Relazione per il progetto di Intelligenza Artificiale e Laboratorio

Docente: **Gian Luca Pozzato**

Gruppo di lavoro:

- **Alessandro Salogni**
  Matricola: 906298
  E-mail: alessandro.salogni@edu.unito.it

- **Riccardo Perotti**
  Matricola: 906237
  E-mail: riccardo.perotti@edu.unito.it

Metropolitana di Londra (Progetto Prolog)

Modellazione del dominio

La modellazione del dominio è stata affrontata seguendo le linee guida forniteci e poi ampliata attraverso la creazione di nuovi fatti e di nuove regole in grado di modellare altri aspetti utili all'elaborazione di differenti euristiche.
In particolare, per quanto riguarda la metropolitana, ci siamo concentrati sulla modellazione dei seguenti aspetti:

- Il percorso delle linee della metropolitana;
- Le stazioni;
- Le coppie di stazioni adiacenti;
- Il numero minimo di cambi tra coppie di linee della metropolitana.

Per quello che riguarda la modellazione delle soluzioni abbiamo invece modellato i seguenti aspetti del dominio:

- Le azioni che un agente può compiere;
- Lo stato in cui un agente si può trovare in un determinato istante.

Percorsi delle linee della metropolitana

Uno dei primi aspetti modellati è stato quello dei percorsi affrontati dalle linee della metropolitana di Londra, ovvero la sequenza di stazioni attraverso le quali una determinata linea passa. Per modellare tale aspetto abbiamo creato dei fatti strutturati nella seguente maniera:

```
percorso(Linea, Dir, ListaFermate).
```

La variabile *Linea* modella il nome della linea metropolitana – ad esempio *"Metropolitan"* o *"Victoria"*, mentre la variabile *ListaFermate* modella, attraverso una lista, la sequenza di stazioni nella quale la linea si ferma.
L'ultima variabile *Dir* modella la direzione in cui i treni di una linea possono essere presi e può assumere i valori 0 e 1: nel primo caso si segue l'ordine delle fermate in *ListaFermate*, nel secondo caso si segue l'ordine della lista invertita. L'inversione della lista, nel caso in cui la direzione assunta sia 1, viene fatta dalla seguente regola, la quale prende la lista di stazioni ordinate e le restituisce invertite:

```
percorso(Linea,1,LR) :- percorso(Linea,0,L), reverse(L,LR).
```

A partire dal concetto di percorso abbiamo realizzato modelli di metropolitana di dimensioni crescenti, in grado di permetterci di simulare scenari via via più complessi. In particolare abbiamo realizzato 4 modelli differenti:

- *Metro_zone_1_simplified*: rappresenta il modello più semplice disponibile e racchiude solo le linee metropolitane della Zona 1 non sovrapposte totalmente (non sono presenti *"Hammersmith & City"* e *"Metropolitan"* che si sovrappongo a *"Circle"* e *"District"* che si sovrappone a *"Circle"*) e con un'unica diramazione (*"Northern"* è considerata come unica attraverso l'utilizzo della linea *"Northern East"*).
- *Metro_zone_1*: rappresenta tutte le linee metropolitane della Zona 1 senza le limitazioni precedenti (vengono rappresentate *"Hammersmith & City"* e *"Metropolitan"*; *"Northern"* viene suddivisa in *"Northern West"* e *"Northern East"*; *"District"* viene rappresentata e suddivisa in *"District North-South"* e *"District East-West"*).
- *Metro_zone_2*: espande le linee metropolitane della Zona 1 aggiungendo le stazioni della Zona 2.
- *Metro_zone_3*: è il modello più complesso ed espande le linee metropolitane della Zona 2 aggiungendo le stazioni della Zona 3;
- *Metro_zone_3_close*: il modello è del tutto analogo a quello precedente, a parte per l'aggiunta di fatti che indicano quali stazioni sono chiuse e quindi solo di passaggio.

In ognuno di questi modelli sono stati aggiunti altri due aspetti fondamentali: la tratta e la fermata. Questi concetti vengono modellati con delle regole a partire dai fatti che rappresentano i percorsi.

Con il concetto di tratta si intende le coppie di stazioni adiacenti su una linea in una determinata direzione, ovvero la stazione di partenza e la stazione di arrivo per quella tratta. Per modellarlo è stata creata le seguente regola:

```
tratta(Linea,Dir,SP,SA) :- percorso(Linea,Dir,LF), member_pair(SP,SA,LF).
```

Con il concetto di fermata si intende le stazioni in cui i treni di una determinata linea si fermano per fare salire e scendere l'agente. Per modellarlo è stata creata la seguente regola:

```
fermata(Stazione,Linea) :- percorso(Linea,0,P), member(Stazione,P).
```

Stazioni

Il concetto di stazione vero e proprio, che rappresenta un aspetto fondamentale del dominio, è stato modellato attraverso dei fatti così strutturati:

```
stazione(Stazione, Coord1, Coord2)
```

La variabile *Stazione* indica il nome della stazione, mentre le variabili *Coord1* e *Coord2* rappresentano rispettivamente la latitudine e la longitudine reale della stazione (i dati sono stati recuperati da un file CSV). Tali coordinate, attraverso degli opportuni calcoli, permettono di calcolare il costo – in minuti – necessario per percorrere una tratta tra due stazioni adiacenti. Oltre a ciò, possono essere utilizzate per il calcolo della distanza in linea d'aria tra due stazioni qualsiasi, informazione utilizzata da un'euristica.

