
ANNO ACCADEMICO 2024/2025

Intelligenza Artificiale e Laboratorio

Teoria

Altair's Notes



DIPARTIMENTO DI INFORMATICA

CAPITOLO 1	INTRODUZIONE	PAGINA 5
-------------------	---------------------	-----------------

- | | | |
|-----|---|---|
| 1.1 | Il Corso in Breve...
Motivazioni — 5 | 5 |
|-----|---|---|

CAPITOLO 2	IL PROLOG	PAGINA 8
-------------------	------------------	-----------------

- | | | |
|-----|--|----|
| 2.1 | Le Basi
Liste — 10 | 8 |
| 2.2 | Interprete PROLOG
Breve Ripasso di Logica — 11 • Risoluzione SLD — 13 • Il Cut — 14 | 10 |
| 2.3 | Strategie di Ricerca in PROLOG
Ricerca nello Spazio degli Stati — 16 • Cammini (Labirinto) — 16 • Strategie di Ricerca — 17 | 15 |

CAPITOLO 3	ANSWER SET PROGRAMMING	PAGINA 19
-------------------	-------------------------------	------------------

Premessa

Licenza

Questi appunti sono rilasciati sotto licenza Creative Commons Attribuzione 4.0 Internazionale (per maggiori informazioni consultare il link: <https://creativecommons.org/version4/>).



Formato utilizzato

Box di "Concetto sbagliato":

Concetto sbagliato 0.1: Testo del concetto sbagliato

Testo contenente il concetto giusto.

Box di "Corollario":

Corollario 0.0.1 Nome del corollario

Testo del corollario. Per corollario si intende una definizione minore, legata a un'altra definizione.

Box di "Definizione":

Definizione 0.0.1: Nome delle definizioni

Testo della definizione.

Box di "Domanda":

Domanda 0.1

Testo della domanda. Le domande sono spesso utilizzate per far riflettere sulle definizioni o sui concetti.

Box di "Esempio":

Esempio 0.0.1 (Nome dell'esempio)

Testo dell'esempio. Gli esempi sono tratti dalle slides del corso.

Box di "Note":

Note:-

Testo della nota. Le note sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive.

Box di "Osservazioni":

Osservazioni 0.0.1

Testo delle osservazioni. Le osservazioni sono spesso utilizzate per chiarire concetti o per dare informazioni aggiuntive. A differenza delle note le osservazioni sono più specifiche.

1

Introduzione

1.1 Il Corso in Breve...

1.1.1 Motivazioni

Definizione 1.1.1: Intelligenza Artificiale

L'intelligenza artificiale (o IA, dalle iniziali delle due parole, in italiano) è una disciplina appartenente all'informatica che studia i fondamenti teorici, le metodologie e le tecniche che consentono la progettazione di sistemi hardware e sistemi di programmi software capaci di fornire all'elaboratore elettronico prestazioni che, a un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana.

Note:-

Meh, in realtà l'IA è una disciplina di confine. Però le tematiche sono prettamente informatiche.

IA In breve:

- Area di ricerca dell'informatica.
- Si occupa di tutto ciò che serve per rendere un computer intelligente come un essere umano.
- Interessata a problemi *intelligenti*: problemi per cui non esiste/non è noto un algoritmo di risoluzione¹.

Note:-

Il cubo di Rubik non è un gioco intelligente >:(

Ci sono tante sotto-aree di ricerca:

- Rappresentazione della conoscenza e ragionamento.
- Interpretazione/sintesi del linguaggio naturale.
- Apprendimento automatico.
- Pianificazione.
- Robotica.

¹Tris, il labirinto, etc.

Si collega a tante discipline, oltre all'informatica:

- Filosofia.
- Fisica.
- Psicologia.

