Relazione del corso di Ragionamento Automatico (Verona, a.a. 2010-2011)

Paolo Parise

12 aprile 2011

1 Introduzione

Partendo da un'implementazione didattica Prolog [1] del Wumpus World descritto in [2] verranno aggiunte nuove funzionalità e presentata una tecnica di verifica.
L'obbiettivo per l'agente in posizione iniziale [1,1] é di recuperare l'oro evitando il Wumpus e le buche. Una volta recuperato l'oro esegue una procedura di ritorno alla cella iniziale [1,1] e termina. Si rimanda alla letteratura per ulteriori dettagli e varianti del gioco.
Di seguito lo scheletro del programma agente:

```
step :-
        make_percept_sentence(Percept),

        tell_KB(Percept),

        ask_KB(Action),

        apply(Action),

        time(T),
        New_T is T+1,
        retractall(time(_)),
        asserta(time(New_T)),

        step,
        !.
```

Questo *agente basato sulla conoscenza* é implementato secondo il *calcolo delle situazioni* che si appoggia sulla seguente ontologia:

- le azioni eseguibili dall'agente (es. `rebound`,`grab`,`shoot`,`forward`,`turnleft`, etc.);
- le *situazioni* che denotano gli stati risultanti dall'applicazione di un'azione;

- i fluenti ovvero funzioni o predicati che variano da una *situazione* all'altra (es. `agent_location`, `agent_orientation`);
- eventuali funzioni o predicati atemporali (es. `wumpus_location`).

Alla generica *situazione* `S` l'agente percepisce informazioni dall'ambiente (`make_percept_sentence(Percept)`), aggiunge al DB tale conoscenza eventualmente inferendone dell'altra (`tell_KB(Percept)`), sceglie l'azione migliore da fare in base alla conoscenza acquisita in `S` e nelle situazioni precedenti (`ask_KB(Action)`), applica l'azione (`apply(Action)`) e passa alla *situazione* `S+1`.

2 Una mappa arbitraria

L'obbiettivo é di estendere l'ambiente di gioco a mappe di dimensione e forma arbitraria. Sarà possibile ad esempio creare mappe anche non quadrate con mura interne. L'unico vincolo é che la mappa sia interamente circondata da mura. Questo é stato fatto modificando il gestore dell'ambiente e generalizzando l'agente stesso in modo tale che sappia muoversi nel nuovo contesto. Dalla parte del gestore l'idea é di dichiarare escplicitamente le celle `wall(L)` ricordando che la percezione `bump` in `L` é unicamente basata sul valore di verità di `wall(L)`. Il gestore dell'ambiente prima:

```
wall([X,LE]) :- inf_equal(LE,0).
wall([LE,Y]) :- inf_equal(LE,0).
wall([X,Y]) :- land_extent(LE), inf_equal(LE,X).
wall([X,Y]) :- land_extent(LE), inf_equal(LE,Y).
```

Il gestore dell'ambiente dopo:

```
initialize_land(randomMap):-
        % others initializations
        ...
        % list of wall locations
        asserta(wall_location([0,1])),
        ...
        asserta(wall_location([6,6])).
wall(X) :- wall_location(X).
```

Dalla parte dell'agente non é stata apportata alcuna modifica poiché il comportamento é già indipendente dalla morfologia della mappa. Si riporta il codice che genera la percezione `bump`:

```
bumped(yes) :-
        agent_location(L),
        wall(L),
        !.
bumped(no).
```

Di seguito l'aggiornamento della KB dell'agente:

```
add_wall_KB(yes) :-
        agent_location(L),
        retractall(is_wall(L)),
        asserta(is_wall(L)),
        !.
add_wall_KB(no).
```

Di seguito la scelta dell'agente di eseguire il `rebound`:

```
act(strategy_reflex,rebound) :-
        agent_location(L),
        is_wall(L),
        is_short_goal(rebound),!.
```

Di seguito l'esecuzione dell'azione `rebound`:

```
apply(rebound) :-
        agent_location(L),
        agent_orientation(O),
        Back_O is (O+180) mod 360,
        location_toward(L,Back_O,L2),
        retractall(agent_location(_)),
        asserta(agent_location(L2)).
```

Si consulti il file `wumpusWall.pl` per il codice completo.