Coppie di stazioni adiacenti

Un terzo concetto, modellato attraverso l'uso di fatti, è stato quello delle coppie di stazioni adiacenti fra di loro. Tale aspetto ha reso il calcolo di un'euristica molto più rapido dell'utilizzo della semplice tratta, la quale permetteva il ripetersi di alcune situazioni – stesse stazioni adiacenti fra di loro ma in linee differenti – che ne rallentavano il funzionamento. La modellazione del fatto è avvenuta con la seguente struttura:

```
coppie_stazioni(Stazione1, Stazione2, Dir).
```

Le variabili *Stazione1* e *Stazione 2* indicano la coppia di stazioni adiacenti fra di loro, ovvero la stazione da cui l'agente può partire e quella in cui può arrivare con una linea qualsiasi della metropolitana e senza la presenza di stazioni intermedie. La variabile Dir, la quale può assumere il valore 0 o 1, ha il semplice scopo di evitare la riscrittura delle coppie di stazioni con i nomi invertiti. A tale scopo è stata creata la seguente regola che permette di invertire le coppie:

```
coppie_stazioni(SP, SA, 1) :- coppie_stazioni(SA, SP, 0).
```

Numero minimo di cambi tra coppie di linee della metropolitana

Un ultimo aspetto modellato per il dominio è stato quello del numero minimo di cambi che un agente deve compiere se si trova su un treno di una linea per arrivare su un altro di un'altra linea. Tale aspetto, che si è reso molto utile per velocizzare il calcolo di un'euristica, è stato modellato attraverso la descrizione di fatti così strutturati:

```
cambi(Linea1, Linea2, Dir, N_Cambi)
```

La variabile *Linea1* e *Linea2* indicano rispettivamente due linee della metropolitana differenti tra di loro, mentre la variabile *N_Cambi* indica il numero di cambi che un agente deve necessariamente fare per passare da una linea all'altra. La variabile *Dir* – come in precedenza – ha lo scopo di evitare la riscrittura delle coppie di linee invertite e può assumere valore 0 e 1. A tale scopo è stata creata una regola per invertire le coppie di linee:

```
cambi(Linea1, Linea2, 1, Cambi) :- cambi(Linea2, Linea1, 0, Cambi).
```

Stato dell'agente

Per quanto riguarda il calcolo delle soluzioni è stato creato il concetto di stato dell'agente, il quale indica la sua posizione attuale nella metropolitana di Londra. Tale stato viene descritto dalla seguente coppia:

```
[at(Stazione), Location]
```

Il primo funtore *at(Stazione)* serve ad indicare la stazione in cui l'agente si trova in un determinato istante, mentre la variabile *Location* serve per indicare se l'agente si trova a terra, "*ground*", o su un treno di una linea metropolitana in una determinata direzione, "*in(Linea, Dir)*".
A partire dalla descrizione di stato dell'agente è stato possibile realizzare lo stato iniziale in cui l'agente si trova all'inizio del problema e lo stato finale, il quale deve essere raggiunto dall'agente:

```
iniziale([at(Stazione), ground]).

finale([at(Stazione), ground]).
```

Azioni che l'agente può compiere

Per cercare di raggiungere lo stato finale a partire da quello iniziale l'agente deve potersi muovere da uno stato ad un altro. A tale scopo esistono tre azioni che l'agente può compiere in particolari situazioni e con costi differenti fra di loro: "sali", "scendi" e "vai".
L'azione "sali" rappresenta l'azione di salire sul treno di una linea *Linea* passante per la stazione in cui l'agente si trova in una determinata direzione *Dir*. Tale azione può essere eseguita nel momento in cui lo stato dell'agente ha come *Location* "*ground*" e il suo costo è di 4 minuti, a causa della possibilità di spostamenti per prendere linee diverse nella stessa stazione e ai tempi di attesa per il passaggio dei treni. L'effetto di tale azione è quello di trasformare la *Location* dell'agente in *in(Linea, Dir)*.
L'azione di "scendi" rappresenta l'azione di scendere dal treno in cui l'agente si trova per andare a terra in una stazione. Tale azione può essere eseguita quando l'agente si trova su una qualsiasi linea, ovvero il suo stato corrente ha come *Location* il valore *in(Linea, Dir)*, e il costo nel compierla è di 1 minuto. L'effetto di tale azione è quello di trasformare la *Location* in "*ground*". Per sperimentare il comportamento dovuto alla chiusura di alcune stazioni è stata creata una versione differente di questa azione, la quale vieta ad un agente di scendere dal treno in tali situazioni.