Questo insegnamento ha l'obiettivo di approfondire le conoscenze di Intelligenza Artificiale con particolare riguardo alle capacità di un agente intelligente di fare *inferenze* sulla base di una *rappresentazione esplicita della conoscenza* sul dominio. In questo corso si faranno anche sperimentazione di metodi di ragionamento basati sul paradigma della *programmazione logica*, sull'uso di *formalismi a regole* (CLIPS) e su *reti bayesiane* (ragionamento probabilistico²).

Programma:

- Dal punto di vista metodologico saranno a rontate problematiche relative a:
 - Meccanismi di ragionamento per calcolo dei predicati del primo ordine.
 - Programmazione logica.
 - Ragionamento non monotono.
 - Answer set programming.
- Queste metodologie verranno a rontate dal punto di vista sperimentale con l'introduzione dei principali costrutti del *Prolog*, lo sviluppo di strategie di ricerca in Prolog e l'utilizzo dell'ambiente *CLINGO* nella risoluzione di problemi in cui sia necessaria l'applicazione di meccanismi di ragionamento non monotono e del paradigma dell'Answer Set Programming.

Domanda 1.1

E le novità dell'AI che vanno di moda?

Risposta: vengono trattate in altri corsi (TLN, RNDL, AAUT, ELIVA, AGINT).

²Odio la probabilità con tutto il mio cuore <3

2

Il PROLOG

Definizione 2.0.1: PROLOG

PROLOG (Programming Logic) è un *linguaggio dichiarativo* basato sul *paradigma logico*:

- Non si descrive cosa fare per risolvere un problema.
- Si descrive la situazione reale con *fatti* e *regole* e si chiede all'interprete di verificare se un *goal* segue oppure no secondo una logica classica.

Note:-

Il PROLOG è equivalente alla logica dei predicati del primordine.

2.1 Le Basi

Definizione 2.1.1: Fatti

Si rappresenta con dei *fatti* un dominio di interesse.

Esempio 2.1.1 (Fatto)

Fatto per descrivere che un alimento contiene più calorie di un altro:

- piuCalorico(wurstel, banana).
- Rappresenta il fatto che il würstel è un alimento maggiormente calorico rispetto alla banana.

Definizione 2.1.2: Regole

Si rappresentano le possibili inferenze con delle *regole*:

`head := subgoal1, subgoal2, ..., subgoaln`

Esempio 2.1.2 (Regola)

`felino(X) := gatto(X)`

Rappresenta la regola che permette di concludere che i gatti sono felini.

Idee di base del PROLOG:

- Regole ricorsive.
- L'interprete analizza i fatti e le regole nell'ordine in cui si trovano nel programma.
- Meccanismo di pattern matching per unificare variabili e termini.
- L'interprete, dato un programma, cerca di dimostrare un goal considerando fatti e applicando regole, nel secondo caso generando sotto-goal.

Definizione 2.1.3: Clausole

Le clausole sono i fatti o le regole. Contengono:

- Atomi:
 - Costanti.
 - Numeri.
- Variabili.
- Termini Composti, ottenuti applicando funtori a termini.

Note:-

Un programma PROLOG è un insieme di clausole.

Osservazioni 2.1.1

- L'estensione dei file PROLOG è 'pl'.
- In PROLOG le variabili hanno l'iniziale maiuscola.
- L'unica struttura dati nativa è la lista.
- Per eseguire swi: swipl.
- Per compilare: ['nomefile.pl'].
- Il comando ';' indica possibili alternative.
- Il comando 'trace.' consente un'esecuzione passo per passo.
- '\+' rappresenta la negazione per fallimento.
- L'ordine è importante perché PROLOG "legge" dall'alto verso il basso.

Qualche predicato *built-in*:

- `var(X)`: indica se X è una variabile.
- `ground(X)`: indica se X è istanziata.
- `atom(X)`: indica se X è atomica.

2.1.1 Liste

Definizione 2.1.4: Lista

La *lista* è la struttura dati principale in PROLOG. Una lista è caratterizzata da una testa e da una coda:

- Testa: primo termine (a sinistra) della lista.
- Coda: la lista dei termini dal secondo (incluso) in poi.