3 Una strategia di classificazione

L'obbiettivo é di scrivere una strategia per esplorare e classificare le celle di una mappa in celle `Safe (S)` e celle `Unsafe (NS)`.

Definizione 3.1 (Cella `S`)**.** Una cella é classificata `S` secondo le seguenti regole:

- la cella [1,1] é `S`;
- se in `L` non viene percepita ne' `stench` ne' `breeze` allora ogni cella adiacente a `L` é `S`.

Definizione 3.2 (Cella `NS`)**.** Una cella é classificata `NS` se e solo se non é `S`.

L'idea é che l'agente esplori solamente celle sicure (`S`) senza rischiare la caduta in una pit ne' tanto meno nella cella del Wumpus. Si noti che alcune celle `NS` possono risultare effettivamente esplorabili ma solo agl'occhi dell'osservatore esterno che conosce la posizione di ogni pit e del Wumpus. Il problema dell'inferenza certa da parte dell'agente della posizione del Wumpus é trattato nella sezione seguente. L'agente esplorerá la mappa fino a quando non esistono più celle `good` guardando prima le celle `good` adiacenti(`forward,turnleft e turnright`) e poi le celle `good` non adiacenti (`goTo(L)`). Si riporta la definizione di alcune tipologie di celle e la procedura di esplorazione dell'agente:

```
good(L) :-
        is_wumpus(no,L),
        is_pit(no,L),
        no(is_visited(L)).

medium(L) :-
        is_visited(L).

risky(L) :-
        no(deadly(L)).

deadly(L) :-
        is_wumpus(yes,L),
        is_pit(yes,L),
        no(is_visited(L)).

safe(L) :-
        medium(L),
        no(is_wall(L)).

unsafe(L) :-
        no(safe(L)).
```

```
act(strategy_find_out,forward) :-
        ...
        good(_),
        location_ahead(L),
        good(L),
        no(is_wall(L)),
        !.

act(strategy_find_out,turnleft) :-
        ...
        good(_),
        agent_orientation(O),
        Planned_O is (O+90) mod 360,
        agent_location(L),
        location_toward(L,Planned_O,Planned_L),
        good(Planned_L),
        no(is_wall(Planned_L)),
        !.

act(strategy_find_out,turnright) :-
        ...
        good(_),
        agent_orientation(O),
        Planned_O is (O-90) mod 360,
        agent_location(L),
        location_toward(L,Planned_O,Planned_L),
```

```
        good(Planned_L),
        no(is_wall(Planned_L)),
        !.

act(strategy_find_out,goTo) :-
        good(_),
        !.

act(strategy_go_out,climb) :-
        true.
```

Si consulti il file `wumpusExplorator.pl` per il codice completo.

4 Una strategia di ricerca

Le risorse (energia e armi) a disposizione dell'agente possono essere finite, inoltre l'esplorazione della mappa può essere impedita dalla sospetta presenza del Wumpus. In questa sezione viene proposta una strategia per l'individuazione esatta (quando possibile) dell'unico e immobile Wumpus. L'unicità e l'immobilitá del Wumpus semplifica notevolmente la strategia.

Tramite `certein_wumpus(Count,AL)` l'agente mantiene un contatore `Count` per ogni cella `AL`, contatore che viene incrementato ogni volta che percepisce `stench` in `L` adiacente a `AL`. Se esiste un unico massimo tra i contatori `C` di celle `W` non visitate e non `good` (v. `candidate(C,W)`) allora l'agente inferisce la posizione del Wumpus nella cella corrispondente. Se questa condizione non vale e non ci sono più celle `good` da esplorare allora il problema non ha soluzione. Di seguito la condizione di fermata:

```
is_wumpus(WumpusLocation) :-
  findall(C,certein_wumpus(C,_),WC),
  max_list(WC,MaxC),
  findall(Candidate,candidate(MaxC,Candidate),Candidates),
  length(Candidates,1),
  nth(1,Candidates,WumpusLocation).

candidate(C,W) :-
        certein_wumpus(C,W),
        no(medium(W)),
        no(good(W)).
```

Si consulti il file `wumpusHunter.pl` per il codice completo.