L'azione di "vai" rappresenta l'azione di spostarsi da una stazione alla successiva – una tratta – sul treno della linea e direzione in cui si sta viaggiando. Tale azione può essere eseguita quando l'agente si trova su una linea, ovvero il suo stato corrente ha come *Location* il valore *in(Linea, Dir)*, e il suo costo è calcolato come il tempo necessario per spostarsi da una stazione all'altra ad una velocità media di 36km/h. Per il calcolo abbiamo utilizzato la distanza in linea d'aria tra le stazioni, calcolata a partire dalle coordinate reali delle stesse. L'effetto di tale azione è di spostare l'agente nella stazione di destinazione, ossia cambiare il valore di *at(Stazione)* con la stazione di destinazione della tratta.

Implementazione algoritmi di ricerca

Dopo la modellazione del dominio si è passati alla realizzazione degli algoritmi di ricerca, il cui scopo è quello di permettere il calcolo delle possibili soluzioni. Con soluzioni si intende la sequenza di azioni che permette ad un agente di arrivare da un punto di partenza ad un punto di arrivo.

Iterative Deepening

L'algoritmo Iterative Deepening è un algoritmo ottimo, non informato – il costo delle azioni è unitario e non cambia da azione ad azione – e che si sviluppa in profondità iterativamente, ovvero aumentato ad ogni iterazione una soglia che indica la profondità massima da raggiungere per l'iterazione corrente.
L'idea base per l'implementazione è stata quindi quella di partire da una normale ricerca in profondità, e aggiungere un controllo in grado di verificare che la profondità raggiunta non fosse superiore a quella della soglia impostata all'iterazione corrente. Nel dettaglio, la soglia viene decrementata ad ogni livello a cui si arriva e non può scendere al di sotto di 0. Alla conclusione di ogni iterazione – non sono state trovate soluzioni alla profondità corrente – la soglia viene aumentata di uno e si ricomincia la computazione dall'inizio.

A star

L'algoritmo A Star è un algoritmo ottimo e basato su euristiche, che si sviluppa in ampiezza. L'implementazione realizzata fa uso del concetto di nodi per mantenere informazioni sullo stato raggiunto:

```
nodo(S, AzPerS, CostoAzPerS, EurDaS)
```

La variabile *S* indica lo stato corrente dell'agente, mentre le altre variabili indicano rispettivamente: le azioni per arrivarci, il costo delle azioni per arrivarci – *g(n)* - e la stima del costo per arrivare allo stato finale a partire da *S* - *h(n)* -. Tali nodi vengono man mano espansi derivando i figli in base alla lista di azioni che possono essere effettuate dallo stato corrente. I figli così generati vengono inseriti in una coda di priorità, il cui scopo è quello di mantenere i nodi ordinati in maniera crescente per *f(n)*, dove *f(n)* è la somma di *g(n)* e *h(n)*. Ricorsivamente viene richiamato l'algoritmo di A Star sul nodo in cima alla coda di priorità, il quale viene rimosso e aggiunto ai nodi visitati.
Per quanto riguarda l'espansione di un nodo viene eseguita prendendo in considerazione un'azione possibile alla volta ed andando a creare il nodo che deriva dall'esecuzione dell'azione, tenendo conto dell'aggiornamento del costo e dell'euristica. Una volta creato il nodo si continua con l'azione successiva, fino all'esaurimento delle stesse, e a restituire la lista di nodi figli del nodo di partenza. È necessario prestare attenzione quando si cerca di creare un nodo figlio già visitato, il cui costo per arrivarci può essere differente. In questo caso bisogna riconsiderare il nodo solo se il percorso per raggiungere lo stato ha un costo inferiore al precedente, cosa non possibile se l'euristica è ammissibile e consistente. Per risolvere il problema è stata implementata la regola *revisit_node* che dice se un nodo è da rivisitare o meno, e in caso affermativo restituisce la nuova lista di visitati senza il vecchio stato.

