

Lezione22

Table of contents

- [Problemi di Pianificazione](#)
 1. [Ricerca non informata](#)
 1. [A*](#)
 2. [`banana.pl`](#)
 3. [`blocks.pl`](#)
 4. [`puzzle.pl`](#)

Problemi di Pianificazione



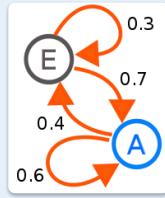
Un **problema di pianificazione** è uno d'identificazione di un insieme di azioni che consenta all'agente di partire da uno stato attuale, o stato presente, arrivando a uno degli stati di goal. All'agente è consentito lavorare in modo autonomo, obbligandolo però a costruire un **piano**, ovvero una sequenza di azioni che permettano a questo di trovarsi a suo agio.

La costruzione automatizzata del piano, o **pianificazione proposizionale**, usa la tecnica a nome **Stanford Research Institute Problem Solver** (STRIPS), sviluppata da Richard Fikes e Nils Nilsson nel '71, allo SRI International. Il problema di pianificazione è una quintupla

$$\langle P, A, I, G, \tilde{G} \rangle$$

in cui abbiamo:

- $P \neq \emptyset$ è un insieme finito di **proposizioni** della solita logica proposizionale (*True*, *False*) che ci permettono di descrivere il mondo;
- $A \neq \emptyset$ è un insieme finito di **azioni** (direzioni sx, dx), che negli esempi che vedremo saranno applicate da piccoli robot in mondi discreti;
- $I \neq \emptyset$ è un sottoinsieme finito delle proposizioni P detto **stato iniziale** del mondo, ovvero una descrizione di come è il mondo in un dato istante ("Oggi piove", "L'agente si trova nell'aula A1");
- $G \neq \emptyset$ è un sottoinsieme finito delle proposizioni P detto **goal asserito**, qualcosa che vogliamo sia vero quando l'agente è soddisfatto (vogliamo sia vero che l'agente si trovi nell'aula I , partendo dalla $A1$);
- $\tilde{G} \neq \emptyset$ è un sottoinsieme finito delle proposizioni P detto **goal negato**, che deve essere vero (negato) quando è vero il goal asserito (vogliamo che l'agente si diriga verso l'aula I ma senza il computer).



Un processo markoviano con probabilità

Una catena di Markov è un modello stocastico (matematico probabilistico) descrivente una sequenza di possibili eventi, in cui la probabilità che ogni evento accada, è determinata da eventi precedenti. Un processo di Markov soddisfa la *proprietà di Markov*: le predizioni che riguardano risultati futuri, basandosi solamente sul presente, sono esatte e tanto precise quanto lo sarebbero, se conoscessimo l'intera catena degli eventi anticipatamente.

I processi di Markov sono base dei metodi di simulazione stocastica Monte Carlo, ma anche applicazione in statistica bayesiana, termodinamica, fisica, chimica, economia, finanza e informatica.

Se il mondo si muove a causa delle azioni dell'agente e queste sono deterministiche, quindi il risultato è noto, allora quello che stiamo cercando è di arrivare a uno stato F tale per cui tutti i goal asseriti sono veri; ma anche tutti quegli stati che non lo sono.

$$G \subseteq F \quad \wedge \quad \tilde{G} \cap F = \emptyset$$

Se pensiamo alla descrizione di un'azione A , ci vengono in mente delle linee di codice. In questo caso però, ci serve una descrizione che ci spieghi quale è il risultato ultimo dell'esecuzione, senza necessariamente descrivere come arrivarci.

$$\langle n, R, \tilde{R}, T, \tilde{T} \rangle$$

- n è il *nome dell'azione*, un simbolo;
- R è l'insieme delle proposizioni che devono essere vere, per rendere l'azione applicabile, detto *precondizione asserita*;
- \tilde{R} è l'insieme delle proposizioni che invece dicono cosa deve essere falso, detto *precondizione negata* tale che $R \cap \tilde{R} = \emptyset$;
- T è ciò che sarà vero una volta che l'azione è stata completata, detto *postcondizione asserita*;
- \tilde{T} è ciò che sarà falso dopo l'azione, detto *postcondizione negata* tale che $T \cap \tilde{T} = \emptyset$.

