Relazione esercizi PROLOG e CLINGO

Maurizio Dominici, Alessandro Serra, Andrea Aloi

10 maggio 2014

Indice

Parte I

PROLOG

Capitolo 1

Domini

In questa sezione verranno illustrati i domini presi in esame per gli esercizi. Vediamo subito quali sono:

1.1 Dominio dei Cammini (10x10)

Il dominio dei cammini consiste in un inseme di posizioni (100, disposte con coordinate di un quadrato 10x10), tra cui una iniziale e una finale, che possono essere libere oppure occupate. Alla luce di ciò si deve cercare una sequenza di azioni che portino dalla posizione iniziale a quella finale, passando da posizioni definite (entro i confini della mappa insomma) ed evitando le posizioni occupate, precedentemente definite nel dominio.

1.2 Dominio dei Cammini (20x20)

Il dominio dei cammini consiste in un inseme di posizioni (400, disposte con coordinate di un quadrato 20x20), tra cui una iniziale e una finale, che possono essere libere oppure occupate. Alla luce di ciò si deve cercare una sequenza di azioni che portino dalla posizione iniziale a quella finale, passando da posizioni definite (entro i confini della mappa insomma) ed evitando le posizioni occupate, precedentemente definite nel dominio.

1.3 Dominio del mondo dei blocchi

La descrizione del dominio del mondo dei blocchi è nota in letteratura. Consiste nell'avere a disposizione dei blocchi che possono essere inpilati uno sopra l'altro. Ogni blocco può essere sopra a un altro blocco oppure sul tavolo.

Sul tavolo ci possono essere infiniti blocchi, mentre su un blocco può esserci, immediatamente, un solo blocco. Viene inoltre specificato, tramite il fatto `clear`, quando sopra a un blocco non ce n'è un altro. Vi sono, inoltre, due configurazioni proposte per questo dominio. La prima è meno complessa in quanto presenta pochi blocchi. La seconda configurazione è più intricata, con più blocchi rispetto alla prima e più mosse necessarie per arrivare al goal.

1.4 Dominio della Metropolitana di Londra

Come si evince dal nome di questo dominio, in questa sede viene illustrata parte della metropolitana di Londra. Vi è una suddivisione in Linee, caratterizzate da una lista di fermate (stazioni della metropolitana) e da due possibili direzioni. Ogni fermata è caratterizzata da un nome e da due coordinate indicanti la dislocazione rispetto alla stazione di London Bridge, avente infatti coordinate `(0, 0)`. Le azioni possibili sono quella di salire (`Sali(Linea, Direzione)`, possibile quando ci si trova in una stazione della Linea su cui si desidera salire. È poi possibile scendere (`Scendi(Stazione)`, quando si è su un treno di una linea in cui è presente la Stazione indicata. Infine vi è l'azione di Vai (`Vai(Linea, Direzione, StazionePartenza, StazioneArrivo`), che indica, una volta che si è su un treno, lo spostamento da una stazione all'altra (consecutive) su una stessa linea.

Capitolo 2

Ricerche in profondità

In questa parte verranno illustrati gli esercizi inerenti la ricerca in profondità sui vari domini.

2.1 Dominio dei Cammini (10x10)

2.1.1 Ricerca in profondità semplice

La prima ricerca che prendiamo in esame è la così detta ricerca semplice, ovvero una ricerca senza controlli di cicli. Come ci si aspetterebbe, a causa dell'assenza di controlli sugli stati già esplorati, una volta lanciato l'interprete prolog con la ricerca semplice viene restituito `Out of local stack`. Questo è dovuto, in effetti, a una ricursione infinita in quanto i cammini possibili prevedono cicli. L'interprete espande sempre la chiamata ricursiva più "profonda" e, la presenza di cicli nei cammini, non garantisce la terminazione.

2.1.2 Ricerca in profondità con controllo di cicli

Per ovviare a questo problema si è implementata la ricerca in profondità con controllo di cicli. L'idea è quella di utilizzare un insieme indicante le posizioni precedentemente visitate, in maniera da "potare" i rami decisionali generanti cicli. L'insieme è chiamato `visitati` e, a ogni passo di ricorsione, gli viene aggiunta la posizione attuale.

