Rush Hour ASP solver

Fiorenzo Tittaferrante

15 gennaio 2023

1 Introduzione

Rush Hour è un rompicapo logico che consiste nel spostare le varie auto per poter trovare una soluzione: come quando si è in macchina e bloccati nel traffico, l'obiettivo è uscire dall'ingorgo spostando i veicoli che impediscono all'automobile rossa di passare.

Sul piano di gioco si posiziona l'auto rossa da far uscire dall'ingorgo e le auto del traffico. Le auto possono muoversi solamente verso l'alto o verso il basso, oppure a destra o a sinistra, in base alla direzione che ciascuna di esse possiede. Inoltre le auto hanno una dimensione di 2 oppure di 3 unità; in particolare l'auto rossa è lunga 2 unità.

1.1 Convenzioni e scelte progettuali

Il rompicapo è organizzato in una griglia 6×6 in cui ciascun punto è rappresentato da una coppia (X, Y), dove la X rappresenta la riga e la Y la colonna.

Stiamo affrontando un problema di pianificazione e il tempo svolge un ruolo fondamentale: ogni regola è descritta in primo luogo dal tempo T in cui quell'azione accade.

I nomi date alle auto sono: r (*red*) per l'auto rossa e $b1$, $b2$, $b3$ (e così via) per tutte le altre auto (di colore blu). Inoltre, per ogni auto è indicato l'orientamento di quest'ultima tramite `h` (*horizontal*) e `v` (*vertical*).

La scelta progettuale più importante riguarda come strutturare un auto. Tra le varie possibilità analizzate vi sono:

1. Strutturare l'auto con la sua dimensione e l'orientamento, per poi specificare successivamente la posizione.
2. Strutturare l'auto con la coordinata che rimane fissa (X per le auto orizzontali e Y per le auto verticali), seguita dalle altre coordinate che indica in quale riga o colonna si trovano i blocchi che compongono l'auto;
3. Strutturare l'auto come un insieme di blocchi creati separatamente: complesso da gestire.

1.1.1 Versione 1

La prima soluzione rende le auto indipendenti dalla propria lunghezza; è necessario semplicemente indicare una sola posizione, indicata dalla coppia (X, Y) più piccola.

1.1.2 Versione 2

La seconda soluzione tiene traccia della posizione di ogni auto come se fosse un singolo blocco. Questo approccio è dipendente dalle dimensioni delle auto coinvolte nel puzzle, ovvero, per auto di dimensioni diverse è necessario creare dei predicati ad hoc per le posizioni e il loro aggiornamento, le azioni e la loro esecuzione.

2 Implementazione versione 1

2.1 Premesse

Il tempo in cui eseguire le azioni è fornito da linea di comando, invece la dimensione della griglia è fissa. Il fatto `moveTo` non ha un effettivo utilizzo, ma è usato per chiarezza del codice. Segue `orientation` che indica, come detto inizialmente, le ***orientazioni*** possibili: `h` per *horizontal* e `v` per *vertical*. Il fatto `cars` rappresenta per semplicità il nome (o colore) di ciascuna auto.

Iniziamo nel definire la parte estensionale del programma con i seguenti fatti.

```
time(0..1).
grid(1..6,1..6).
moveTo(right; left; up; down).
orientation(h;v).
cars(r;b).
```

L'auto è caratterizzata dal nome, dalla dimensione e dalla direzione.

```
car(r, 2, h).
car(b, 3, v).
```

Definito l'oggetto auto, si procede identificando la ***posizione*** occupata indicandola tramite il valore dell'ascissa e dell'ordinata più piccola. Il punto nella griglia che si indica, quindi, è il punto iniziale dell'auto: per un auto orizzontale è il punto più a sinistra, per un auto verticale è il punto più in alto.

```
position(0, r, 3, 1).
position(0, b, 2, 4).
```

Ciò che ci si aspetta quindi è di trovare, al tempo 0, il primo blocco dell'auto r in $(3, 1)$ (riga 3, colonna 1) ed essendo r orizzontale e lunga 2, il secondo blocco è in posizione $(3, 2)$.
Lo stesso ragionamento è valido per b e l'immagine sottostante rappresenta la situazione descritta dai fatti appena visti.

