milano Centrale è collegata sia alla stazione di Aversa, sia alla stazione di Napoli Centrale. Nel primo caso con un trip di tipo treno che contiene una lista di transfer che modellano il cambio a Napoli Centrale e che ha una durata totale di 286 minuti, nel secondo caso con un trip di tipo *Carpooling* (passaggio in auto venduto da *BlaBlaCar*) al costo di 40,27€ e con una durata di 8h e 10 minuti.

Per arrivare a destinazione, seguendo il passaggio in auto, si po' notare un altro arco uscente dal nodo Napoli C.le che rappresenta un altro treno che congiunge quest'ultimo nodo con la stazione di Aversa che parte alle 15:42 e arriva a destinazione alle 16:05. Ripetiamo che il grafo di *Figura 22* è un grafo semplificato rispetto a quello che realmente Wayfinder utilizza. Infatti, per esempio, consideriamo solo un trip che congiunge LIN a NAP, nonostante ce ne siano centinaia al giorno. Lo stesso discorso vale per i treni che collegano Milano ad Aversa con un cambio a Napoli e per i treni che collegano Napoli Centrale alla stazione di Aversa (in questo caso ne consideriamo due per avere la possibilità di trovare un percorso finito, sia attraverso il *carpooling* sia attraverso il cammino che contiene l'arco di tipo *flight*).

Una volta presentato il grafo di esempio ci focalizziamo sulla ricerca dello shortest path dell'algoritmo A* utilizzando per prima la metrica di *costo*. Quella che segue è una figura che contiene il grafo dove su ogni singolo arco è rappresentato il valore della funzione costo (valori di *g*) e su ogni nodo il valore della funzione euristica (valori di *h*) attraverso la metrica scelta.

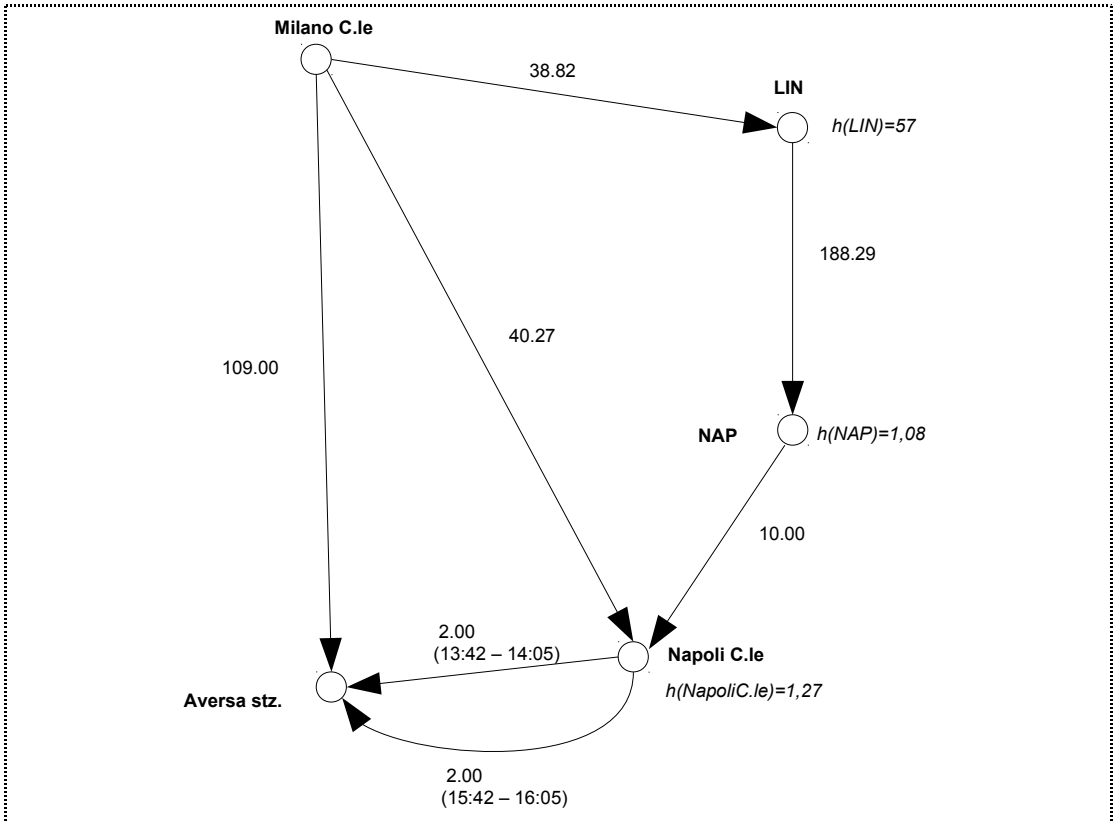

Figura 23: Sottografo della tratta Milano Centrale – Aversa stazione, dove sugli archi sono presenti i costi e sui nodi i valori dell'euristica dati dalla metrica di costo.