Note:-

Rappresentata come [Head | Tail].

```
?- [1,2,3,4,5] = [Head | Tail].
Head = 1
Tail = [2,3,4,5] = [Head | Tail]
Yes

?- [a, ciao, [], 2, [1, saluti]] = [Head | Tail].
Head = a
Tail = [ciao, [], 2, [1, saluti]]
Yes
```

Figure 2.1: Le liste in PROLOG.

Predicati *built-in*:

- `length(Lista, N)`: ha successo se la *Lista* contiene *N* elementi.
- `member(Elemento,Lista)`: ha successo se la *Lista* contiene il termine *Elemento*.
- `select(Elemento,Lista,Rimanenti)`: rimuove *Elemento* da *Lista* e restituisce *Rimanenti*.

2.2 Interprete PROLOG

Domanda 2.1

Come avviene l'esecuzione di programmi PROLOG?

- Esecuzione mediante *backward chaining* in profondità.
- Si parte dal *goal* che si vuole derivare:
 - *Goal* = congiunzione di formule atomiche G_1, G_2, \dots, G_n .
 - Si vuole dimostrare, mediante risoluzione, che il goal segua logicamente dal programma.
- Una regola $A : -B_1, B_2, \dots, B_m$ è applicabile a G_i se:
 - Le variabili vengono rinominate.
 - A e G_i unificano.

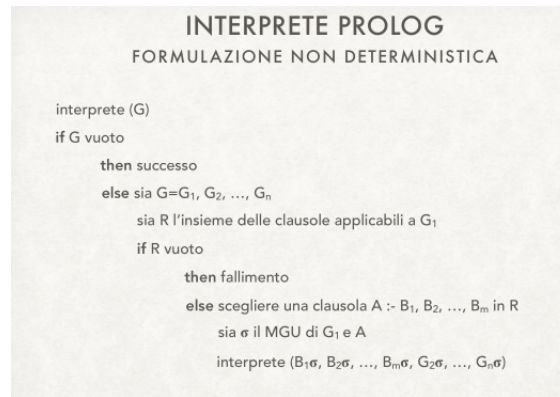


Figure 2.2: Una formulazione non deterministica di come funziona l'interprete PROLOG.

Note:-

MGU è il Most General Unifier: minimo sforzo per rendere uguali due variabili (il fatto e il goal).

- La computazione ha successo se esiste una computazione che termina con successo.
- Non determinismo: non è specificata la regola scelta in R.
- Ma l'interprete PROLOG si comporta in modo *deterministico*:
 - Le clausole vengono considerate nell'ordine in cui sono scritte nel programma.
 - Viene fatto backtracking all'ultimo punto di scelta ogni volta che la computazione fallisce.
- In caso di successo, l'interprete restituisce una sostituzione per le variabili che compaiono nel goal.

2.2.1 Breve Ripasso di Logica**Definizione 2.2.1: Logica Classica**

Conseguenza logica definita semanticamente: dato una teoria e una formula, diciamo che la formula segue dalla teoria se essa è vera in tutti i modelli della teoria.

Esempio 2.2.1 (Gatti)

- I gatti miagolano: $\text{gatto} \rightarrow \text{miagola}$.
- I persiani sono gatti: $\text{persiano} \rightarrow \text{gatto}$.
- Si vuole dimostrare che i persiani miagolano: $k \models \text{persiano} \rightarrow \text{miagola}$.