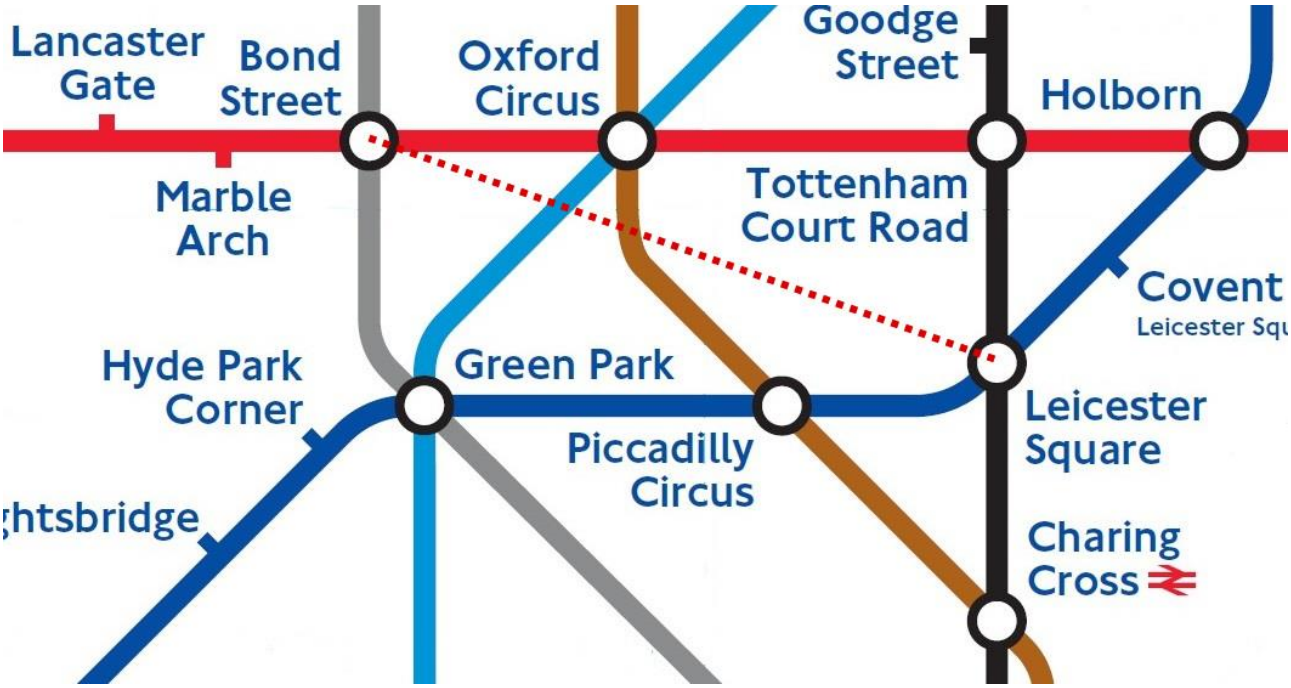
IDA star

L'algoritmo IDA Star è un algoritmo di ricerca ottimo e basato su euristiche, che si sviluppa in profondità. Tale algoritmo è simile all'Iterative Deepening, con la differenza dell'aggiornamento della soglia ad ogni iterazione; in questo caso la soglia non viene incrementata costantemente di uno, ma viene aggiornata con la *f(n)* minima tra i nodi che hanno una *f(n)* che ha superato la soglia corrente.
Un'ulteriore differenza sta nel fatto che anche in questo caso si utilizza il concetto di nodo (rispetto alla struttura vista in precedenza vengono escluse le azioni per arrivare allo stato corrente).
Un problema dell'algoritmo risiede nell'estrarre la nuova soglia a partire dalle possibili soglie candidate. Per farlo è stato necessario l'utilizzo di un fatto dinamico – attraverso l'uso delle regole *retract* e *asserta*, utili rispettivamente per ritrarre il vecchio valore e asserire quello nuovo – che assume, man mano che i vari rami vengono visitati, il valore possibile della nuova soglia. Se tale soglia possibile non deve essere aggiornata la computazione fallisce, e quindi si riparte dai punti di scelta precedenti. Se invece tale soglia dev'essere

aggiornata, viene eseguito l'aggiornamento come descritto in precedenza e la computazione avrebbe successo portando alla conclusione dell'algoritmo, senza però una soluzione. Per questo motivo viene aggiunto un *fail* dopo l'aggiornamento, che permette di continuare l'esplorazione dei rami rimanenti (che potrebbero avere a loro volta una possibile soglia più piccola di quella appena considerata).
Alla fine dell'iterazione la soglia possibile rimasta diventa il valore effettivo della soglia e il fatto dinamico viene re-inizializzato per l'iterazione successiva.

Euristiche

Per il funzionamento degli algoritmi informati appena descritti – A Star e IDA Star – sono state implementate differenti euristiche ammissibili – requisito necessario per l'ottimalità dalla soluzione –, le quali ci hanno permesso di verificare le prestazioni degli algoritmi sulla base di stime più o meno accurate. In particolare, sono state realizzate tre euristiche di base, che cercano di rilassare i vincoli imposti dal problema seguendo idee differenti, e due euristiche che vanno a combinare le euristiche base.
Per l'illustrazione di tali euristiche si farà riferimento all'immagine che segue, la quale rappresenta un frammento della metropolitana di Londra.



Distanza in linea d'aria

La prima e più semplice euristica realizzata è quella che calcola i minuti necessari per spostarsi dalla stazione corrente alla stazione di destinazione attraverso il percorso più breve possibile tra le stesse, ovvero il percorso equivalente alla distanza in linea d'aria – la

linea tratteggiata nell'immagine sopra. Tale euristica risulta essere ammissibile, infatti come detto in precedenza il percorso calcolato sarà sicuramente minore tra tutti quelli possibili – disuguaglianza triangolare – e inoltre non considera i tempi di salita e discesa dai treni.
Per calcolare tale euristica si usano le coordinate reali delle stazioni e da questi si calcola la distanza. A partire dalla distanza si calcolano i minuti necessari per spostarsi ad una velocità costante di 36 km/h.