L'agente deciderà di compiere l'azione quando le precondizioni asserite sono tutte vere nel mondo corrente e le precondizioni negate sono tutte false

$$R \subseteq S \quad \wedge \quad \tilde{R} \cap S = \emptyset$$

e compiute queste, l'agente si porterà in un mondo descritto dallo stato di partenza, tolte le postcondizioni negate e aggiunte le postcondizioni asserite

$$S' = (S \setminus \tilde{T}) \cup T$$

Fare questo vuole dire che pianificare è una *ricerca nello spazio degli stati del mondo*, come nella costruzione di un albero, dove la ricerca dei nodi azioni e i nodi figli associati, porta in stati del mondo ottenibili. L'albero di ricerca viene costruito, non fornito: fintanto che all'azione che vogliamo non riusciamo ad arrivare, continuiamo la ricerca con una possibilità di incontrare un ciclo comunque non trascurabile. La costruzione dell'albero viene seguita per la *frangia* (insieme dei nodi foglia).

1. L'albero viene inizializzato con una frangia formata da un nodo che contiene lo stato iniziale I . I nodi dell'albero potrebbero anche contenere altre informazioni, come il costo.
2. Se lo stato iniziale è soddisfacente, ovvero contiene tutti i G ma non \tilde{G} , allora l'agente si trova nella sua configurazione finale.

3. Altrimenti, viene prelevato un nodo dalla frangia H . Il nodo viene verificato se essere uno stato di goal o meno: se così è, la sequenza di azioni è stata trovata; se così non è, il nodo viene espanso costruendo i suoi figli, cercando la sequenza di azioni P applicabili.
 1. Se abbiamo trovato uno stato non condizionale in cui i goal sono soddisfatti, allora quello è lo stato di goal S a cui siamo arrivati: ricostruire la sequenza di azioni P per arrivarci, fornisce il piano.
 2. Se a forza di espandere la frangia, togliendo e inserendo, tagliando i cicli, la frangia si svuota, allora il goal è irraggiungibile.

Ricerca non informata

L'aggiunta del nodo può essere modificata, presentando diversi metodi di espansione.

- Il metodo **Breadth-First Search** (BFS) prende un nodo e lo espande con conseguente analisi del livello in ampiezza. L'aggiunta del nodo sul fondo comporta che l'albero debba essere per forza memorizzato nella sua interezza, o meglio, in ogni istante la frangia dell'albero deve essere in memoria, perché tutti i figli vanno analizzati: in un istante, dobbiamo avere in memoria tutti i figli del nodo espanso.
- Il metodo **Depth-First Search** (DFS) aggiunge un nodo alla testa della frangia, cercando nello spazio degli stati in profondità. Se la profondità è finita, un ramo che raggiunga la foglia corrente è sufficiente in memoria: la quantità di memoria da usare è lineare rispetto le azioni necessarie per arrivare al goal.

Questi due tipi di costruzione dell'albero di ricerca e di pianificazione, prendono il nome di **ricerche non informate**, perché alla fine quello che a noi interessa è trovare una sequenza più breve con un certo costo.

Una ricerca via di mezzo, esiste e si chiama **ricerca per approfondimenti successivi** della ricerca in profondità (Iterative Deepening Depth-First Search ID-DFS)

- Una ricerca DFS viene usata andando a fissare una profondità massima n , al cui possono succedere 2 opzioni:
 1. viene trovato un nodo goal con meno di n azioni, il costo è lineare;
 2. viene espanso l'albero fino al livello n ma il goal non viene trovato.
- Il massimo della ricerca n viene aumentato, ripartendo dall'inizio nella ricerca e ripetendo le stesse procedure di aumento azioni, fintanto che questo numero non superi la profondità massima che sappiamo esistere, dell'albero.