2.1.3 Ricerca in profondità limitata

La ricerca in profondità limitata consiste nel definire un limite di profondità entro il quale la ricerca si deve fermare. Se è possibile dimostrare l'esistenza di un percorso lungo al più quanto il limite fornito viene restituita la sequenza

risultato. Viceversa vi sarà un fallimento. L'implementazione usata per questa strategia consiste nel passaggio del limite a ogni chiamata ricorsiva. Esso viene decrementato di uno al passaggio alla successiva chiamata ricorsiva e, ogni volta, viene controllato che questo limite sia maggiore di zero. Risulta chiaro che una volta arrivati alla base della ricorsione si è a un livello di profondità pari a quello inizialmente definito. Come detto: se il goal viene raggiunto entro il livello stabilito si ha la soluzione, altrimenti, arrivati alla base della ricorsione, si ha un fallimento.

2.1.4 Ricerca in profondità ad approfondimento iterativo

Per questa versione viene riutilizzata la precedente ricerca (2.1.3) impostando il limite inizialmente a 1 (quindi si avrà successo solo nel caso in cui posizione iniziale e finale coincidano). A ogni iterazione viene incrementato il limite di un'unità e nuovamente esplorato il cammino con la la ricerca in profondità limitata. Il principale vantaggio di questo approccio è la sicurezza di trovare una soluzione minima, in quanto i passi hanno tutti ugual costo (non ci sono cammini pesati) e in quanto il procedimento può essere visto (seppure il comportamento sia diverso) in maniera simile a una ricerca in ampiezza. Questo non è del tutto vero: infatti ogni volta si esplorerà in profondità il ramo in questione e, solo successivamente, si esploreranno, sempre in profondità, i nodi di un altro ramo. Tuttavia, considerando il fatto che si esplorerà sempre prima un cammino più corto di uno lungo, la prima soluzione trovata, come per la ricerca in ampiezza, avrà la proprietà desiderabile di essere anche la soluzione minima. In effetti questa strategia combina i vantaggi della ricerca in profondità con quelli della ricerca in ampiezza, con l'importante caratteristica di ridurre la complessità spaziale (in termini di utilizzo della memoria stack).

2.2 Dominio dei Cammini (20x20)

2.2.1 Ricerca in profondità semplice

Anche in questo caso, come per il dominio illustrato nella sezione 1.1, verrà restituito l'errore `Out of local stack`. I motivi sono i medesimi illustrati nella sezione 2.1.1.

2.2.2 Ricerca in profondità con controllo di cicli

Come fatto precedentemente, nella sezione 2.1.2, per risolvere il problema 2.2.1, si è utilizzato un controllo di cicli. Non si ripete il procedimento in questa sede in quanto sufficientemente illustrato nella sezione 2.1.2.

2.2.3 Ricerca in profondità limitata

Sebbene il funzionamento sia del tutto simile a quanto illustrato nella sezione 2.1.3, la maggiore profondità del dominio implica un numero sensibilmente maggiore delle possibilità di esplorazione in profondità. Questo si ripercuote in maniera piuttosto evidente sui tempi di esecuzione, di gran lunga superiori a quelli del dominio di dimensioni più piccole. È un fatto sicuramente atteso, dovuto all'enorme differenza di numero di possibili percorsi di una data lunghezza rispetto al dominio 1.1.

2.2.4 Ricerca in profondità ad approfondimento iterativo

Le maggiori conseguenze enfatizzate nella sezione 2.2.3 si riscontrano con questa strategia. In effetti, assumendo che la soluzione minima sia lunghezza `l`, sarà necessario esplorare prima tutti i possibili cammini di tutte le lunghezza che vanno da `1` a `l-1` e, constatando quanto ci si possa impiegare a trovare tutti i cammini di una data lunghezza a ogni livello, il tempo di attesa è molto lungo. Le stampe inserite a ogni nuova chiamata con lunghezza maggiore sono indizio di ciò (più il numero della profondità massima aumenta e più, ovviamente, queste stampe si fanno rarefatte).