Numero minimo di salite e discese

La seconda euristica base realizzata si concentra sugli altri due aspetti del problema prima non considerati: il tempo di salita e di discesa dalle linee della metropolitana. In particolare, tale euristica calcola il numero minimo di salite e discese che un agente deve compiere per arrivare dallo stato corrente alla stazione di destinazione. A tale scopo vengono utilizzati i fatti che indicano il numero di cambi minimo necessari per passare da una linea all'altra.
In quest'euristica ci si può trovare in tre casi distinti:

- **L'agente si trova nella stazione di destinazione, a terra**: in questo caso non devo più muovermi per raggiungere il mio obiettivo e quindi non devo più salire e scendere dalla metro; ciò comporta un costo di 0 minuti;
- **L'agente si trova in una stazione diversa da quella di destinazione, a terra**: in questo caso si andrà a vedere il numero minimo di linee che si devono prendere per arrivare a destinazione; si considerano tutte le linee che passano da dove mi trovo e da dove devo arrivare a coppie, e si prende il valore minore. Tale valore sarà il numero minimo di cambi n, equivalente ad un numero n di salite e ad un numero n di discese, al quale deve essere aggiunto il costo di salita dalla stazione di partenza e quello di discesa nella stazione di arrivo;
- **L'agente si trova su una linea**: in questo caso si andrà a vedere il numero di cambi dalla linea in cui mi trovo nello stato corrente a tutte quelle che passano dallo stazione di arrivo, scegliendo tra tutti i valori possibili quello minimo. Come in precedenza, il numero di cambi n equivale a n salite e n discese, al quale devo aggiungere il costo di discesa nella stazione di destinazione.

Tale euristica risulta essere ammissibile, infatti il costo complessivo sarà sempre inferiore al costo stimato; sicuramente si avranno sempre un numero di cambi necessari maggiori o uguali a quelli determinati dall'euristica, e mai inferiori. Come si vede dall'immagine, se l'agente si trova a terra a Bond Street e deve arrivare a Leicester Square, l'euristica determinerà un numero di cambi di linea minimo pari a uno – rossa/nera, grigia/blu – con

un costo complessivo di 1*(4 + 1) + (4 +1) = 10, ma nella realtà potrei avere una soluzione con un cambio o più – rossa/marrone/blu – ma mai di meno.

Percorso minimo senza salite e discese

La terza euristica base - la più complessa delle tre - per calcolare la stima fa uso di A Star che lavora su un dominio semplificato. In particolare, l'algoritmo di ricerca calcola il percorso più breve per arrivare da una stazione di partenza ad una di arrivo passando per le varie stazioni, ma escludendo il fatto di dover scendere e salire per cambiare linea; in altre parola è come se si avesse una grande metro con un'unica linea, che può portare in tutte le stazioni seguendo i percorsi delle varie linee a disposizione. A tale proposito vengono utilizzati i fatti che indicano le coppie di stazioni adiacenti tra di loro, ovvero stazioni collegate da una qualsiasi linea.
Tale euristica risulta essere ammissibile, infatti il percorso calcolato è il più breve possibile e di conseguenza il percorso della soluzione reale sarà al più uguale a quello calcolato, con l'aggiunta degli eventuali cambi di metro.

Euristiche composte

Le euristiche composte realizzate sono due:

- Distanza in linea d'aria + Numero minimo di salite e discese
- Percorso minimo senza discese e salite + Numero minimo di salite e discese

Tali euristiche sfruttano una euristica che calcola un percorso – sempre inferiore a quello reale – e un'euristica che calcola il numero minimo di salite e discese necessarie per arrivare a destinazione dal punto in cui l'agente si trova. Queste coppie di euristiche non si sovrappongo mai e di conseguenza le euristiche ottenute sono a loro volta ammissibili.

Esperimenti

Vengono riportati alcuni esperimenti effettuati. Si annotano le elaborazioni per ciascun algoritmo (e ciascuna euristica per gli algoritmi informati) sulla zona 1 e 3.

Zona 1

Stazione di partenza: ***Bayswater*** Stazione di arrivo: ***Southwark***

Algoritmo	Euristica	Tempo (ms)	Nodi espansi
Iterative deepening	-	58	-
A Star	1 - Linea d'aria	55	216
A Star	2 - Min. Salite e Discese	20	128
A Star	3 - A Star senza salite e discese	240	130
A Star	Linea d'aria + Min. Salite e Discese (1,2)	3	17
A Star	A Star senza salite e discese + Min. Salite e Discese (3,2)	35	14
IDA Star	1 - Linea d'aria	140	-
IDA Star	2 - Min. Salite e Discese	730	-
IDA Star	3 - A Star senza salite e discese	Computazione molto onerosa	-
IDA Star	Linea d'aria + Min. Salite e Discese (1,2)	8	-
IDA Star	A Star senza salite e discese + Min. Salite e Discese (3,2)	Computazione molto onerosa	-

Si consideri A Star. Si osserva che il passaggio di utilizzo dalla distanza in linea d'aria all'euristica del minimo numero di salite e discese, determina un'efficienza maggiore, grazie al fatto che la seconda euristica può essere più precisa in distanze non troppo elevate, e non è troppo complessa da calcolare. Come si vedrà per la zona 3, questo non sarà più vero, poiché la distanza in linea d'aria su distanze elevate permette di aver una stima più accurata, unita ad una computazione molto snella.