Lo svantaggio è che tutte le volte viene ricostruito l'albero: l'obiettivo di mantenere basso l'uso di memoria, non riusciamo a raggiungerlo con questo metodo, ma possiamo tenere conto del fatto che a causa degli incrementi di n , DFS viene applicato ad alberi che condividono la parte più vicina alla radice. I nodi dei primi livelli devono essere ricostruiti, senza però peggiorare la già complessità computazionale asintotica. Inoltre da dire che implementare la ricerca ID-DFS in linguaggio, è più semplice data la natura ricorsiva dell'algoritmo DFS.

A*

Una miglioria degli algoritmi visti, può essere fatta introducendo un costo positivo e additivo. Per ogni nodo N dell'albero, viene identificata la funzione di costo, non associata allo stato ma al nodo. La **funzione euristica** di costo f è definita come il costo minimo necessario $g(N)$, per partire dalla radice I , passare dal nodo N e arrivare allo stato di goal h a costo minimo. Spezzata in addendi, la funzione può essere definita come segue

$$f(N) = h(N) + g(N)$$

dove $g(N)$ è il costo necessario per raggiungere nodo N , mentre $h(N)$ il costo per raggiungere il goal.

h spesso non è disponibile, costringendoci a usare la funzione $h^*(N)$ tale che

$$0 \leq h^*(N) \leq h(N)$$

La funzione $h^*(N)$ ci dà una stima, anche se grezza, del costo di h . Possiamo per esempio ipotizzare che $h^*(N) = 0$, indicando che h è determinata da g . Se troviamo una stima che non eccede mai il costo effettivo per andare da nodo a goal interessato, allora la funzione è euristica (motivo per cui così si chiama). Continuando con questa ipotesi, possiamo dire che se la funzione $f^*(N)$ è uguale a $h^*(N) + g(N)$, ovvero che approssima abbastanza bene quella che dovrebbe essere la $f(N)$ originale, dandoci il minimo numero di passi per raggiungere il goal, allora l'algoritmo prenderà il nome di A^* .

A^* è un algoritmo per la ricerca del nodo più vicino alla radice, con il costo per raggiungere suddetto nodo, il più basso possibile. Proprio come gli altri algoritmi visti, anche questo ha un costo asintotico nel caso pessimo $\mathcal{O}(b^d)$ con b la media dei successori per stato (branching factor) e d la quantità di nodi espansi alla profondità della soluzione, in quanto tutti i nodi vengono inseriti in memoria.

📄 La nascita di A^* come parte di un progetto



Peter E. Hart



Nils John Nilsson



Bertram Raphael

A^* (pronunciato "A-star") è uno degli algoritmi di ricerca più conosciuti e diffusi nell'informatica data la sua completezza, ottimità ed efficienza. Presentato per la prima volta dagli informatici Peter Hart, Nils Nilsson e Bertram Raphael allo SRI International nel '68, questo algoritmo è tutt'oggi ancora la scelta predominante rispetto ad altri più efficienti.



Shakey al Computer History Museum

A^* nasce come parte del progetto Shakey, il quale scopo era quello di costruire un robot capace di pianificare le proprie azioni. L'idea originale dell'algoritmo era quella di fare una ricerca per cui la somma dei nodi è la minore, con obiettivo un nodo termine; studi successivi hanno però dimostrato che per un costo algebrico fissato, A^* può essere utilizzato per ricercare i passaggi ottimi per qualsiasi problema di ricerca.



Il logo di Google Maps

Nonostante Google **G** non abbia mai rilevato quale algoritmo di ricerca usi, possiamo supporre che una versione di A^* , in concomitanza con l'algoritmo di Dijkstra, venga usata dall'applicazione Google Maps, per la ricerca del percorso più breve per raggiungere una destinazione. Invece che avere delle funzioni probabilistiche che definiscono i pesi per ciascun nodo, o meglio, manovra, i pesi sono descritti da tempi di percorrenza con vincoli aggiuntivi come i tempi di attesa per ciascuna intersezione semaforica.