2.3 Dominio del mondo dei blocchi - prima configurazione

Su questo dominio non ci sono significative differenze rispetto al Dominio dei cammini, conseguentemente non spenderemo ulteriori commenti

2.4 Dominio del mondo dei blocchi - seconda configurazione

Più interessante è la seconda configurazione che, essendo molto più complessa della prima, causa un incremento importante dei tempi necessari a trovare

una soluzione. Tralasciando la Ricerca in profondità semplice, che anche qui, a causa di cicli, porta a un errore `Out of local stack`.

2.4.1 Ricerca in profondità con controllo di cicli

La ricerca in profondità con controllo di cicli risulta, a causa della natura del dominio, molto lunga. Il risultato trovato è ben distante da una soluzione ottimale, questo è dovuto al fatto che, in effetti, l'algoritmo si occupa solamente di evitare i possibili cicli, tuttavia, l'insieme delle mosse possibili, molto vasto: conseguentemente la soluzione include diverse mosse non ottimali che inficiano sulle prestazioni temporali.

2.4.2 Ricerca in profondità ad approfondimento iterativo

Anche qui vengono spese ulteriori parole sulla Ricerca in profondità limitata e si passa subito alla Ricerca in profondità ad approfondimento iterativo. I problemi temporali incontrati in 2.4.1 si ripercuotono, ovviamente anche qui; tuttavia questo dominio è particolarmente interessante in quanto permette di evidenziare una proprietà fondamentale dell'iterative deepening: nonostante il tempo impiegato sia paragonabile alla soluzione proposta in 2.4.1, con l'approfondimento iterativo viene trovata una soluzione ottima, per il discorso già affrontato nella sezione 2.1.4.

2.5 Dominio della metropolitana di Londra

Questo dominio, affrontato con le tecniche di ricerca in profondità già ampiamente illustrate, non offre particolari nuovi spunti di discussione rispetto ai domini precedenti. Questo, fondamentalmente, è dovuto al fatto che la caratteristica principale che differenzia questo dominio dai precedenti è la presenza di peso sugli archi del grafo (laddove, precedentemente, ogni mossa aveva una valore unitario). La mera ricerca in profondità, non tenendo conto di questo, "appiattisce" questo dominio rendendolo del tutto simile ai precedenti. Sarà più interessante l'analisi all'interno del capitolo 4, sulle Ricerche Informate.

Capitolo 3

Ricerche in ampiezza

In questa parte verranno illustrati gli esercizi inerenti la ricerca in ampiezza sui vari domini.

3.1 Dominio dei Cammini (10x10)

3.1.1 Ricerca in ampiezza semplice

Con la ricerca in ampiezza, teoricamente, si dovrebbe trovare una soluzione anche in presenza di cicli infiniti, proprietà non garantita, invece, con la ricerca in profondità. Tuttavia, senza escludere gli stati precedentemente visitati, lo spazio a disposizione dello stack viene esaurito rapidamente, causando un errore di `Out of global stack`. Questo, come detto, è comprensibile in quanto la ricerca in ampiezza espande ogni cammino possibile, compresi i cicli. Prima del raggiungimento del goal è possibile, quindi, che l'espansione continua di cammini che andrebbero idealmente "potati" (ovvero i cicli), occupi un tale spazio in memoria da compromettere il funzionamento desiderato dell'algoritmo.

3.1.2 Ricerca in ampiezza con stati visitati

Questo raffinamento risolve il problema esposto nella sezione precedente (3.1.1), consentendo il raggiungimento della soluzione in un tempo più che accettabile.

3.2 Dominio dei Cammini (20x20)

In questo dominio, a differenza di quanto accadeva per la ricerca in profondità, non si vedono sostanziali differenze di tempo con il dominio precedente (3.1) per quanto riguarda la Ricerca in ampiezza con stati visitati[1]. Questa è una caratteristica sicuramente positiva, che rende la ricerca in ampiezza preferibile sulla ricerca in profondità per questa tipologia di problemi; oltretutto, la prima soluzione trovata, ha anche la proprietà di essere ottimale.

3.3 Dominio del mondo dei blocchi - prima configurazione

Questa prima configurazione del dominio del mondo dei blocchi non presenta particolari problematiche, trovando le soluzioni in tempo accetabile.