• Semantica: tavola di verità

$\text{gatto} \rightarrow \text{miagola}$			$\text{persiano} \rightarrow \text{gatto}$			$\text{persiano} \rightarrow \text{miagola}$		
0	1	0	1	0	1	0	0	1
0	1	1	1	0	1	0	0	1
0	1	0	0	1	0	0	1	0
0	1	1	0	1	0	0	1	1
1	0	0	0	0	1	1	0	1
1	1	1	1	0	1	1	0	1
1	0	0	0	1	1	1	1	0
1	1	1	1	1	1	1	1	1

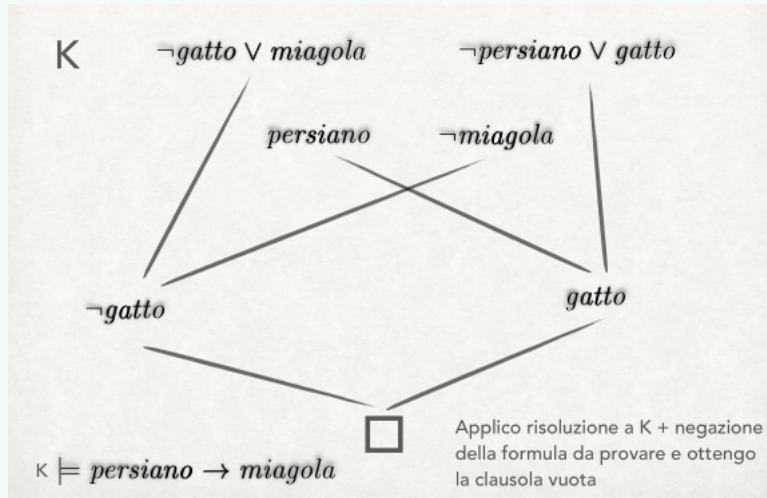
$\text{gatto} \rightarrow \text{miagola} \wedge \text{persiano} \rightarrow \text{gatto}$

- Tuttavia il processo è molto laborioso già con poche formule e basi di conoscenza piccole.
- Metodo di prova: procedura/algorithm che calcola/dimostra se una formula è conseguenza logica della teoria.
 - **Corretto**: se l'algorithm dimostra F da K , allora F è conseguenza logica di K .
 - **Completo**: se F è conseguenza logica di K , allora l'algorithm dimostra F da K .

Risoluzione:

- Si applica a formule in forma di **clausole** (disgiunzioni di letterali¹).
- Si basa su un'unica regola di inferenza:
 - Date due clausole $C_1 = A_1 \vee \dots \vee A_n$ e $C_2 = B_1 \vee \dots \vee B_m$.
 - Se ci sono due letterali A_i e B_j tali che $A_i = \neg B_j$, allora posso derivare la clausola **risolvente** $A_1 \vee \dots \vee A_{i-1} \vee A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m$.
 - Il risolvente è conseguenza logica di $C_1 \cup C_2$
- Data una teoria (insieme di formule) K e una formula F , dimostro che F è conseguenza logica di K per refutazione (dimostrare che $K \cup \neg F$ è inconsistente).
- Si parte dalle clausole $K \cup \neg F$, risolvendo a ogni passo due clausole e aggiungendo il risolvente all'insieme di clausole.
- Si conclude quando si ottiene la clausola vuota.

Esempio 2.2.2 (Risoluzione gatti)



Inoltre:

- Se le due clausole $C_1 = A_1 \vee \dots \vee A_n$ e $C_2 = B_1 \vee \dots \vee B_m$ contengono variabili, i due letterali A_i e B_j devono essere tali che si possa fare l'**unificazione** tra i due:
 - Unificazione: sostituzione α di variabili con termini o uguaglianza di variabili affinché $A_i = \neg B_j$.
 - Clausola risolvente $[A_1 \vee \dots \vee A_{i-1} \vee A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m] \alpha$.
 - Le sostituzioni di α sono applicate a $A_1 \vee \dots \vee A_{i-1} \vee A_{i+1} \vee \dots \vee A_n \vee B_1 \vee \dots \vee B_{j-1} \vee B_{j+1} \vee \dots \vee B_m$.