Il problema della scimmia e della banana è uno dei problemi più popolarizzati nello studio dell'Intelligenza Artificiale. Originariamente creato per verificare l'intelligenza di un primate, questo problema può essere utilizzato nel campo dell'informatica, per verificare che un dato programma, sia capace di raggiungere un goal, nel minore tempo possibile. Il problema è formulato come segue.

Una scimmia si trova in una stanza. Sospese al soffitto, ci sono delle banane, non raggiungibili dalla scimmia. Nella stanza ci sono tuttavia una sedia e un bastone. La scimmia sa come muoversi nella stanza, portarsi appresso degli oggetti, usare la sedia per raggiungere la banana o agitare il bastone nell'aria. Quale è la migliore sequenza di azioni per la scimmia, in modo che questa riesca a raggiungere le banane?

Gruppo Ricercatori e Utenti Logic Programming



Il logo di GULP

Anche nota come GULP, l'associazione italiana di programmazione logica dal 1986 organizza ogni anno una conferenza che prende il nome di "Conferenza Italiana di Logica Computazionale" (CILC). L'associazione ha proprio come logo, una scimmia che indulge a mangiare una banana, come nel problema classico.

All'inizio del programma, una serie di azioni (`action`) sono definite. La scimmia è prevista essere sul pavimento e di non avere già la banana. Una volta eseguito il programma, questo ci garantirà che la scimmia abbi la banana in mano e che questa sia ancora presente nella stanza.

Le posizioni (`position`) nella stanza sono enumerate all'inizio e sono 5: la stanza è formata da queste possibilità nello spazio.

```
% ...  
action(climb_up,  
      [at(X), box_at(X), floor],  
      [],  
      [],  
      [floor]) :-  
    position(X).  
% ...
```

Salire sulla scatola (`climb_up`) prevede che la scimmia sia in posizione `X`, che la scatola anche lei sia in `X` e che la scimmia sia sul pavimento (`floor`). Questo ci basta per far sì che la scimmia non sia più sul pavimento. In un'azione, genericamente, le cose che indichiamo sono:

1. un'azione, che in questo caso è `climb_up`, il nome;

2. la precondizione asserita, quello che deve essere vero per rendere l'azione fattibile;
3. la precondizione negata, quello che deve essere falso per rendere l'azione fattibile;
4. cosa sarà `true`. dopo che l'azione viene compiuta;
5. cosa sarà eliminato una volta compiuta l'azione.

```
% ...
action(take_banana,
      [at(X), banana_at(X)],
      [floor],
      [have_banana],
      [banana_at(X)]) :-
    position(X).
% ...
```

Anche se ci sembra che gli argomenti dell'azione siano variabili, così non è, siccome `take_banana` può essere applicata sulle sole 5 posizioni nella stanza e non altro.

```
% ...
plan(Plan, State, GoalPositive, GoalNegative) :-
    plan_dfs(Reversed, [node(State, [State], [])], GoalPositive, GoalNegative),
    reverse(Plan, Reversed),
    !.
% ...
```

Un predicato `plan` si occupa di costruire un piano.

Lo stato di partenza è lo stato attuale `State`, l'insieme dei goal che vogliamo asserire `GoalPositive` e l'insieme dei goal che vogliamo negare `GoalNegative`. L'argomento che produce è il piano `plan_dfs`, lista delle azioni dove non ci saranno mai variabili siccome valorizzate dalla parte condizionale della clausola.

```
% ... TODO: uncomment
test(Plan) :-
    plan(Plan,
        [floor, at(south), box_at(east), banana_at(center)],
        [have_banana, floor],
        []).
% ...
```

Il predicato `test`, in cima al programma commentato, serve a fare la pianificazione di un esame.