3.4 Dominio del mondo dei blocchi - seconda configurazione

Questa seconda configurazione risulta, invece, molto più interessante della prima. La maggiore complessità porta a un accrescimento notevole delle configurazioni di stati e, conseguenza di ciò, è il raggiungimento dell'errore `Out of global stack`. La memoria viene quindi rapidamente esaurita, permettendo di apprezzare la caratteristica principale che si aveva, invece, nell'iterative deepening (come illustrato nella sezione 2.4.2) dove, seppur in tempo ampio, veniva trovata una soluzione.

3.5 Dominio della metropolitana di Londra

La ricerca in ampiezza si adatta molto bene a questo dominio che, non presentando una complessità ampia come il dominio 1.3, permette il raggiungimento del goal in un tempo particolarmente vantaggioso. Anche in questa sede ci sarebbero da fare le considerazioni fatte nella sezione 2.5 sull'appiattimento del dominio.

[1] continua a presentare l'errore `Out of global stack`, invece, la Ricerca in ampiezza semplice

Capitolo 4

Strategia di ricerca informate

In questo capitolo utilizzeremo strategie di ricerca informate quali A* (4.2) e ID A* (4.3). Per illustrare nel dettaglio gli esiti ottenuti e poter fare tutte le considerazioni del caso, partiremo con una carrellata delle euristiche utilizzate per i vari domini, al fine di una più chiara comprensione delle future considerazioni.

4.1 Euristiche utilizzate

4.1.1 Euristica dei cammini

Per il dominio dei cammini (sia 1.1 che 1.2), l'euristica utilizzata è la Distanza di Manhattan. La scelta è ricaduta su questa euristica perché è sembrata quella che meglio aderisse ai movimenti consentiti tra le varie posizioni (in effetti non è presente lo spostamente in diagonale) presenti nel dominio.

4.1.2 Euristica del mondo dei blocchi

Le considerazioni fatte per trovare l'euristica da utilizzare nel mondo dei blocchi sono state di natura più complessa rispetto a quelle effettuate per il dominio dei cammini. L'idea che ha guidato la creazione dell'euristica è stata considerare stato iniziale, attuale e finale come insiemi di condizioni. L'avvicinarsi dello stato attuale a quello finale è stato, quindi, il rendere l'insieme dello stato attuale mano a mano più simile a quello dello stato finale, finendo poi per farli coincidere una volta trovata la soluzione. L'euristica utilizzata è stata quindi la differenza insiemistica tra l'insieme degli elementi (ovvero le condizioni) dell'insieme dello stato attuale e l'insieme degli elementi dello

stato finale[1]. Il motivo di questa scelta è appunto cercare di trovare la mossa che più assottilia la differenza tra i due insiemi, fino al raggiungimento della parificazione degli stessi. Una caratteristica necessaria per il concetto di euristica è che la stessa non sovrastimi mai il numero di mosse. Questo è stato verificato in diversi esempi che vengono illustrati qui di seguito.

```
1   S1 = {clear(a), on(a,b), clear(d), clear(e), ontable(b), ontable(e), ontable(d)}
2   S2 = {ontable(e), on(d,e), on(b,d), on(a,b), clear(a)}
3   |S1 − S2| = 4
```

Come si può vedere la cardinalità della differenza (riga 3) tra i due insiemi è 4 e le mosse necessarie per il passaggio dallo stato rappresentato dell'insieme `S1` (riga 1) allo stato rappresentato dall'insieme `S2` (riga 2) sono invece 8[2]. Il numero di mosse non risulta sovrastimato. Si noti che potrebbe sembrare che, al fine di una stima più precisa dell'euristica, l'idea di moltiplicare per 2^3 il risultato possa essere vincente. Purtroppo questo non è sempre verificato come dimostra quest'altro esempio:

```
1   S1 = {ontable(a), ontable(b), ontable (d), ontable (e), clear(a), clear(b), clear(d), clear(e)}
2   S2 = {ontable(e), on(d,e), on(b,d), on(a,b), clear(a)}
3   |S1 − S2| = 6
```

In questo particolare esempio, in cui i blocchi sono tutti disposti sul tavolo senza altri blocchi sopra di essi, la cardinalità della differenza tra i due insiemi è 6, ma anche le mosse necessarie sono 6[4].