¹Formule atomiche o negazione di formule atomiche.

	costante c_2	variabile x_2	composto s_2
costante c_1	unificano se $c_1 = c_2$	unificano con x_2/c_1	non unificano
variabile x_1	unificano con x_1/c_2	unificano con x_1/x_2	unificano con x_1/s_2
composto s_1	non unificano	unificano con x_2/s_1	unificano se il functore in s_1 e s_2 è lo stesso e gli argomenti unificano

Figure 2.3: Unificazione di due termini.

Note:-

Per ragioni d'efficienza, PROLOG non fa *occur check*, ossia una variabile X unifica con $f(X)$.

2.2.2 Risoluzione SLD

Per arrivare a un linguaggio di programmazione PROLOG si vuole una strategia efficiente.

Definizione 2.2.2: Risoluzione SLD

Linear resolution with Selection function for Definite clauses:

- K con clausole *definite*:
 - Clausole di Horn: al più un letterale non negato.
 - Strategia linear input: a ogni passo di risoluzione, una *variante* di una clausola è sempre scelta nella K di partenza (programma) mentre l'altra è sempre il risolvente del passo precedente (goal, la negazione di F al primo passo).
 - Variante: clausola con variabili rinominate.

Note:-

NON LSD.

Domanda 2.2

Ma perché ci si limita alle clausole di Horn?

Risposta: si rimuove la parte "intuitiva" che non può essere implementata nel PROLOG. Inoltre le clausole di Horn garantiscono la completezza.

Derivazione SLD per un goal G_0 da un insieme di clausole K è:

- Una sequenza di clausole goal G_0, G_1, \dots, G_n .
- Una sequenza di varianti di clausole di K C_1, C_2, \dots, C_n .
- Una sequenza di MGU $\alpha_1, \alpha_2, \dots, \alpha_n$, tali che G_{i+1} è derivato da G_i e da C_{i+1} attraverso la sostituzione α_{i+1} ,

Tre possibili tipi di derivazioni:

- Successo se G_n è vuoto (**true**).
- Fallimento finito, se non è possibile derivare da G_n alcun risolvibile e G_n non è vuoto (**false**).
- Fallimento infinito, se è sempre possibile derivare nuovi risolventi (loop infinito).

Due forme di non determinismo:

- Regola di calcolo per selezionare a ogni passo l'atomo B_i del goal da unificare con una clausola.
- Scelta di quale clausola utilizzare a ogni passo di risoluzione.

Definizione 2.2.3: Regola di calcolo

Funzione che ha come dominio l'insieme dei goal e per ogni goal seleziona un suo atomo.

Note:-

La regola di calcolo non influenza correttezza e completezza del metodo di prova.

Domanda 2.3

Come si costruisce l'albero SLD?

Data una regola di calcolo, è possibile rappresentare tutte le derivazioni con un albero SLD:

- Nodo: goal.
- Radice: goal iniziale G_0 .
- Ogni nodo $\leftarrow A_1, \dots, A_m, \dots, A_k$, dove A_m è l'atomo selezionato dalla regola di calcolo, ha un figlio per ogni clausola $A \leftarrow B_1, \dots, B_k$ tale che A e A_m sono unificabili con MGU α . Il nodo figlio è etichettato con il goal $\leftarrow [A_1, \dots, A_{m-1}, B_1, \dots, B_k, A_{m+1}, \dots, A_k]\alpha$. Il ramo dal padre al figlio è etichettato con α e con la clausola selezionata.

Scelte per rendere la strategia deterministica:

- Regola di computazione: *leftmost* (viene sempre scelto il sottogol più a sinistra).
- Clausole considerate nell'*ordine in cui sono scritte nel programma*.
- Strategia di ricerca: *in profondità con backtracking*.
 - Non è completa perché se una computazione che porta al successo si trova a destra di un ramo infinito l'interprete non la trova, perché entra, senza mai uscirne, nel ramo infinito.