La ricerca A* parte del nodo sorgente Milano Centrale esplorando i nodi vicini e calcolando il costo per raggiungerli e sommando l'euristica, quindi avremo:

- $f(Aversa\,Stazione)=109$, costo del solo arco per raggiungere la destinazione;
- $f(Napoli\,Centrale)=40{,}27+1{,}27=41{,}54$, costo del trip in BlaBlaCar più l'euristica calcolata in Napoli Centrale;
- $f(LIN)=38{,}82+57=95{,}82$, costo del trip ncc più l'euristica calcolata dall'aeroporto di Linate per la destinazione.

Una volta esauriti i vicini di Milano Centrale, l'algoritmo sceglie come *current node* il nodo con il valore di *f* più basso, selezionando quindi Napoli Centrale (eliminandolo dalla lista OPEN e aggiungendolo in quella CLOSED) e esplorandone i *neighbors*, andando quindi a selezionare solamente Aversa stazione e considerando solo il trip che è possibile collegare temporalmente (quello delle ore 15:42). A questo punto viene calcolato il costo per arrivare in quel nodo: $g(Aversa\,Stazione)=40{,}27+2=42{,}27$. Questo valore di *g* per il nodo Aversa è più basso rispetto a quello calcolato in precedenza (per il trip treno da Milano Centrale) e quindi l'algoritmo lo aggiorna e ricalcola la funzione *f:* $f(Aversa\,Stazione)=42{,}27+0=42{,}27$, l'euristica non esiste essendo Aversa i l nodo destinazione.

A questo punto la lista OPEN è composta dal nodo di LIN con una $f(LIN)=95{,}82$ e il nodo Aversa con il suo nuovo valore $f(Aversa\,Stazione)=42{,}27$. L'algoritmo sceglie il nodo di Aversa e termina ricostruendo lo shortest path generato, visto che il *current* è diventato il nostro nodo *goal*.

Il cammino minimo è quindi composto dai nodi di Milano Centrale (sorgente), Napoli Centrale e Aversa Stazione (goal) e dagli archi che li congiungono: un trip in BlaBlaCar da Milano a Napoli Centrale e il treno che collega Napoli ad Aversa È possibile vederlo sul grafo di *Figura 25* seguendo gli archi di colore rosso.

Con questo tipo di euristica l'algoritmo è riuscito a generare un percorso ottimale (il migliore in termini di prezzo) su un grafo inteso come una rete di trasporto. Favorendo la velocità dell'esplorazione, evitando ad esempio la visita dei nodi collegati all'aeroporto di Linate (sfavorevoli per la stima euristica). In questo caso premiando, oltretutto, un percorso multimodale.

Adesso, invece, studiamo come la ricerca A* genera lo shortest path sullo stesso grafo utilizzando, però, la metrica di durata. Di seguito riportiamo il grafo con i nuovi valori delle funzioni *g* e *h*.