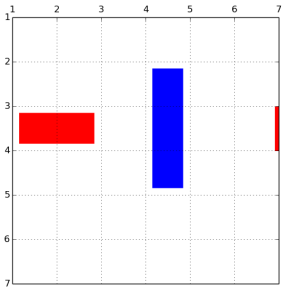


Figura 1: Situazione iniziale al tempo 0

2.2 Occupazione

È proprio grazie alla dimensione e all'orientamento dell'auto che è possibile capire tutte le posizioni nella griglia che l'auto occupa.

La regola `busy(T, C, X, Y..Y+N-1)` calcola le ***posizioni occupate*** da una determinata auto C al tempo T ed ha arità 4 con, rispettivamente:

- Il tempo;
- Il nome dell'auto;
- La coordinata X per le auto orizzontali;
- Un range di Y, per le auto orizzontali, per le posizioni di ciascun blocchetto che compongono l'auto lunga N.

Di regole `busy` ve ne sono 2 per distinguere i casi di auto orizzontale e verticale e il corpo contiene `car(C, N, h)`, per ottenere che C sia orizzontale e `position(T, C, X, Y)` per avere la posizione occupata da C al tempo attuale T:

```
busy(T, C, X, Y..Y+N-1) :- time(T), T<l,
                           position(T, C, X, Y), car(C, N, h).

busy(T, C, X..X+N-1, Y) :- time(T), T<l,
                           position(T, C, X, Y), car(C, N, v).
```

Queste due regole graficamente generano la situazione raffigurata nell'immagine sottostante.

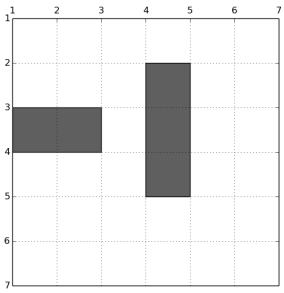


Figura 2: Posizioni occupate al tempo 0

2.3 Posizioni libere

A questo punto si definiscono `not_free(T,X,Y)` e `free(T,X,Y)` che indicano rispettivamente le posizioni della griglia che sono ***non libere*** e quelle che sono ***libere***; in questo modo viene gestita la griglia e le singole posizioni, tralasciando le auto.

La regola `not_free` serve per essere negata nella `free` per ottenere la lista di tutte le posizioni libere:

```
not_free(T,X,Y) :- time(T), T<l,
                   grid(X,Y), car(C,_,_), busy(T,C,X,Y).

free(T, X, Y) :- time(T), T<l,
                 grid(X,Y), car(C,_,_),
                 not not_free(T,X,Y).
```

Graficamente si ottiene una griglia dove in verde sono rappresentate tutte le posizioni libere.

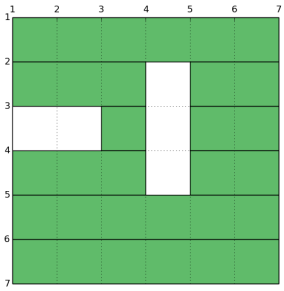


Figura 3: Posizioni libere al tempo 0

2.4 Mosse eseguibili

Le ***mosse eseguibili*** sono descritte dalle regole `executable`: la testa della regola contiene, oltre che il tempo e il nome dell'auto, la mossa, le coordinate X e le coordinate Y, in base al tipo di movimento che l'auto può fare.

Un auto orizzontale per spostarsi a destra deve lavorare solamente sulle coordinate Y dato che la riga rimane sempre la stessa. La posizione d'interesse si trova a $Y + N$ ed è quella che deve essere libera.