Di seguito altri esempi:

```
1   S1 = {ontable (a), on(e,a), on(d,e), on(b,d), clear(b)}
2   S2 = {ontable(e), on(d,e), on(b,d), on(a,b), clear(a)}
3   |S1 − S2| = 3
```

```
1   S1 = {ontable(e), ontable(a), (ontable(d), on(b,d), clear(e), clear(a), clear(b)}
2   S2 = {ontable(e), on(d,e), on(b,d), on(a,b), clear(a)}
3   |S1 − S2| = 4
```

4.1.3 Euristica della Metropolitana di Londra

L'euristica utilizzata per il dominio della metropolitana di Londra è la distanza euclidea. La scelta è ricaduta su questa euristica perché ci sembrava

[1] si noti che la differenza insiemistica non è commutativa, quindi l'ordine di minuendo (insieme degli stati attuali) e sottraendo (insieme degli stati finali) risulta determinante.

[2] Le mosse necessarie sono, nell'ordine: prendere A, appoggiare A, prendere D, appoggiare D sopra E, prendere B, appoggiare B sopra a D, prendere A, appoggiare A sopra a B.

[3] questo perché sembrerebbe che le mosse necessarie per portare un elemento in una posizione differente siano due: prendere l'elemento e riporlo

[4] nell'ordine: prendere D, appoggiare D su E, prendere B, appoggiare B su D, prendere A, appoggiare A su B.

molto aderente al concetto di distanza tra due fermate della metropolitana in quanto, si presume, che per quanto possibile le linee della metropolitana tendono a limitare le curve. Affinché fosse possibile utilizzare questa euristica con A* (e conseguentemente, quindi, con IDA*) è stata necessaria una modifica all'implementazione dell'algoritmo di A* rispetto a quella degli altri domini. La modifica è avvenuta, nella fattispecie, per la funzione g che ha bisogno, per essere calcolata, della stazione attuale e della prossima stazione in input al fine di calcolarne la distanza effettuata. A questo punto il calcolo della funzione g avviene aggiungendo la distanza euclidea alla g precedentemente calcolata (laddove, in presenza di domini senza pesi su ogni movimento, veniva semplicemente incrementata di 1[5]).

4.2 A-star

L'algoritmo A* è noto in letteratura, si rimanda ad essa per una spiegazione esaustiva dello stesso.

4.2.1 Cammini

In entrambi i domini dei cammini, la soluzione viene trovata in un tempo quasi istantaneo, enfatizzando la differenza e l'importanza dell'uso di ricerche informate su questa tipologia di problemi rispetto ai risultati ottenuti in 2.1, 2.2, 3.1 e 3.2.

4.2.2 Mondo dei Blocchi

Il mondo dei blocchi, in queste particolari configurazioni proposte, presenta qualche problematica in più. Sebbene nella prima configurazione non si incontrino particolari controindicazioni nell'uso di A*, per quanto riguarda la seconda configurazione vengono evidenziati tutti i limiti precedentemente riscontrati in 3.4. In effetti, questo particolare dominio piuttosto vasto, porta all'esplorazione di una quantità enorme di stati, causando un errore di `Out of global stack`.

4.2.3 Metropolitana di Londra

Il funzionamento di A* (opportunamente modificato come mostrato nella sezione 4.1.3) in questo contesto è piuttosto proficuo, portando al conseguimento del goal in tempo relativamente breve. Si noti che la soluzione (ottima

[5] l'incremento di 1 rappresenta l'incremento del conto di un passo in più

per natura stessa dell'algoritmo A*) è ovviamente differente dalle soluzioni trovate in 2.5 e in 3.5, in quanto il costo di un passo, in questo contesto, non è più costante.

4.3 Iterative Deepening A-star

L'algoritmo ID-A* è una variante di A* implementata, come suggerisce il nome, con un approfondimento iterativo. I cammini creati da A* vengono espansi per livelli di profondità, come spiegato precedentemente nella sezione 2.1.4, riducendo la memoria stack utilizzata (quindi riducendo la complessità spaziale). Esaminiamo ora, caso per caso, i risultati ottenuti.