Lo stato di partenza è lo stato attuale `floor`, la posizione è `south`, la scatola è a `est` e la banana è `center`. Il risultato del test deve essere la scimmia sul pavimento con in mano la banana (`[have_banana, floor]`), la postcondizione negata non c'è (`[]`).

```
% ...
plan_dfs(Plan, Fringe, GoalPositive, GoalNegative) :-
    Fringe = [FringeHead | FringeRest],
    actions([], Actions),
    expand(Expanded, FringeHead, Actions),
    append(Expanded, FringeRest, NewFringe),
    plan_dfs(Plan, NewFringe, GoalPositive, GoalNegative),
```

```
!.
% ...
```

Viene costruita la frangia andando a scegliere tutte le azioni che possono essere applicate dal nodo di testa che abbiamo. In `plan_dfs` c'è sempre una lista vuota e una lista prodotta di azioni (`actions([], Actions)`), che ha lo scopo di prelevare tutte le azioni scritte nel sorgente. La testa della frangia `FringeHead` viene presa, le azioni che l'agente ha a disposizione vengono memorizzate `Actions` e vengono costruiti i nodi espansi `Expanded`: tutti i nodi che possiamo raggiungere partendo dal nodo di frangia, vengono costruiti. I nodi espansi vengono uniti al resto della frangia mediante `append`: i nodi espansi vengono passati `Expanded`, il resto della frangia passato `FringeRest`, ottenendo una nuova frangia `NewFringe`. La pianificazione viene ripresa una volta ancora e finisce la ricorsione soltanto quando il goal viene trovato, perché non c'è altro da fare.

```
% ...
expand([node(Post, [State | States], [Action | Actions]) | ExpandedRest], Node, [Action |
ActionRest]) :-
    Node = node(State, States, Actions),
    action(Action, PrePositive, PreNegative, PostPositive, PostNegative),
    subset(PrePositive, State),
    intersection(PreNegative, State, []),
    subtract(State, PostNegative, Post1),
    union(Post1, PostPositive, Post),
    nonin(Post, States),
    expand(ExpandedRest, Node, ActionRest),
    !.
expand(Expanded, Node, [_ | ActionRest]) :-
    expand(Expanded, Node, ActionRest),
    !.
expand([], _, []).
% ...
```

L'espansione `expand()` è la parte più complessa del programma.

- Un nodo `node` è un simbolo di funzione che raggruppa
 - gli stati `State` in cui si trova;
 - gli stati attraversati `States` fino ad arrivare al nodo, che ci serve in modo da tagliare i cicli (non attraversiamo 2 volte uno stato);
 - la sequenza di azioni `Action | Actions` che ci hanno portato fino al nodo corrente.

Utilizzando la definizione e formule viste, riusciamo a costruire gli stati futuri e per ognuno di questi riusciamo a espandere il resto della frangia. `subset()`, `intersection()`, `subtract()` e `union()` sono operazioni che trattano le liste come fossero insiemi

Il goal

```
test(Plan).
```

esegue un esame del programma con il test fornito.

Quello che ci viene detto è che utilizzando la ricerca in profondità con DFS, prima la scimmia si muove a nord partendo da sud, poi torna al centro e infine raggiunge l'est della stanza. Anche la scatola viene spostata inutilmente a nord, anziché essere portata sul centro subito. Sappiamo che per come è fatto l'algoritmo di ricerca, non per certo viene trovato il percorso minimo e questo è proprio il caso.

```
test(Plan).  
  
Plan = [ move_to(north),  
         move_to(center),  
         move_to(east),  
         move_box_to(north),  
         move_box_to(center),  
         climb_up,  
         take_banana,  
         climb_down  
       ]  
  
0.870 seconds cpu time
```