Il miglioramento di prestazioni invece non si verifica per quanto riguarda l'euristica che usa a sua volta l'algoritmo A Star non considerando le salite e le discese: infatti, A Star diventa molto più lento, anche se comunque i nodi espansi rimangono quasi gli stessi. Evidenti aumenti di prestazioni si hanno nuovamente con le euristiche accoppiate: la prima coppia di euristiche permette la migliore velocità di elaborazione. Questo è dovuto al fatto che

coinvolge euristiche molto veloci da calcolare (soprattutto la linea d'aria) e abbastanza precise. La seconda coppia invece accusa un peggioramento sulla velocità, ma consente di avere il minimo numero di nodi espansi, poiché l'utilizzo della terza euristica è molto più focalizzata.

IDA Star trae giovamento dalle euristiche che richiedono computazioni più semplici, come nel caso della distanza in linea d'aria. Come si osserva dalla tabella, la prima euristica è molto più veloce della seconda, che a sua volta è più lenta da calcolare rispetto alla prima. Il risultato più soddisfacente si ottiene con la prima coppia di euristiche: unendo la velocità di calcolo della distanza in linea d'aria con la precisione su distanze brevi della seconda euristica, si ottiene il miglior tempo per IDA Star.

Zona 3

Stazione di partenza: ***Ealing Broadway*** Stazione di arrivo: ***North Greenwich***

Algoritmo	**Euristica**	**Tempo (ms)**	**Nodi espansi**
Iterative deepening	-	4430	-
A Star	1 - Linea d'aria	95	396
A Star	2 - Min. Salite e Discese	350	790
A Star	3 - A Star senza salite e discese	390	187
A Star	Linea d'aria + Min. Salite e Discese (1,2)	10	51
A Star	A Star senza salite e discese + Min. Salite e Discese (3,2)	70	31
IDA Star	1 - Linea d'aria	510	-
IDA Star	2 - Min. Salite e Discese	Computazione molto onerosa	-
IDA Star	3 - A Star senza salite e discese	Computazione molto onerosa	-
IDA Star	Linea d'aria + Min. Salite e Discese (1,2)	120	-

IDA Star	A Star senza salite e discese + Min. Salite e Discese (3,2)	Computazione molto onerosa	-

L'aumento di distanza tra la stazione di partenza e di arrivo fa sì che A Star si comporti in modo leggermente differente. La distanza in linea d'aria appare molto più efficiente della seconda euristica, contrariamente a quanto accadeva nel caso precedente. Per ottenere un nuovo miglioramento, occorre utilizzare la terza euristica, che grazie alla precisione permette di espandere molti meno nodi, a scapito del tempo di esecuzione. Si nota la differenza rispetto all'esperimento condotto sulla zona 1: la terza euristica, su lunghe distanze, è molto più efficace, anche se più lenta.
Le prestazioni migliori però si ottengono ancora una volta con le coppie di euristiche: la prima coppia consente un tempo di esecuzione migliore di tutti gli altri, e molti meno nodi espansi rispetto alla terza euristica. La seconda coppia invece migliora ancora il numero di nodi espansi, a scapito ancora della velocità di esecuzione causata dalla terza euristica nella coppia.

Per quanto riguarda IDA Star, il comportamento rimane simile al precedente, con la differenza che l'elaborazione con la seconda euristica risulta molto più lenta, poiché quest'ultima richiede più sforzo computazionale. Ci si potrebbe chiedere come questa sia molto più onerosa da computare, mentre la coppia di euristiche 1 e 2 no (l'euristica di cui si parla è qui compresa): questo è perché la distanza in linea d'aria è veloce da calcolare, e inoltre permette di focalizzarsi meglio dell'euristica 2 sull'obiettivo (vero sulle grandi distanze).

Zona 3, con stazioni chiuse

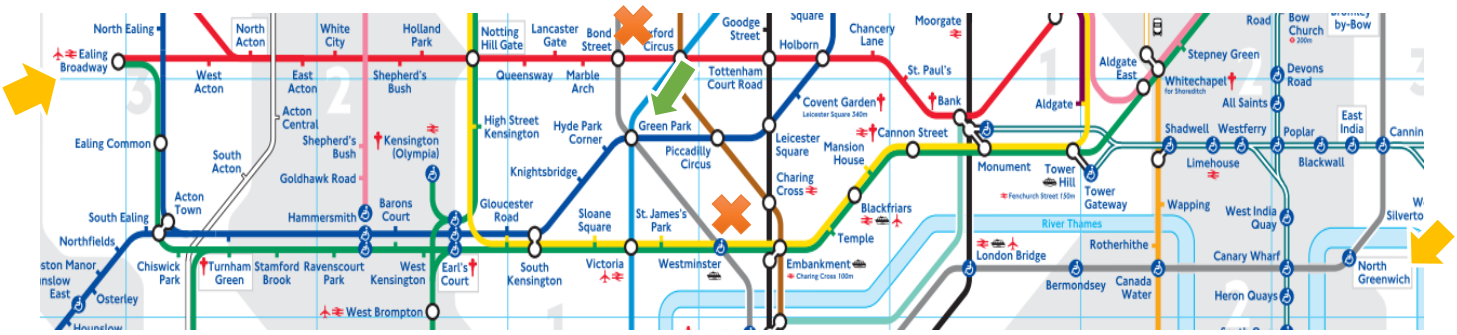
Stazione di partenza: ***Ealing Broadway*** Stazione di arrivo: ***North Greenwich***
Stazioni chiuse: ***Westminster, Bond Street***