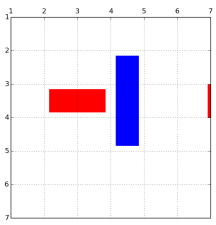
Per eseguire questa mossa è necessario che al tempo T l'auto si trovi in (X, Y). Un ragionamento equivalente allo spostamento verso destra vale per quello verso il basso di un auto verticale.

```
executable(T, C, right, X, Y+1) :- time(T), grid(X,Y+N),
                                   car(C, N, h), free(T, X, Y+N),
                                   position(T, C, X, Y).
```

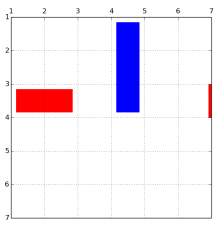
Lo spostamento verso sinistra e verso l'alto invece, sono leggermente differenti. In particolare la differenza sostanziale riguarda la `free`, poichè la posizione che deve essere libera è proprio quella immediatamente a sinistra (o in alto) con ordinata $Y - 1$ (o $X - 1$).

```
executable(T, C, left,  X, Y-1) :- time(T), grid(X,Y-1),
                                   car(C, N, h), free(T, X, Y-1),
                                   position(T, C, X, Y).
```

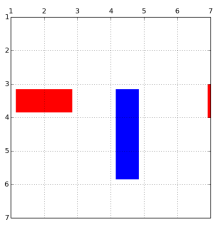
Nel nostro caso di esempio, il predicato `executable` genera le mosse mostrate in figura 4.



(a) Mossa a destra di r



(b) Mossa in alto di b



(c) Mossa in basso di b

Figura 4: Mosse eseguibili al tempo 0

2.5 Scelta della mossa

A questo punto si prosegue scegliendo una ed una sola mossa eseguibile, a patto che il tempo sia ancora valido:

```
1{move(T, C, MOV) : car(C,N,_), executable(T,C,MOV,X,Y)}1 :- time(T), T<l.
```

2.6 Aggiornamento posizioni

L'ultimo punto da sviluppare è l'***aggiornamento delle posizioni*** al nuovo tempo. È necessario fare una distinzione tra le posizioni delle auto statiche, ovvero quelle che non effettuano nessuna mossa, dalla posizione dell'unica auto che si sposta.

Supponendo di aver effettuato una mossa al tempo 0 come quella raffigurata, si devono aggiornare le posizioni al tempo 1 di tutte le auto.

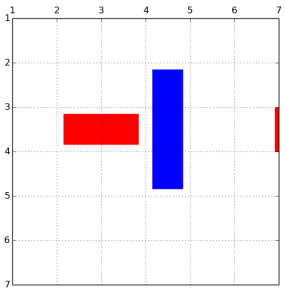


Figura 5: Mossa eseguita al tempo 0 e nuove posizioni al tempo 1

Eseguendo lo spostamento a destra, la nuova posizione (X, Y) al tempo T è dato se, nel tempo precedente, l'auto era di una posizione a sinistra e la mossa che è stata effettuata al tempo precedente era lo spostamento di quell'auto verso destra.

I casi *left*, *up* e *down* sono equivalenti.

```
position(T,C,X,Y) :- moveTo(right), time(T), time(T-1), T<=l,
                     grid(X,Y-1), position(T-1,C,X,Y-1), car(C,_,h),
```

Per le auto che non hanno effettuato nessun movimento è indispensabile mantenere nel nuovo tempo le stesse posizioni che occupavano nel tempo precedente. È stato sufficiente negare il predicato `move` nel tempo precedente per avere la certezza che l'auto non si è mossa dalla posizione che occupava.

```
position(T,C,X,Y) :- time(T), time(T-1), T<l,
                     position(T-1,C,X,Y), car(C,_,_),
                     not move(T-1, C, _).
```

2.7 Goal

Definite le regole per generare le posizioni libere o occupate, le mosse possibili e la mossa da eseguire, si procede definendo il *goal*: l'auto rossa deve giungere alla fine della sua riga ed essendo lunga 2, la posizione finale deve essere $(3, 5)$. Infatti:

```
goal(T) :- time(T), T<l, position(T,r,3,5).
:- not goal(_).
```

3 Ottimizzazioni

3.1 Ottimizzazioni sul codice

Prendendo in esempio la configurazione 2 (file *configuration/sample02.lp*) ed analizzando le varie statistiche del grounder tramite il comando:

```
gringo rush_hour.lp configuration/sample02.lp -c l=40 | clasp --stat
```

Si ottengono le seguenti informazioni (le più rilevanti):

```
Time         : 7.954s (Solving: 7.79s 1st Model: 4.76s Unsat: 3.02s)
CPU Time     : 7.815s

Choices      : 237130
Conflicts    : 71504    (Analyzed: 71503)
Restarts     : 227      (Average: 314.99 Last: 237)
```