Risultato al goal `test(Plan)`.
Notare il tempo di CPU impiegato dal server

blocks.pl

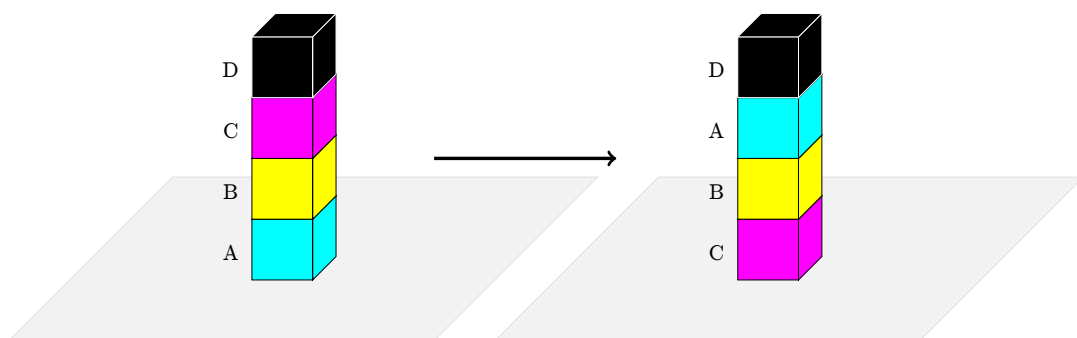
 [Vedere il programma](#) blocks.pl

L'esempio che vediamo, di un programma utilizzando BFS, è più classico ancora dell'esempio della scimmia e della banana, per quando si parla di pianificazione.

Esiste un mondo formato da 4 cubi (`block(a..d)`) e un braccio meccanico fissato al soffitto. Un goal che possiamo richiedere, è la costruzione di una pila formata da questi blocchi, andando a prendere i singoli blocchi originariamente su di un tavolo.

In termini di codice, quello che cambia sono le azioni, i piani e il nome del piano che sarà `plan_bfs()`. Tutto il resto rimane identico all'esempio visto sopra.

```
% ... TODO: uncomment  
test(Plan) :-  
    plan(Plan,  
        [hand_empty, on_table(a), on(b, a), on(c, b), on(d, c), clear(d)],  
        [hand_empty, on_table(c), on(b, c), on(a, b), on(d, a), clear(d)],  
        []).  
  
% ...
```



Il predicato `test` è quello che prevede di costruire il piano.

Partiamo da uno stato iniziale in cui la mano robotica è vuota (`hand_empty`), il blocco `a` è sul tavolo (`on_table(a)`), il blocco `b` è appoggiato su di `a` (`on(b, a)`), il blocco `c` poggiato su `b`, il blocco `d` su `c` e non c'è niente sul blocco `d` (`clear(d)`). Stiamo partendo da una pila di cubi già predisposta: vogliamo arrivare a un'altra configurazione della pila, ovvero il goal.

Il goal a cui vogliamo arrivare è la mano libera, blocco `c` sul tavolo, blocco `b` su blocco `c`, blocco `a` su `b`, blocco `d` su `a`, con nulla sopra al blocco `d`.


```
% ...
action(take_from_table(X),
    [hand_empty, clear(X), on_table(X)],
    [],
    [holding(X)],
    [hand_empty, clear(X), on_table(X)]) :-
    block(X).
% ...
```

L'agente ha a disposizione l'azione `take_from_table()` che prende un blocco `X` sul tavolo. Se il blocco non ha altro blocco sopra di lui e il braccio meccanico è vuoto, allora è possibile prendere il blocco. Se viene preso, il braccio tiene il blocco (`holding(X)`) e non sarà più vero che questo è libero, che non c'è niente sopra `X` e che `X` sia sul tavolo.

```
% ...
action(put_on_table(X),
    [holding(X)],
    [],
    [hand_empty, on_table(X), clear(X)],
    [holding(X)]) :-
    block(X).
% ...
```

`put_on_table()` si comporta al contrario.

