Università degli studi di Torino

Dipartimento di Informatica

Intelligenza Artificiale e Laboratorio

Relazione di progetto

Prolog e Clingo

Studente: Lorenzo Botto a.a. 2021/2022

Indice

1. Prolog

Per la realizzazione del progetto in Prolog, ho scelto il gioco del puzzle scorrevole da 8 che viene giocato su una griglia 3 per 3 con 8 tessere quadrate etichettate da 1 a 8, più un quadrato vuoto. L'obiettivo è riorganizzare le tessere in modo che siano in ordine. Si può far scorrere le tessere orizzontalmente o verticalmente nel quadrato vuoto.

1.1 Descrizione implementazione

Per l'implementazione del progetto, ho sviluppato due versioni di cui entrambe utilizzano l'algoritmo A* per la ricerca nello spazio degli stati ma con due euristiche differenti:

- euristica delle caselle fuori posto: conta le caselle fuori posto dello stato considerato rispetto allo stato goal;
- euristica della distanza di Manhattan: calcola il numero di azioni necessarie per portare una casella nel posto corretto, per ogni casella, e infine somma tutti i valori.

1.2 Com'è stato implementato

Il progetto si compone di tre file principali:

- dominio.pl: dove sono presenti lo stato iniziale e lo stato finale da raggiungere;
- regole.pl: dove si controlla che una certa azione sia applicabile e si trasforma uno stato nello stato successivo applicando una determinata azione;
- ricerca.pl: dove contiene il cuore dell'algoritmo di ricerca.

Si parte da uno stato iniziale applicando l'algoritmo di ricerca/6 con questi parametri:

- stato corrente da considerare;
- numero dei passi per arrivare allo stato corrente;
- lista degli stati visitati;

- lista delle frontiere;
- lista dei visitati, in un formato diverso dalla precedente che mi servirà per il controllo delle mosse possibili, in modo che non venga aggiunta una frontiera con uno stato già visitato. È una struttura ausiliaria che tengo perché se usassi l'altra lista dei visitati i tempi sarebbero maggiori in quanto ha una struttura diversa (la struttura viene spiegata in seguito);
- lista della azioni, che conterrà il risultato finale.

La ricerca si compone delle seguenti fasi:

- si trovano tutte le mosse applicabili allo stato considerato, controllando che sia applicabile tramite il predicato applicabile/2;
- si calcola il costo di ogni mossa possibile applicando questi passi:

    - viene trasformato lo stato corrente in uno stato nuovo controllando che non sia già stato visitato, altrimenti non viene inserito in una lista delle frontiere e si passa a considerare la prossima mossa possibile;
    - vengono contate le differenze tra lo stato nuovo e lo stato finale, ovvero il numero di caselle fuori posto tra lo stato nuovo e lo stato finale;
    - il costo totale sarà dato dal numero di caselle fuori posto sommato al numero delle azioni da fare per arrivare allo stato nuovo;
    - viene inserito in una lista delle frontiere un elemento con questo formato [costo_totale, [lista_stato_corrente], [lista_stato_nuovo], numero_passi]

- si aggiungono le nuove frontiere calcolate alla lista delle frontiere già calcolate precedentemente;
- si estrae la frontiera di costo minimo dalla lista delle frontiere (nel formato descritto sopra);
- si elimina la frontiera di costo minimo dalla lista delle frontiere;
- si richiama l'algoritmo di ricerca sullo stato nuovo, fino a quando lo stato nuovo non è uguale allo stato finale e si termina la ricerca, con questi parametri:

- stato nuovo, estratto dalla lista delle frontiere, di costo minimo;
- numero dei passi estratto dalla frontiera di costo minimo sommando 1 perché per arrivare allo stato nuovo ci sarà un'azione in più da effettuare;
- lista dei visitati, a cui aggiungo lo stato nuovo nel formato [stato_nuovo, stato_precedente] in quanto mi servirà per ricostruirmi il percorso all'indietro una volta terminata tutta la ricerca;
- la lista delle frontiere modificata come descritto in precedenza;
- lista dei visitati nel formato pulito [stato_visitato] a cui viene aggiunto lo stato corrente che ho visitato.
- lista delle azioni che conterrà il risultato.