Abbiamo un numero elevato di scelte, di conflitti e di restart. Per limitare e cercare di diminuire il più possibile questi dati ci si è incentrati nel modificare alcune regole, soprattutto le `position`: la regola che ha un impatto maggiore è quella che ricalcola le posizioni delle auto che non effettuano nessuna mossa:

```
position(T,C,X,Y) :- time(T), time(T-1), T<l,
                     position(T-1,C,X,Y), car(C,_,_),
                     not move(T-1, C, _).
```

La variabile *dummy* `_` aumenta il numero di atomi e di regole. La decisione presa corrisponde a sostituire queste variabili anonime con quello che davvero ci si aspetta di avere. In basso è possibile vedere la regola modificata.

```
position(T,C,X,Y) :- time(T), time(T-1), T<l,
                     position(T-1,C,X,Y), car(C,N,O), dim(N), orientation(O),
                     not move(T-1, C, right), not move(T-1, C, left),
                     not move(T-1, C, up), not move(T-1, C, down).
```

In particolare a `car` sono state aggiunte la dimensione dell'auto (inserita all'inizio del programma con il fatto `dim(2;3).`) e il suo orientamento.
Anche la `not move` è stata espansa e scritta in modo esplicito con tutti i 4 movimenti che un auto (a prescindere che sia orizzontale o verticale) non può fare per poter essere considerata "ferma" ed è proprio questa modifica che ha portato a risultati migliori.

Sono state fatte ulteriori piccole modifiche nei corpi delle varie funzioni per essere più stringenti e per limitare ulteriormente le scelte e i conflitti che il programma ha.

Infatti, analizzando nuovamente il grounding notiamo un netto miglioramento delle performance:

```
Time         : 4.124s (Solving: 3.95s 1st Model: 2.91s Unsat: 1.04s)
CPU Time     : 4.051s

Choices      : 116361
Conflicts    : 48310    (Analyzed: 48309)
Restarts     : 171      (Average: 282.51 Last: 39)
```

3.2 Multishot

Dare un tempo massimo di esecuzione per una determinata configurazione non è molto vantaggioso in termini di tempo poichè il goal potrebbe essere soddisfatto in un tempo minore. Si vorrebbe partire dal tempo 0 e, una volta raggiunto il goal minimo, il programma deve terminare.

La situazione appena descritta rappresenta il cosiddetto *incremental solving*. Con il *multishot* è stato possibile effettuare la ricerca della soluzione di una configurazione in modo incrementale tramite un programma in python e modificando il codice sorgente del risolutore di Rush Hour.

Sono le direttive `#program check(l).` e `#program step(l).` che forniscono la variabile tempo al programma. Infatti, le teste delle regole sono cambiate sostituendo la variabile `T` con l'atomo `l`.

La nuova testa e il corpo di `position` diventa:

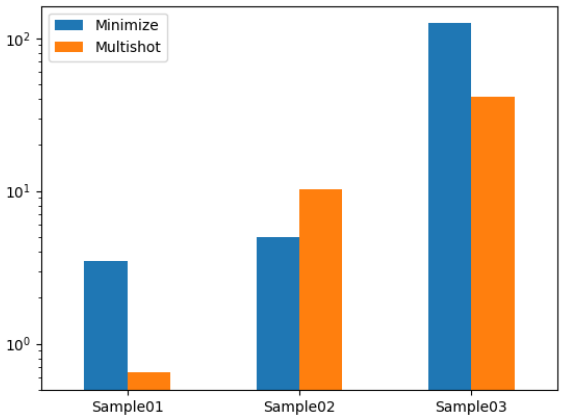
```
position(l,C,X,Y) :- position(l-1,C,X,Y), car(C,N,O), dim(N), orientation(O),
                     not move(l-1, C, right), not move(l-1, C, left),
                     not move(l-1, C, up), not move(l-1, C, down).
```

Confrontando i tempi di esecuzione tra una versione del programma che utilizza `#minimize` con una che utilizza il multishot si ha un notevole guadagno di tempo. Eseguiamo i programmi con i seguenti comandi:

```
clingo configuration/sample01.lp rush_hour_minimize.lp -c l=40

python3 inc.py configuration/sample01.lp rush_hour_multishot.lp
```

Non è noto a priori quante mosse sono necessarie per giungere al goal: alcune volte il multishot è più lento di una minimize poichè può capitare che il tempo scelto per il programma `rush_hour_minimize.lp` è proprio quello minimo e l'esecuzione è molto veloce. Questa situazione è capitata analizzando i tempi con la configurazione `sample02`.