Con in mano `X`, mettiamo questo sul tavolo. Non staremo più tenendo il blocco, la mano sarà libera e non ci sarà niente sopra `X`.

```
% ...
action(take(X),
    [hand_empty, clear(X), on(X, Y)],
    [],
    [holding(X), clear(Y)],
    [hand_empty, clear(X), on(X, Y)]) :-
    block(X),
    block(Y),
    X \= Y.
% ...
```

`take()` funziona nello stesso modo ma non agiamo sopra il tavolo, piuttosto su di un altro blocco appoggiato. `X` non sarà più sul tavolo, ma sarà vero l'essere su `Y`. Se sorge la domanda "come mai anche se l'operazione è la stessa, ci serve un predicato diverso dal prendere sul tavolo?", possiamo rispondere dicendo che l'agente ha azioni che gli sono fornite e non può inventarsele: proprio per come è fatto, così è prevista esistere una funzione apposita che permetta il prelevamento da un blocco, piuttosto che dal tavolo.

```
% ...
action(put_on(X, Y),
    [holding(X), clear(Y)],
    [],
    [hand_empty, clear(X), on(X, Y)],
    [holding(X), clear(Y)]) :-
    block(X),
    block(Y),
```

```
X \= Y.  
% ...
```

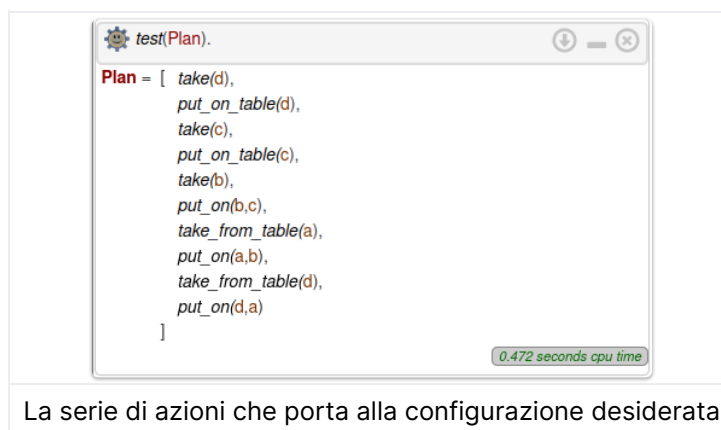
`put_on()` assomiglia a `put_on_table()` ma l'operazione viene fatta su di un altro blocco.

Il goal

```
test(Plan).
```

esegue un esame con il test fornito nel programma.

Il problema del mondo dei blocchi è uno dei più studiati perché questo si comporta bene anche per approcci più complicati.



puzzle.pl

 [Vedere il programma](#) puzzle.pl

Questo esempio di puzzle, utilizza l'algoritmo di ricerca passi minimi A^* . Una funzione euristica, per come l'algoritmo è fatto, deve essere fornita.

Il puzzle è composto da 3×3 tessere, 9 sono le celle di cui una vuota \square . L'unico predicato che viene messo è `at()`: ci dice la riga, la colonna e che numero c'è in quella posizione. Dato uno stato iniziale, vogliamo raggiungerne un altro.

6	5	2		1	2	3
4	8	3	→	4	5	6
1	7	\square		7	8	\square

Le azioni che abbiamo a disposizione sono le 4 azioni parametriche che muovono un certo N , quindi anziché muovere le caselle come verrebbe naturale, muoviamo una tessera alla volta, al posto di quella vuota. N è un numero 1..8, R sono le righe 0..2 e C le colonne 0..2.

```
% ...  
action(move_right(N),  
    [at(R, C, N), at(R, CP1, 0)],  
    [],  
    [at(R, CP1, N), at(R, C, 0)],  
    [at(R, C, N), at(R, CP1, 0)]) :-  
    label(N),  
    row(R),  
    column(C),
```

```

C < 2,
CP1 is C + 1.
% ...

```

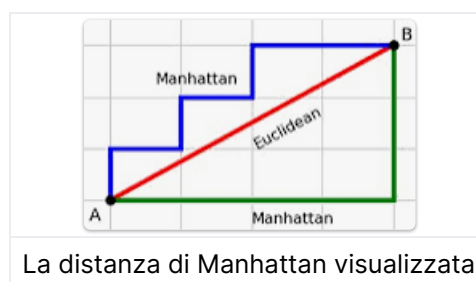
Per esempio: quando possiamo muovere la tessera **N** a destra (`move_right()`)? La preconditione è che la tessera **N** in posizione **R** (row) e **C** (column), la tessera **R** in posizione **CP1** (**C** + 1 con **C**<2) è vuota (\square). Stiamo dicendo che per muovere a destra **N**, questa non può essere sul bordo e quello che c'è in mezzo deve essere nella stessa riga della posizione vuota, che stiamo spostando a sinistra. Compiuta l'azione: **N** si è spostato dove prima c'era \square , che si è spostato al posto di **N**, non è più vero che \square è in scorsa posizione e lo stesso per la tessera **N**.

```

% ...
h_star(HS, node(State, _, _)) :-
    h_star(HS, State, 0, 8).
% ...