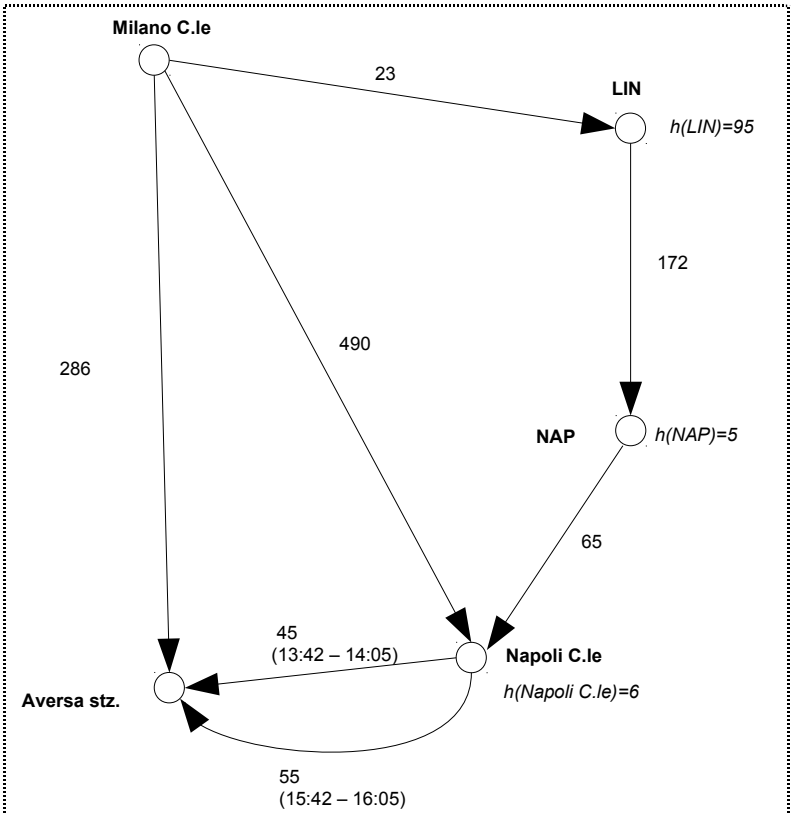

Figura 24: Sottografo della tratta Milano Centrale – Aversa stazione, dove sugli archi sono presenti i costi e sui nodi i valori dell'euristica dati dalla metrica di durata.

L'algoritmo parte dalla sorgente (Milano Centrale) e esplora i sui vicini calcolando costo e euristica e aggiungendoli nella lista OPEN con le seguenti funzioni *f*:

- $f(LIN)=23+95=118$, durata del trip ncc più l'euristica del nodo di LIN;
- $f(Aversa)=286$, che come addendo presenta solo la funzione *g* in termini di durata;
- $f(Napoli\,Centrale)=490+6=496$, costo in termini di durata per arrivare a Napoli con il trip di BlaBlaCar e l'euristica per raggiungere il nodo *goal*.

A questo punto l'algoritmo sceglie l'aeroporto di Linate rimuovendolo dalla lista OPEN e aggiungendolo in quella di CLOSED. Vengono esplorati i vicini di Linate,

solo l'aeroporto di Napoli, aggiungendolo alla lista OPEN con $f(NAP)=23+172+5=200$, costo del cammino per arrivare a NAP più l'euristica per il nodo goal. Tra tutti i nodi della lista di OPEN, viene preso quello con il valore di *f* più basso, selezionando quindi NAP, rimuovendolo dalla lista OPEN e aggiungendolo nella CLOSED e vengono esplorati i suoi vicini. IL nodo di Napoli Centrale è presenta nella list di OPEN con un valore di *f* pari a 496, ma passando per questo path per il nodo si spende *g=260* di conseguenza il cammino che segue il trip *flight* e che passa per Napoli centrale è migliore rispetto al trip BlaBlaCar, sempre in termini di durata. Quindi per Napoli Centrale la funzione *f* viene aggiornata al valore $f(Napoli\,Centrale)=23+172+65+6=266$, risultando quindi essere il nuovo *current node* e viene rimosso dalla lista di OPEN e aggiunto a quella di CLOSED. L'unico vicino di Napoli Centrale è Aversa stazione e ha un valore di *g* pari a 286. A questo punto l'algoritmo calcola il costo *g* per arrivare a Napoli Centrale più il costo per collegarlo ad Aversa. Questo costo è pari a 305 (costo del trip ncc, costo del trip flight, costo del trip transfer e costo del treno finale), quindi maggiore del costo già salvato (esiste un percorso migliore per il nodo di aversa), non facendo quindi un'operazione di aggiornamento. Nella lista di OPEN è rimasto solo il nodo di Aversa stazione con *f=286*. La stazione viene scelta come *current node* e l'algoritmo termina, dando come shortest path il treno che collega Milano Centrale ad Aversa con un cambio a Napoli Centrale, il percorso ottimale in termini di durata del viaggio. In questo caso l'algoritmo ha provato ad esplorare la strada che sembrava più promettente per la stima euristica utilizzata, ritornando però sui propri passi quando il costo accumulato dei vari trip, collegati attraverso un flight, era maggiore rispetto al costo del trip in treno.

Il cammino minimo è quindi formato solamente dai nodi sorgente e destinazione collegati da un arco che rappresenta un trip in treno con una durata di 4h e 46 minuti. È possibile vederlo nella *Figura 25* seguendo gli archi di colore blu.