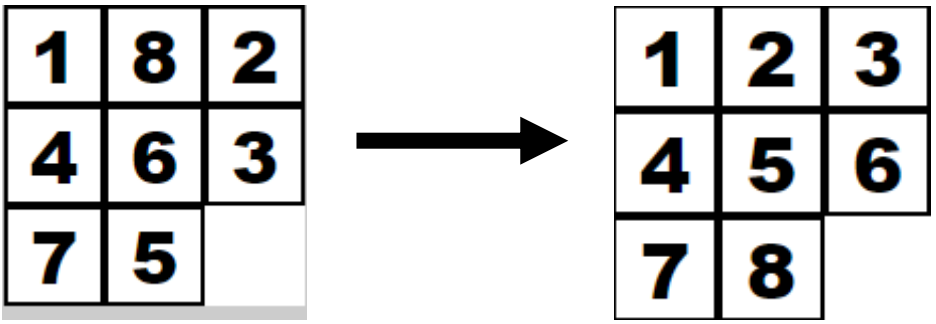
Alla fine dell'algoritmo di ricerca, quando lo stato considerato è uguale allo stato finale, tramite la lista dei visitati dove ogni elemento ha il formato [stato_nuovo, stato_precedente] si ricalcola il percorso all'indietro trovando la mossa applicata per passare dallo stato precedente allo stato nuovo e riapplicando il calcolo allo stato precedente fino ad arrivare allo stato iniziale. Si salvano le mosse calcolate in una lista che verrà restituita alla fine come risultato dell'algoritmo.

Per quanto riguarda l'algoritmo con l'euristica della distanza di Manhattan è uguale, solo che non verranno contate le caselle fuori posto e il costo totale sarà diverso, ma il procedimento è lo stesso. In questo caso verrà calcolato il numero di azioni necessarie per portare una casella nel posto corretto, per ogni casella, e infine vengono sommati tutti i valori.

1.3 Risultati

1.3.1 Primo caso

Primo caso semplice di risoluzione del problema con lo stato iniziale [1,8,2,4,6,3,7,5,0].



I risultati con l'euristica delle caselle fuori posto sono soddisfacenti:

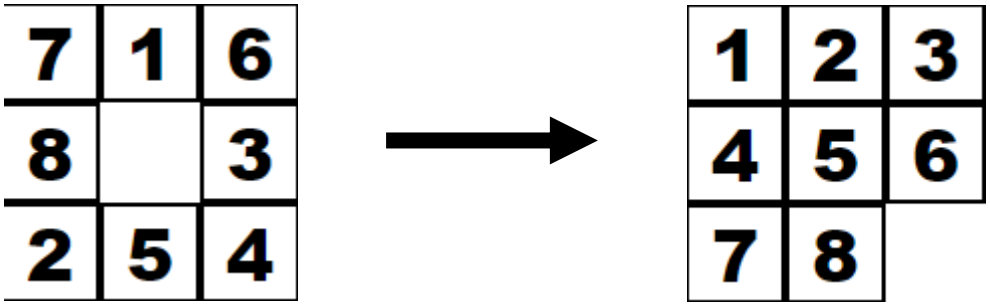
```
2 ?- statistics(cputime, TStart), prova(L), write(L), statistics(cputime, TEnd), T is TEnd-TStart.
[sinistra,sopra,sopra,destra,sotto,sotto,sinistra,sopra,destra,sotto]
TStart = T, T = 0.015625,
L = [sinistra, sopra, sopra, destra, sotto, sotto, sinistra, sopra, destra|...],
TEnd = 0.03125.
```