4.3.1 Cammini

In questo dominio non vi sono sostanziali differenze con l'uso di A* e di IDA*. Teniamo buono, quindi, quanto detto nella sezione 4.2.1.

4.3.2 Mondo dei Blocchi

Nel mondo dei blocchi la differenza è invece più interessante. La prima configurazione non denota differenze nel raggiungimento della soluzione; è invece nella seconda configurazione che le differenze diventano apprezzabili. In effetti, laddove con A* si raggiungeva l'errore `Out of global stack`, la desiderabile proprietà dell'Iterative Deepening di ridurre la complessità spaziale, porta al raggiungimento del goal senza incappare in errori dovuti alla scarsità di memoria disponibile. L'arrivo alla soluzione avviene in tempistiche lunghe ma, come già ampiamente commentato, questo è dovuto alla particolare complessità di questa configurazione in questo dominio.

4.3.3 Metropolitana di Londra

Concludiamo con il dominio della metropolitana di Londra. Qui non vi sono apprezzabili differenze rispetto all'utilizzo di A*, che forniva già ottimi risultati. Vale ancora la pena di rimarcare che la soluzione ottima trovata differisce da quelle ottenute con ricerche in profondità e in ampiezza. Questo è ovviamente dovuto all'"appiattimento" del dominio in quei contesti (come spiegato nella sezione 2.5).

Parte II

CLINGO

Capitolo 5

Enunciato del problema delle Cinque Case

In questa parte verrà descritto lo svolgimento dell'esercizio delle Cinque Case. Il problema viene così enunciato:

> Cinque persone di nazionalità diverse vivono in cinque case allineate lungo una strada, esercitano cinque professioni distinte, e ciascuna persona ha un animale favorito e una bevanda favorita, tutti diversi fra loro. Le cinque case sono dipinte con colori diversi. Sono noti i seguenti fatti:
>
> 1. L'inglese vive nella casa rossa.
> 2. Lo spagnolo possiede un cane.
> 3. Il giapponese è un pittore.
> 4. L'italiano beve tè.
> 5. Il norvegese vive nella prima casa a sinistra.
> 6. Il proprietario della casa verde beve caffè.
> 7. La casa verde è immediatamente sulla destra di quella bianca.
> 8. Lo scultore alleva lumache.
> 9. Il diplomatico vive nella casa gialla.
> 10. Nella casa di mezzo si beve latte.
> 11. La casa del norvegese è adiacente a quella blu.
> 12. Il violinista beve succo di frutta.
> 13. La volpe è nella casa adiacente a quella del dottore.
> 14. Il cavallo è nella casa adiacente a quella del diplomatico.
>
> Trovare chi possiede una zebra.

Capitolo 6

Formulazione della soluzione

Il file che formalizza e risolve il problema si chiama `houses.cl` ed è suddiviso in 6 sezioni.

6.1 Dominio

In questa sezione vengono elencate le entità che figurano nel problema e i valori che queste assumono. L'entità `casa` è stata semplicemente codificata con un numero da 1 a 5 (usando il costrutto *Interval* di `clingo`) poiché le nazionalità dei suoi inquilini, le loro professioni e i loro animali e bevande preferiti sono stati implementati come predicati binari delle istanze di `casa`. Seguono quindi le entità `colore`, `nazionalita`, `animale`, `professione` e `bevanda`. L'uso del carattere `;` è una sintassi abbreviata per non dover enunciare più volte lo stesso *atomo* con valori diversi. Ad es: `a(x;y;z)` equivale a: `a(x). a(y). a(z).`

```
1  %% Dominio
2
3  % Casa
4  casa(1..5).
5
6  % Colore
7  colore(rossa;verde;bianca;gialla;blu).
8
9  % Nazionalita'
10 nazionalita(inglese;spagnolo;giapponese;italiano;norvegese).
11
12 % Animale
13 animale(cane;lumache;volpe;cavallo;zebra).
14
15 % Professione
16 professione(pittore;scultore;diplomatico;violinista;dottore).
17
18 % Bevanda
19 bevanda(te;caffe;latte;succo_di_frutta;altro).
```