Algoritmo	**Euristica**	**Tempo (ms)**	**Nodi espansi**
Iterative deepening	-	5500	
A Star	1 - Linea d'aria	250	696
A Star	2 - Min. Salite e Discese	360	951

A Star	3 - A Star senza salite e discese	1070	418
A Star	Linea d'aria + Min. Salite e Discese (1,2)	50	234
A Star	A Star senza salite e discese + Min. Salite e Discese (3,2)	260	122
IDA Star	1 - Linea d'aria	19000	-
IDA Star	2 - Min. Salite e Discese	Computazione molto onerosa	-
IDA Star	3 - A Star senza salite e discese	Computazione molto onerosa	-
IDA Star	Linea d'aria + Min. Salite e Discese (1,2)	2000	-
IDA Star	A Star senza salite e discese + Min. Salite e Discese (3,2)	Computazione molto onerosa	-

L'esperimento con le stazioni chiuse è stato riportato per analizzare quanto peggiorano gli algoritmi di ricerca in generale. Si può notare che A Star, rispetto all'esperimento precedente, abbia espanso molti più nodi. Questo perché siccome l'agente non può scendere presso una stazione chiusa, A Star ha dovuto esplorare altre vie per raggiungere la destinazione. In particolare, per andare da Ealing Broadway fino a North Greenwich, l'agente senza stazioni chiuse prenderebbe la metro verde, cambierebbe a Westminster e proseguirebbe sulla grigia fino a destinazione. Se fosse chiusa solo *Westminster*, l'agente non cambierebbe più in quest'ultima, bensì prenderebbe la metro rossa e cambierebbe a *Bond Street*, esplorando ovviamente più nodi e aumentando il tempo computazionale. Se infine *Bond Street* fosse chiusa, l'agente si vedrebbe costretto a ripiegare sulla metro blu, cambiando a *Green Park*, con ulteriori nodi esplorati.

In generale, si nota che l'andamento delle computazioni segue quello dell'esperimento sulla zona 3 (senza le stazioni chiuse).



Gironi Champions League (ASP)

Per la risoluzione del problema di soddisfacimento di vincoli, in grado di realizzare i gironi di Champions League e le partite conseguenti, siamo partiti dal file suggeritoci, dai quali abbiamo preso le 32 squadre e realizzato i letterali ground che le rappresentano:

```
dati_squadra(Nome, Nazione, Citta, Fascia).
```

In tali letterali sono rispettivamente indicati il *Nome*, la *Nazione* e la *Città* della squadra descritta, oltre che la *Fascia* di appartenenza. In totale si ha la presenza di 4 differenti fasce, con equo numero di squadre – 8 squadre – in ognuna, indicate dai valori 1, 2, 3, 4; 1 indica la fascia di squadre più forti, mentre 4 la fascia di squadre più deboli. Tale dato permette di creare dei gironi con 4 squadre appartenenti a fasce differenti, come succede nella realtà.
Un ulteriore aspetto del dominio modellato attraverso dei letterali ground è quello che rappresenta i gironi possibili:

```
girone(Nome, Colore).
```

In totale sono stati rappresentati 8 gironi, i cui nomi sono rappresentati da una lettera compresa tra la A e la H, divisi in due gruppi da 4 gironi: i primi 4 – dalla A alla D – di colore rosso e gli ultimi 4 – dalla E alla H – di colore blu. Tale distinzione di colori, presente nella realtà, permette di distribuire equamente le squadre della stessa nazione tra i due possibili gruppi.
A partire da questi dati siamo stati in grado di risolvere il problema andando a soddisfare tutti i vincoli imposti dallo stesso, oltre che ad aggiungere ulteriori vincoli – come quelli sopra descritti – che permettano di avvicinare il problema maggiormente alla realtà.

Composizione dei gironi

Per la composizioni degli 8 gironi da 4 squadre ciascuno, siamo partiti dalla definizione di un letterale in grado di rappresentare l'appartenenza di una squadra ad un determinato girone:

```
assegna_girone(Squadra,Nazione,Girone)
```

Tali letterali permettono di indicare a che *Girone* appartiene una determinata *Squadra*, indicando anche la *Nazione* di quest'ultima per il soddisfacimento dei vincoli futuri, e vengono creati attraverso l'utilizzo di due aggregati: il primo in grado di assegnare ad ogni

squadra un unico girone, e il secondo in grado di assegnare ad ogni girone esattamente 4 squadre.
Attraverso questi aggregati è stato dunque possibile creare dei primi modelli di girone molto grezzi, i quali per l'appunto non rispettano ancora tutti i vincoli imposti dal problema. Per raffinare i risultati abbiamo quindi introdotto una prima serie di integrity constraint:

- Le squadre dello stesso girone non possono essere della stessa nazione;
- Le squadre della stesso girone non possono essere della stessa fascia;
- Squadre Russe e Ucraine non possono essere nello stesso girone – dovuto ai conflitti politici (in riferimento alla realtà della Champions 2018-2019);