Stessa identica cosa per l'euristica di Manhattan:

```
2 ?- statistics(cputime, TStart), prova(L), write(L), statistics(cputime, TEnd), T is TEnd-TStart.
[sinistra,sopra,sopra,destra,sotto,sotto,sinistra,sopra,destra,sotto]
TStart = 0.078125,
L = [sinistra, sopra, sopra, destra, sotto, sotto, sinistra, sopra, destra|...],
TEnd = 0.10937500000000001,
T = 0.031250000000000014.
```

1.3.2 Secondo caso

Secondo caso di risoluzione del problema, molto più complesso, con lo stato iniziale [7,1,6,8,0,3,2,5,4].



I tempi con l'euristica delle caselle fuori sono molto più alti del caso precedente:

```
2 ?- statistics(cputime, TStart), prova(L), write(L), statistics(cputime, TEnd), T is TEnd-TStart.
[sotto,sinistra,sopra,sopra,destra,sotto,destra,sopra,sinistra,sotto,sotto,destra,sopra,sopra,sinistra,
sotto,sotto,sinistra,sopra,destra,destra,sotto]
TStart = 0.04687500000000001,
L = [sotto, sinistra, sopra, sopra, destra, sotto, destra, sopra, sinistra|...],
TEnd = 35.609375,
T = 35.5625.
```

Invece con l'euristica di Manhattan i tempi sono nettamente migliori rispetto all'euristica precedente:

```
2 ?- statistics(cputime, TStart), prova(L), write(L), statistics(cputime, TEnd), T is TEnd-TStart.
[sotto,sinistra,sopra,sopra,destra,destra,sotto,sinistra,sopra,destra,sotto,sinistra,sotto,destra,sopra
,sopra,sinistra,sotto,sotto,sinistra,sopra,destra,destra,sotto]
TStart = 0.0625,
L = [sotto, sinistra, sopra, sopra, destra, destra, sotto, sinistra, sopra|...],
TEnd = 0.6562500000000001,
T = 0.5937500000000001.
```

1.4 Osservazioni finali

A seguito dei risultati ottenuti dalle due casistiche appena mostrate, posso affermare che:

- l'algoritmo A* è ottimo per questo problema e ho ottenuto risultati soddisfacenti che non mi hanno portato a dover utilizzare un altro algoritmo di ricerca;
- è molto più efficiente, in particolare con problemi più complessi, l'euristica della distanza di Manhattan rispetto all'euristica delle caselle fuori posto;
- l'algoritmo A* utilizza molta memoria per mantenere tutte le strutture necessarie a risolvere il problema, in particolare per problemi più complessi dove gli stati da esplorare sono tanti.

1.5 Come eseguire

Una volta ottenuto il codice dal repository di GitHub https://github.com/lorenzobotto/progettoProlog, caricare il puzzle dell'8 con l'euristica voluta:

- `['ricerca.pl'].` per l'euristica delle caselle fuori posto;
- `['ricerca_manhattan.pl'].` per l'euristica della ricerca di Manhattan.

Infine, eseguire e vedere le tempistiche tramite il codice:

```
statistics(cputime, TStart), prova(L), write(L),
statistics(cputime, TEnd), T is TEnd-TStart.
```

2. Clingo

Per la realizzazione del progetto Clingo ho preso in considerazione un campionato di calcio a 20 squadre con quattro derby. Viene generato un calendario con 19 giornate di andata e 19 giornate di ritorno con dei vincoli spiegati nella sezione successiva.