6.2 Vincoli impliciti

In questa sezione vengono espressi quelli che sono vincoli "naturali", non espressamente descritti nell'enunciato del problema, ma che è più che ragionevole supporre. Vengono inoltre introdotti i predicati binari `coloreDi`, `nazionalitaDi`, `animaleDi`, `prefessioneDi` e `bevandaDi` precedentemenți accennati. Ciascuno associa alla casa (passata come primo argomento) una proprietà (secondo argomento) avente un valore appartenente al relativo dominio.

```
1  %% Vincoli impliciti
2
3  % Ogni casa ha uno ed un solo colore...
4  1 {coloreDi(X,C) : colore(C)} 1 :- casa(X).
5  % ...e due case con lo stesso colore sono la stessa casa
6  :- casa(X), casa(Y), coloreDi(X,C), coloreDi(Y,C), X != Y.
7
8  % In ogni casa abita una persona di una ed una sola nazionalita'...
9  1 {nazionalitaDi(X,N) : nazionalita(N)} 1 :- casa(X).
10 % ...e due case con la persona della stessa nazionalita' sono la stessa casa
11 :- casa(X), casa(Y), nazionalitaDi(X,N), nazionalitaDi(Y,N), X != Y.
12
13 % In ogni casa abita una persona che ama uno ed un solo animale...
14 1 {animaleDi(X,A) : animale(A)} 1 :- casa(X).
15 % ...e due case con la persona che ama lo stesso animale sono la stessa casa
16 :- casa(X), casa(Y), animaleDi(X,A), animaleDi(Y,A), X != Y.
17
18 % In ogni casa abita una persona che svolge una ed una sola professione...
19 1 {professioneDi(X,P) : professione(P)} 1 :- casa(X).
20 % ...e due case con la persona che svolge la stessa preofessione sono la stessa casa
21 :- casa(X), casa(Y), professioneDi(X,P), professioneDi(Y,P), X != Y.
22
23 % In ogni casa abita una persona che predilige una ed una sola bevanda...
24 1 {bevandaDi(X,B) : bevanda(B)} 1 :- casa(X).
25 % ...e due case con la persona che predilige la stessa bevanda sono la stessa casa
26 :- casa(X), casa(Y), bevandaDi(X,B), bevandaDi(Y,B), X != Y.
```

6.3 Convenzioni e funzione di adiacenza

Queste due sezioni trattano il posizionamento, assoluto e reciproco, delle case; hanno gli scopi rispettivamente di:

1. esprimere - soltanto attraverso commenti - quello che consideriamo essere ancora una volta una convenzione "a buon senso" circa la disposizione delle case lungo la via
2. definire una funzione di utilità, `adj`, che, dati due interi rappresentanti le case, restituisca un valore di verità se queste sono adiacenti, ossia se il valore assoluto della loro differenza è esattamente uguale a 1: questo predicato tornerà utile nella sezione successiva

```
1  %% Convenzioni
2
3  % casa(1) e' quella posta piu' a sinistra
4  % casa(5) e' quella posta piu' a destra
5  % casa(X) e' a destra di casa(Y) <=> X > Y
6  % casa(X) e' a sinistra di casa(Y) <=> X < Y
7
8  %% Funzione di adiacenza
9
10 adj(X,Y) :- casa(X), casa(Y), |X-Y|==1.
```

6.4 Vincoli espliciti

Questa sezione costituisce il cuore del problema, poiché codifica i 14 vincoli espressi nell'enunciato che restringono man mano i possibili risultati ($5!^5 = 3125$, inizialmente) fino a giungere alla soluzione desiderata. Viene riportato il codice e i relativi commenti. Ogni vincolo è espresso nella forma