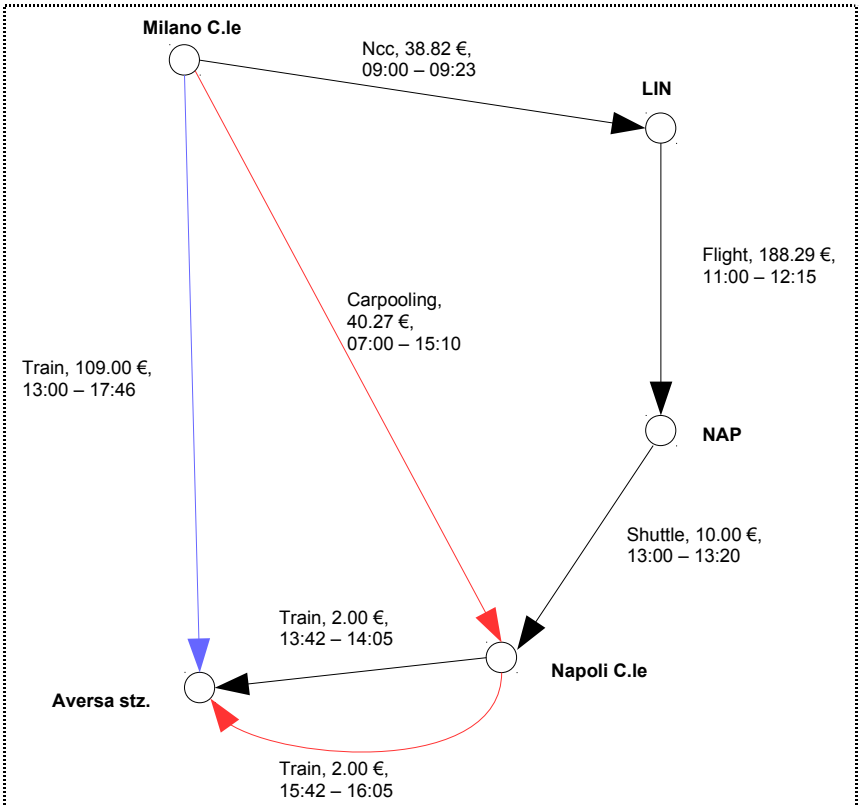

Figura 25: Sottografo di esempio per la tratta Milano Centrale – Aversa con i due shortest path calcolati con la metrica di costo (rosso) e la metrica di durata (colore blu).

Conclusioni

Nel seguente lavoro di tesi è stato presentato il problema della pianificazione di percorsi ottimali su un grafo, analizzando le tecniche per risolverlo.

Per capire cosa si intende per percorso ottimale abbiamo presentato i concetti fondamentali della teoria dei grafi e gli algoritmi più importanti che calcolano lo *shortest path* a partire da un grafo e dai nodi sorgente e destinazione.

La concentrazione è stata posta sull'algoritmo di ricerca A* e su come esso è in grado di calcolare con efficienza i cammini minimi, presentando i maggiori punti di forza. Abbiamo formalizzato come, scegliendo una buona funzione euristica, l'algoritmica riesca a generare lo shortest path senza esplorare inutilmente una buona parte del grafo di input. Riuscendo così a generare l'output in tempi brevi, requisito importante per una buona parte di applicazioni reali.

Come caso di studio abbiamo presentato *Wayfinder*, piattaforma di Waynaut per la generazione di soluzioni di viaggio multimodali, che sfrutta la ricerca A* per risolvere uno dei sotto-problemi del multimodale: trovare i cammini minimi su un grafo inteso come rete di trasporto. Abbiamo descritto le metriche e le relative funzioni *g* e *h* utilizzate dall'algoritmo, che permettono la generazione di percorsi ottimali (come quello di *Figura 26)*. Ovviamente limitandoci ad utilizzare un grafo semplificato, tralasciando il problema della tempo-dipendenza e il problema della creazione del grafo (potenziali obiettivi di un lavoro di tesi successivo).

Figura 26: Percorso ottimale generato dall'algoritmo Wayfinder.

Bibliografia

1. Amit Patel, *Introduction to A* algorithm*, Stanford CS Theory, 2017.

2. Daniel Imms, *A* pathfinding algorithm*, Growing with web, 2016.

3. Abhishek Goyal, Prateek Mogha, Rishabh Luthra, Ms. Neeti Sangwan, *Pathfinding: A* or Dijkstra's?*, International Journals of Multi Dimensional Research, 2014.

4. V. Lacagnina, *Teoria dei grafi: ricerca di percorsi a minimo costo*, Università degli Studi di Palermo, 2003.

5. Keijo Ruohonen, *Graph Theory*, Tampere University Of Technology, 2013.

6. Sebastiano Vigna, *L'algoritmo di Dijkstra*, Università degli Studi di Milano, 2006.

7. L. Di Giovanni, *Cammini minimi: algoritmi di Bellman-Ford e Dijkstra*, Università degli studi Padova, 2010.