4 Implementazione versione 2

In questa soluzione, per definire il puzzle, vengono usati i predicati:

- `time/1`: serve a tenere traccia del numero di mosse effettuate in ogni momento;
- `grid/2`: indica tutte le coordinate della griglia in cui le auto vengono poste;
- `no/2`: indica se una certa posizione della griglia è invalicabile;
- `orientation/1`: indica il verso in cui l'auto può essere posta (orizzontale e verticale);
- `position/6` e `position/7`: indicano la posizione di una certa auto in un certo tempo.

4.1 Posizioni

I predicati `position` con il primo argomento posto a zero indicano la posizione dell'auto `C` al tempo iniziale. `O` indica il verso dell'auto, mentre `C1`, `C2`, `C3` e `C4` indicano le coordinate sulla griglia dell'auto. Le coordinate sono dipendenti da `O` in questo modo:

- Se `O` viene sostituito con l'atomo `h`, allora `C1` rappresenta la coordinata `X` dell'auto, mentre `C1`, `C2` e `C3` rappresentano le coordinate `Y`;
- Se `O` viene sostituito con l'atomo `v`, allora `C1` rappresenta la coordinata `Y` dell'auto, mentre `C1`, `C2` e `C3` rappresentano le coordinate `X`.

Usando questo approccio siamo in grado di spostare tutta l'auto in un solo colpo senza dover cercare tutti i predicati che appartengono a un'auto.

Il problema di questo approccio è che non è parametrizzabile, quindi se ad esempio si volessero aggiungere auto di dimensione 4 è necessario estendere il programma.

Per questione di comodità, da ora in poi vedremo le regole per le auto di dimensione 2 e orizzontali, ma è intuitivo ricavare le stesse regole per auto di dimensione 3 e poste verticalmente a partire dalle seguenti.

```
position(T+1, C, h, C1, C2, C3) :-
    cause(T, move(_, C), position(T+1, C, h, C1, C2, C3)),
    time(T), time(T+1), car(C),
    grid(C1, C2), grid(C1, C3).
```

Questa regola `position/6` serve ad aggiornare la posizione dell'auto `C` al tempo `T+1` se è stata spostata al tempo `T`.

```
position(T+1, C, h, C1, C2, C3) :-
    not executed(T, move(_, C)),
    position(T, C, h, C1, C2, C3),
    time(T), car(C),
    grid(C1, C2), grid(C1, C3).
```

Questa regola `position/6` corrisponde all'invariante di posizione. Se non è stata compiuta nessuna azione sull'auto `C` al tempo `T` allora la sua posizione sarà la stessa anche al tempo `T+1`.

4.2 Posizioni libere

La regola `not_free/3` indica che una cella (X, Y) della griglia è occupata al tempo T se c'è un auto che occupa quella posizione al tempo T.

```
not_free(T, X, Y) :-
    position(T, _, h, X, Y, _).
not_free(T, X, Y) :-
    position(T, _, h, X, _, Y).
```

La regola `free/3` indica che una cella (X, Y) della griglia è libera al tempo T. se non è occupata al tempo T.

```
free(T, X, Y) :-
    time(T), grid(X, Y),
    not not_free(T, X, Y).
```

4.3 Azioni possibili

La regola `action/2` indica che un'auto `C` può muoversi a destra nel tempo T se la cella alla sua destra immediata non è occupata da nessun'auto ed è valicabile. Lo stesso ragionamento viene applicato per gli spostamenti verso le altre direzioni.

```
action(T, move(right, C)) :-
    car(C), time(T),
    position(T, C, h, X, Y1, Y2),
    free(T, X, Y2+1),
    not no(X, Y2+1),
    grid(X, Y2+1).
```