```

L'algoritmo A^* prevede sia disponibile una funzione h^* (**HS**), che dipende dal problema. La funzione **HS** quantifica un "grado di qualità" dello stato che in questo caso è "quante celle sono fuori posto" e "quanto le celle sono fuori posto". La distanza tra 2 punti, tracciando segmento nello spazio euclideo, è la **distanza euclidea**; la distanza tra 2 punti, data dalla somma della distanza orizzontale più la distanza in verticale, prende il nome di **distanza di Manhattan**. Questa distanza sarà la stima che viene fatta dalla posizione attuale del nodo **N** e la posizione che deve avere nel goal.



Nel momento in cui ci è necessario aggiungere un nodo nella frangia, costruiamo una lista ordinata in senso crescente per `h_star`: quello che costruiamo è una lista di priorità che permette di usare `plan_a_star()` come approssimazione della politica ottima.

Il goal

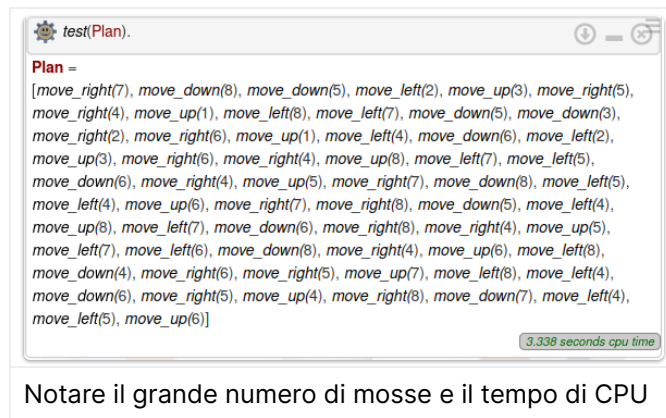
```

test(Plan).

```

produce la serie di mosse per raggiungere la configurazione richiesta.

La quantità di mosse sarà grande perché per come è fatto il puzzle, il raggiungimento del goal richiede molte mosse per spostare lentamente le caselle nelle posizioni adeguate. La sequenza a costo minimo di mosse è calcolata andando a utilizzare nel modo migliore possibile la ricerca, compatibilmente con h^* : funzioni di approssimazione migliori, potrebbero ridurre i tempi di ricerca.



```
test(Plan).  
  
Plan =  
[move_right(7), move_down(8), move_down(5), move_left(2), move_up(3), move_right(5),  
move_right(4), move_up(1), move_left(8), move_left(7), move_down(5), move_down(3),  
move_right(2), move_right(6), move_up(1), move_left(4), move_down(6), move_left(2),  
move_up(3), move_right(6), move_right(4), move_up(8), move_left(7), move_left(5),  
move_down(6), move_right(4), move_up(5), move_right(7), move_down(8), move_left(5),  
move_left(4), move_up(6), move_right(7), move_right(8), move_down(5), move_left(4),  
move_up(8), move_left(7), move_down(6), move_right(8), move_right(4), move_up(5),  
move_left(7), move_left(6), move_down(8), move_right(4), move_up(6), move_left(8),  
move_down(4), move_right(6), move_right(5), move_up(7), move_left(8), move_left(4),  
move_down(6), move_right(5), move_up(4), move_right(8), move_down(7), move_left(4),  
move_left(5), move_up(6)]  
  
3.338 seconds cpu time
```

Notare il grande numero di mosse e il tempo di CPU

18/05/2023