Note:-

Cercare di mettere a destra le computazioni che possano produrre eventuali casini.

2.2.3 Il Cut

Definizione 2.2.4: Cut

Il *cut* è un predicato extra-logico che consente di modificare l'esecuzione dell'interprete PROLOG. CUT (!):

- Predicato sempre vero.
- Se eseguito blocca il backtracking.

Note:-

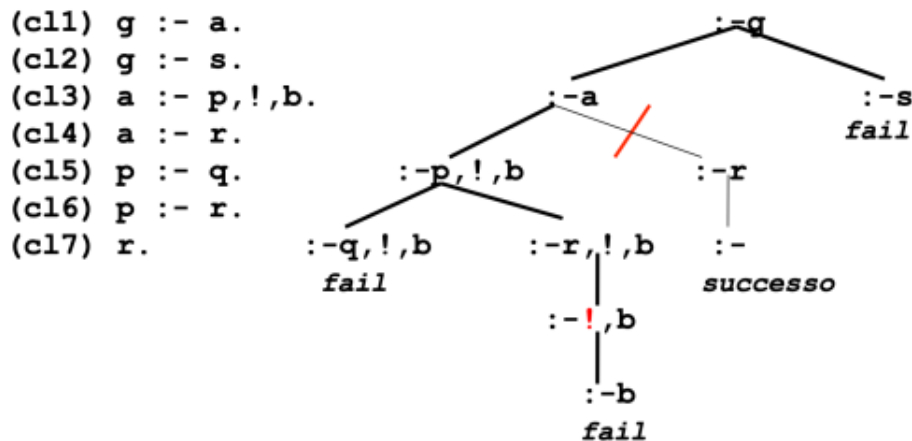
Si rischia di perdere la completezza, ma si guadagna molto in efficienza.

Modello run-time dell'interprete PROLOG:

- Due stack:
 - Stack di *esecuzione*: contiene i record di attivazione (environment) dei predicati in esecuzione.
 - Stack di *backtracking*: contiene l'insieme dei punti di scelta (choice-point).
- In realtà c'è un solo stack, con alternanza di environment e choice-point.

Il cut:

- Rende definitive le scelte fatte nel corso della valutazione dall'interprete PROLOG (eliminazione di choice-point dallo stack di backtracking).
- Altera il controllo del programma.
- Perdita di dichiaratività.



- tagliando alcuni rami dell'albero SLD (=rimuovendo alcuni punti di backtracking) si perde la **completezza**

Figure 2.4: Esempio di cut che provoca la perdita di completezza.

2.3 Strategie di Ricerca in PROLOG

Un problema di ricerca è definito da:

- *Stato iniziale*.
- *Insieme delle azioni* (azione: fa passare da uno stato all'altro).
- Specifica degli obiettivi (goal).
- Costo di ogni azione.

Note:-

Non tutti i problemi hanno una naturale soluzione con la ricerca nello spazio degli stati.

2.3.1 Ricerca nello Spazio degli Stati

Lo spazio degli stati definito implicitamente dallo stato iniziale con un insieme delle azioni, ossia l'insieme di tutti gli stati raggiungibili a partire da quello iniziale.

Definizione 2.3.1: Cammino

Sequenza di stati collegati da una sequenza di azioni.

Corollario 2.3.1 Costo di un Cammino

Somma dei costi delle azioni che lo compongono.

Note:-

Se non si hanno dei costi espliciti si assume che siano tutti uguali (e. g. tutti 1).

Definizione 2.3.2: Soluzione a un Problema

Cammino dallo stato iniziale ad uno stato goal.

Corollario 2.3.2 Soluzione Ottima

Soluzione che ha il costo minimo tra tutte le soluzioni.

Note:-

Non è detto che esista una soluzione. In generale possono esistere 0, 1 o più soluzioni.