4.4 Azione scelta

La regola `executed/2` è quella che effettua l'azione, ovvero tra tutte le azioni generate tramite la regola `action/2` ne sceglie una e la esegue.

```
1 {executed(T, move(Dir, C)): action(T, move(Dir, C))} 1 :-
    time(T), T < t.
```

4.5 Cause

La regola `cause/3` è necessario per effettuare l'update della posizione dell'auto `C`, spostata a destra di una casella tramite la regola `executed/2`. Lo stesso ragionamento viene applicato per gli spostamenti verso le altre direzioni.

```
cause(T, move(right, C), position(T+1, C, h, X, Y1+1, Y2+1)) :-
    executed(T, move(right, C)),
    car(C), time(T+1), time(T),
    position(T, C, h, X, Y1, Y2),
    free(T, X, Y2+1),
    not no(X, Y2+1),
    grid(X, Y2+1).
```

5 Risultati

L'analisi del grounding, con l'aggiunta di auto e con tempi maggiori, è diventata sempre più lunga e complessa. È stato sviluppato un programma in python che si occupa di salvare inizialmente le informazioni riguardo la struttura delle auto, prima di disegnare graficamente le auto nelle posizioni che occupano in un dato tempo.

I disegni sono stati fatti tramite la libreria *matplotlib*, sono stati salvati e successivamente riuniti in un file gif per poter vedere facilmente la risoluzione del caso scelto e fornito al programma.

5.1 Confronto versioni

Nell'analisi delle due versioni sviluppate c'è una sostanziale differenza in termini di tempo. La versione più "stringata" (per convenzione la versione 2) e leggermente più complicata da capire a primo impatto, funziona molto meglio ed ha tempi di esecuzione molto più bassi rispetto all'altro programma (versione 1).

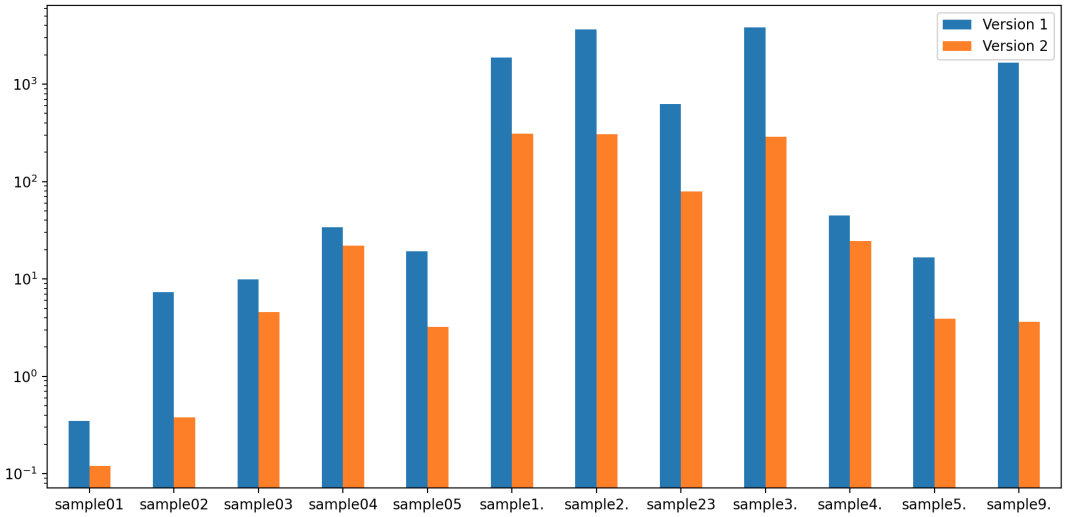


Figura 6: Confronto dei tempi di esecuzione delle due versioni

Per fare il confronto dei tempi mostrato in figura 6, sono state utilizzate le stesse configurazioni, ovviamente codificate nel modo opportuno e senza multishot, facendo raggiungere il goal nel tempo da noi fissato (evitando quindi la `minimize`).

Da notare anche come, ad esempio, per creare l'istanza della configurazione *sample02.lp*, il grounding sia diventato di dimensione:

	Numero righe	*Numero parole*	*Byte*
Versione 1	31184	225893	876816
Versione 2	19248	130794	540640