2.1 Descrizione implementazione

Il sistema che ho sviluppato per generare il calendario associa il numero delle giornate al girone di andata o al girone di ritorno, applicando ad entrambi i seguenti vincoli:

1. massimo dieci partite per ogni giornata, utilizzando gli aggregati;
2. se due squadre hanno giocato una partita in una giornata, non ci può essere un'altra partita nella stessa giornata con quelle squadre controllando qualsiasi combinazione: si controlla che la squadra in casa non giochi un'altra partita in casa, ma nemmeno in trasferta e gli stessi controlli vengono applicati per la squadra in trasferta;
3. si controlla che se due squadre hanno giocato una partita in un certo girone, non ci sia un'altra partita in quel girone uguale o a squadre invertite;
4. non ci possono essere due partite nella stessa giornata con le due squadre in casa della stessa città;
5. per ogni giornata, controllo che la squadra in casa di ogni partita non giochi di nuovo in casa per le prossime due partite. Quindi partendo da una giornata G controllo le giornate G+1 e G+2;

Inoltre, vengono applicati questi vincoli:

- si controlla che non ci sia nel girone di ritorno una partita uguale al girone di andata (squadre invertite è accettabile);
- controllo che la partita nel girone di ritorno sia ad almeno 10 giornate di distanza dalla partita con le stesse squadre nel girone di andata, controllando che la sottrazione tra il numero della giornata non sia <= 10. Questo controllo viene effettuato fino alla giornata di ritorno numero 29 perché si controlla fino

a che l'ultima giornata di andata (19) sia a 10 giornate di distanza; quindi, l'ultima giornata di ritorno da considerare è 19+10=29.

2.2 Risultati

Tutte le esecuzioni utilizzano 8 threads, interrompendo le esecuzioni che durano più di 30 minuti.

La prima esecuzione che ho effettuato è un campionato più ristretto con 12 squadre, un solo derby e distanza tra giornata di andata e giornata di ritorno di almeno 6 giornate. Impiega meno di tre secondi:

```
Models       : 1+
Calls        : 1
Time         : 2.793s (Solving: 2.18s 1st Model: 2.13s Unsat: 0.00s)
CPU Time     : 18.047s
Threads      : 8        (Winner: 2)
```

La seconda esecuzione che ho effettuato è un calendario con 14 squadre, un solo derby e distanza tra giornata di andata e giornata di ritorno di almeno 8 giornate. Impiega 23 secondi:

```
Models       : 1+
Calls        : 1
Time         : 23.253s (Solving: 21.90s 1st Model: 21.66s Unsat: 0.00s)
CPU Time     : 169.766s
Threads      : 8        (Winner: 6)
```

La terza esecuzione che ho effettuato è un calendario con 16 squadre, un solo derby e distanza tra giornata di andata e giornata di ritorno di almeno 10 giornate. Impiega 21 minuti:

```
Models       : 1+
Calls        : 1
Time         : 1294.863s (Solving: 1292.02s 1st Model: 1291.91s Unsat: 0.00s)
CPU Time     : 9933.938s
Threads      : 8        (Winner: 3)
```

La quarta esecuzione che ho effettuato è un calendario con 18 squadre, un solo derby e distanza tra giornata di andata e giornata di ritorno di almeno 10 giornate impiega più di 30 minuti e l'ho terminata.

Le successive esecuzioni che ho effettuato con più derby e 14 squadre in su sono state terminate perché impiegano più di 30 minuti.

2.2 Osservazioni finali

A seguito dei risultati ottenuti posso affermare che:

- quando il numero di derby è maggiore, l'algoritmo impiega più tempo per trovare una soluzione;
- quando la distanza tra una giornata di andata e una giornata di ritorno con le stesse squadre aumenta, l'algoritmo impiega più tempo per trovare una soluzione;
- ha tempistiche ottime con un numero minore di squadre;
- quando il numero delle squadre aumenta, l'algoritmo impiega più tempo per trovare una soluzione.

2.3 Come eseguire

Una volta ottenuto il codice dal repository di GitHub https://github.com/lorenzobotto/progettoAsp-Clingo, si può eseguire con 20 squadre, 4 derby e 8 threads tramite il codice:

```
clingo calendar.cl -t 8
```

Modificare a seconda delle necessità i vari parametri per eseguire con meno squadre o con vincoli diversi.