Stati rappresentati come termini:

- Dipendono dal problema da rappresentare:
 - Mondo dei blocchi: $on(a,b)$, $clear(c)$, ecc.
 - Puzzle dell'8: lista ordinata $[3, 1, v, 4, 7, 8, 5, 6, 2]$.

Azioni specificate tramite:

- Precondizioni: in quali stati un'azione può essere eseguita.
- Effetti.
- $applicabile(AZ, S)$: l'azione AZ è eseguibile nello stato S .
- $trasforma(AZ, S, NUOVO_S)$: se l'azione AZ è applicabile a S , lo stato $NUOVO_S$ è il risultato dell'applicazione di AZ allo stato S .

2.3.2 Cammini (Labirinto)

Specifiche:

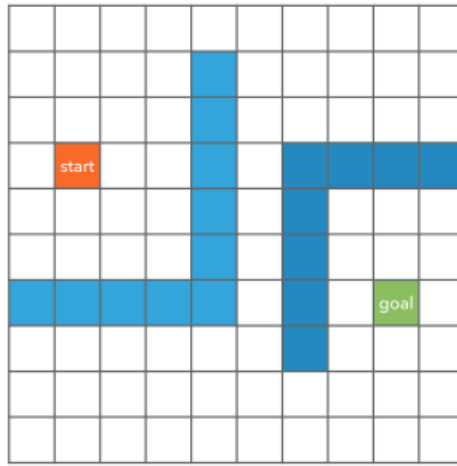
- Trovare un cammino in una griglia rettangolare, con ostacoli in alcune celle.
- Predicati num_righe e $num_colonne$ specificano la dimensione della griglia.
- $pos(Riga, Colonna)$ per rappresentare la posizione dell'agente.
- $occupata(pos(Riga, Colonna))$ per rappresentargli ostacoli.

Azioni:

- Nord.
- Sud.
- Ovest.
- Est.

Azione applicabile quando la sua esecuzione non porta l'agente:

- Fuori dalla griglia.
- In una cella occupata da un ostacolo.



```
num_col(10).
num_righe(10).
```

```
occupata(pos(2,5)). occupata(pos(7,4)).
occupata(pos(3,5)). occupata(pos(5,7)).
occupata(pos(4,5)). occupata(pos(6,7)).
occupata(pos(5,5)). occupata(pos(7,7)).
occupata(pos(6,5)). occupata(pos(8,7)).
occupata(pos(7,5)). occupata(pos(4,7)).
occupata(pos(7,1)). occupata(pos(4,8)).
occupata(pos(7,2)). occupata(pos(4,9)).
occupata(pos(7,3)). occupata(pos(4,10)).
```

```
iniziale(pos(4,2)).
```

```
finale(pos(7,9)).
```

Figure 2.5: Esempio di labirinto.

```
applicabile(nord,pos(R,C)) :-
```

```
R>1,
```

```
R1 is R-1,
```

```
\+ occupata(pos(R1,C)).
```

```
applicabile(sud,pos(R,C)) :-
```

```
num_righe(NR), R<NR,
```

```
R1 is R+1,
```

```
\+ occupata(pos(R1,C)).
```

```
applicabile(ovest,pos(R,C)) :-
```

```
C>1,
```

```
C1 is C-1,
```

```
\+ occupata(pos(R,C1)).
```

```
applicabile(est,pos(R,C)) :-
```

```
num_col(NC), C<NC,
```

```
C1 is C+1,
```

```
\+ occupata(pos(R,C1)).
```

```
trasforma(est,pos(R,C),pos(R,C1)) :- C1 is C+1.
```

```
trasforma(ovest,pos(R,C),pos(R,C1)) :- C1 is C-1.
```

```
trasforma(sud,pos(R,C),pos(R1,C)) :- R1 is R+1.
```

```
trasforma(nord,pos(R,C),pos(R1,C)) :- R1 is R-1.
```

Figure 2.6: Operazioni possibili.

2.3.3 Strategie di Ricerca

3

Answer Set Programming