$$:\text{-}\ A_1, ..., A_n, B$$

dove le regole $A_1, ..., A_n$ (solitamente $n = 2$) descrivono una o due proprietà di una stessa casa, mentre nella condizione B si *nega* quello che è il vincolo, poiché tutta la riga del vincolo è da considerarsi negata. Ad esempio, il primo vincolo afferma che *non può* esserci una casa in cui abita la persona di nazionalità inglese e che abbia il colore C, e che questo colore C sia diverso dal valore "rosso". Nei vincoli 11, 13 e 14 compare la funzione di adiacenza descritta in precedenza, anch'essa negata.

```
%% Vincoli espliciti

% 1. L'inglese vive nella casa rossa
:- nazionalitaDi(X,inglese), coloreDi(X,C), C!=rossa.

% 2. Lo spagnolo possiede un cane
:- nazionalitaDi(X,spagnolo), animaleDi(X,A), A!=cane.

% 3. Il giapponese e' un pittore
:- nazionalitaDi(X,giapponese), professioneDi(X,P), P!=pittore.

% 4. L'italiano beve te'
:- nazionalitaDi(X,italiano), bevandaDi(X,B), B!=te.

% 5. Il norvegese vive nella prima casa a sinistra
:- nazionalitaDi(X,norvegese), X!=1.

% 6. Il proprietario della casa verde beve caffe'
:- bevandaDi(X,caffe), coloreDi(X,C), C!=verde.

% 7. La casa verde e' immediatamente sulla destra di quella bianca
:- coloreDi(X,verde), coloreDi(Y,bianca), X!=Y+1.

% 8. Lo scultore alleva lumache
:- professioneDi(X,scultore), animaleDi(X,A), A!=lumache.

% 9. Il diplomatico vive nella casa gialla
:- professioneDi(X,diplomatico), coloreDi(X,C), C!=gialla.

% 10. Nella casa di mezzo si beve latte
:- bevandaDi(3,B), B!=latte.

% 11. La casa del norvegese e' adiacente a quella blu
:- nazionalitaDi(X,norvegese), coloreDi(Y,blu), not adj(X,Y).

% 12. Il violinista beve succo di frutta
:- professioneDi(X,violinista), bevandaDi(X,B), B!=succo_di_frutta.

% 13. La volpe e' nella casa adiacente a quella del dottore
:- animaleDi(X,volpe), professioneDi(Y,dottore), not adj(X,Y).

% 14. Il cavallo e' nella casa adiacente a quella del diplomatico
:- animaleDi(X,cavallo), professioneDi(Y,diplomatico), not adj(X,Y).
```

6.5 Output

Infine, mediante i predicati `hide#` e `show#`, nascondiamo inizialmente tutti i predicati per poi selettivamente mostrare `coloreDi`, `nazionalitaDi`, `animaleDi`, `prefessioneDi` e `bevandaDi`, tutti con *arità* 2. Senza invocare `hide#`, avremmo ottenuto anche i predicati del dominio e la funzione di adiacenza.

```
%% Output

#hide.
#show coloreDi/2.
#show nazionalitaDi/2.
#show animaleDi/2.
```

```
7  #show professioneDi/2.
8  #show bevandaDi/2.
```

L'output prodotto dall'esecuzione del nostro codice, invocando da terminale il comando `clingo 0 houses.cl`, è il seguente:

```
Answer: 1
coloreDi(5,verde) coloreDi(4,bianca) coloreDi(3,rossa) coloreDi(2,blu)
coloreDi(1,gialla) nazionalitaDi(5,giapponese) nazionalitaDi(4,spagnolo)
nazionalitaDi(3,inglese) nazionalitaDi(2,italiano) nazionalitaDi(1,norvegese)
animaleDi(5,zebra) animaleDi(4,cane) animaleDi(3,lumache) animaleDi(2,cavallo)
animaleDi(1,volpe) professioneDi(5,pittore) professioneDi(4,violinista)
professioneDi(3,scultore) professioneDi(2,dottore) professioneDi(1,diplomatico)
bevandaDi(5,caffe) bevandaDi(4,succo_di_frutta) bevandaDi(3,latte)
bevandaDi(2,te) bevandaDi(1,altro)
SATISFIABLE

Models       : 1
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

che ci consente di formulare il risultato finale:

Casa	**Colore**	**Nazionalità**	**Animale**	**Professione**	**Bevanda**
1	gialla	norvegese	volpe	diplomatico	altro
2	blu	italiano	cavallo	dottore	te
3	rossa	inglese	lumache	scultore	latte
4	bianca	spagnolo	cane	violinista	succo di frutta
5	verde	giapponese	zebra	pittore	caffe

E scoprire così che il proprietario della zebra è il pittore giapponese che vive nella casa verde (l'ultima) e beve caffé.