Per consentire il rispetto di questi vincoli ci siamo avvalsi di alcune regole, come ad esempio la seguente utilizzata nel primo vincolo sulle nazioni:

```
stessa_nazione(SquadraX,SquadraY) :-
  dati_squadra(SquadraX,Naz,_,_),
  dati_squadra(SquadraY,Naz,_,_),
  SquadraX != SquadraY.
```

Tale regola consente di verificare se 2 squadre differenti appartengono alla stessa nazione. Similmente sono state prodotte regole per squadre della stessa fascia, squadre di nazionalità Russa e squadre di nazionalità Ucraina.
L'unico vincolo mancante per rendere i gironi definitivi a questo punto è quello riguardante l'equa suddivisione delle squadre della stessa nazione tra i gironi rossi e i gironi blu: ad esempio, se ho 3 squadre di una nazione ne posso mettere 2 in gironi rossi e 1 in gironi blu. Per far ciò abbiamo creato una regola per recuperare tutte e sole le nazioni delle squadre presenti. Tali nazioni sono state utilizzate per fare dei conteggi sulle squadre attraverso 3 opportune regole:

- Contare il numero di squadre di ogni nazione;
- Contare il numero di squadre nei gironi rossi di ogni nazione;
- Contare il numero di squadre nei gironi blu di ogni nazione.

Queste regole sono state utilizzate attraverso l'utilizzo dell'operatore di aggregazione *#count*, il quale permette di eseguire dei conteggi da assegnare ad una variabile:

```
n_squadre_per_nazione(Naz, N) :- N = #count{Squadra : dati_squadra(Squadra,Na
z,_,_)}, nazione(Naz).
```

A partire da queste regole sono stati poi creati due integrity constraint in grado di rispettare quanto detto in precedenza. Per far ciò, prendono i rispettivi conteggi e utilizzando dei semplici calcoli matematici – la moltiplicazione per 10 evita risultati con la virgola, la

divisione per 2 calcola la metà delle squadre e l'aggiunta di 5 risolve le situazioni in cui il numero di squadre della stessa nazione è dispari – verificano che la disuguaglianza sia corretta.

Composizione delle partite

Per la composizione delle 6 partite che ogni squadra deve disputare nell'arco di 6 giornate siamo partiti dalla creazione di un letterale in grado rappresentare tutte le squadre con le quali una squadra deve giocare in casa:

```
scontri(SquadraX,SquadraY,Girone)
```

Tale letterale, che oltre alle 2 squadre impegnate, indica anche il *Girone* delle stesse, viene creato a partire da un aggregato che, per ogni *SquadraX* di un girone, assegna esattamente le altre 3 squadre dello stesso girone con la quale tale *SquadraX* deve giocare in casa.
A partire da tali scontri e da dei letterali ground *giornata* – con valori da 1 a 6 – vengono composte le partite vere e proprie:

```
partita(SquadraX,SquadraY,Giornata,Girone)
```

Quest'ultime descrivono tutte le coppie di squadre, dello stesso girone, che si scontrano in una determinata giornata e viene creata attraverso l'uso combinato di due aggregati: il primo assegna ad ogni scontro un'unica giornata e il secondo assegna ad ogni giornata esattamente 16 scontri. L'insieme di tali partite rappresentano per ogni squadra tutti i match da disputare in casa e in trasferta.
A questo punto della computazione si hanno quindi le giornate composte, ma in queste potrebbe capitare che la stessa squadra giochi più partite. Tale situazione è da evitare dato che ogni squadra deve necessariamente giocare un'unica partita – in casa o in trasferta – per giornata. Per far ciò sono stati creati 3 vincoli in grado di risolvere il problema:

- Non devono esserci 2 partite in casa di una squadra X contro squadre diverse nella stessa giornata;
- Non devono esserci 2 partite in trasferta di una squadra X contro squadre diverse nella stessa giornata;
- Non devono esserci 2 partite, una in casa e una in trasferta, di una squadra X contro squadre diverse o la stessa squadra nella stessa giornata.

Tali vincoli permettono una migliore realizzazione delle partite, non tenendo conto però che 2 squadre della stessa città non possono giocare in casa entrambe nella stessa giornata. Per eliminare i modelli che non rispettano questa situazione è stato creato un integrity constraint

a tal proposito, il quale fa uso di una regola – di costruzione simile a quella delle nazioni – in grado di verificare se 2 squadre diverse fanno parte della stessa città.
Un ulteriore vincolo rispettato è stato quello di evitare che una stessa squadra giocasse più di 2 partite consecutive in casa o in trasferta. Per far ciò è stato necessario creare 2 integrity constraint, i quali, prese le 3 partite in casa o in trasferta della stessa squadra, controllano che le rispettive giornate non siano tutte 3 consecutive.
Per finire abbiamo deciso di realizzare un ultimo vincolo in grado di suddividere le partite in andata e ritorno; in altre parole una stessa squadra non può giocare contro la medesima squadra in casa e in trasferta nelle prime o ultime 3 giornate, il che vuol dire avere le prime 3 giornate di andata e le ultime 3 giornate di ritorno.