5 Aspetti di verifica

Una volta trovato l'oro l'agente segue una strategia di ritorno alla cella [1,1] per poi uscire dalla mappa.
Allo scopo di scrivere un'interrogazione Prolog `check_condition` che determini

la correttezza o meno di una data procedura P su un particolare input I si procederá con l'enunciare e dimostrare una condizione C di correttezza necessaria e sufficiente seguita dalla sua implementazione (query `check_condition`).
La condizione C sará basata sulle seguenti assunzioni:

1. l'insieme delle celle `medium` (M) é finito e costante;
2. la computazione é deterministica;
3. lo stato corrente s_i dell'agente é unicamente individuato da un insieme finito A di asserzioni. La scelta dell'azione futura é basata esclusivamente su A e M;
4. da 1 2 3 segue che se uno stato s_i é visitato due volte allora nessun nuovo stato verrá visitato, graficamente l'agente percorrerá un cammino ciclico $\boldsymbol{s_i} \rightarrow s_{i+1} \rightarrow ... \rightarrow s_{i+n} \rightarrow \boldsymbol{s_i}$ con $n \geq 0$.
5. poiché esiste un cammino da [1,1] a `gold_location` di sole celle M e per l'assunto 1 allora esiste un cammino da `gold_location` a [1,1] di sole celle M dunque esiste sempre soluzione finita per il problema di ritorno. In altre parole P é corretta se e solo se termina in [1,1].

Definizione 5.1 (Correttezza di P per I)**.** P non corretta per $I \Longleftrightarrow$ applicando P ad I, uno stato s_i é visitato due volte o applicando P ad I, P non termina in [1,1].

Lemma 5.1. La procedura non termina $\Longleftrightarrow$ esiste uno stato s_i visitato due volte.

Dimostrazione.
($\Rightarrow$) poiché vale antecedente e il numero dei possibili stati s_i é finito necessariamente c'é almeno uno stato s_i visitato due volte;
($\Leftarrow$) segue da ipotesi 1, 2 e 3.

Definizione 5.1 suggerisce il test per la correttezza di P su I. Ad ogni passo dell'agente `check_condition` verifica se la computazione é terminata in [1,1] o se é in uno stato giá visitato. Si noti che dalle ipotesi fatte segue che il test é decidibile. Il test dovrá conoscere la definizione del generico stato s_i ovvero su quale insieme A di asserzioni si basa la decisione dell'agente ad ogni passo. Nei due casi di studio successivi verrá proposta l'implementazione di `check_condition` per la particolare procedura e considerati i due input I_1 (v. Figura 1) e I_2 (v. Figura 2).

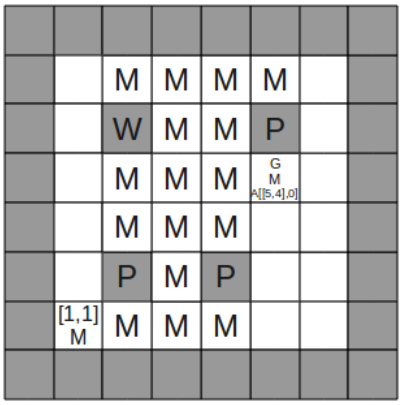

Figura 1: L'input I_1 utilizzato per il testing.

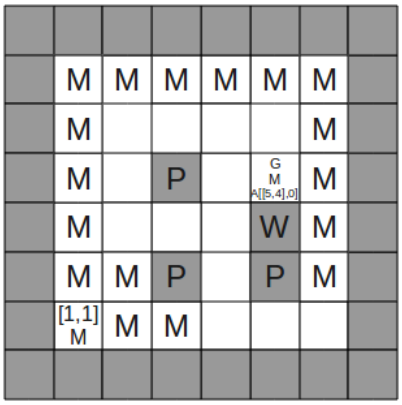

Figura 2: L'input I_2 utilizzato per il testing.

6 Caso di studio: $P1$ verificata su I_1 e I_2

L'agente che segue la strategia `P1` considera in ogni *situazione* le celle `M` adiacenti e nell'ordine si muove nella cella avanti o ruota di 90° a sinistra se la cella a sinistra é `M` o ruota a destra di 90° se la cella a destra é `M` o ruota di 90° verso sinistra. Si rimanda il lettore a [1] per i dettagli implementativi. La scelta dell'azione si basa quindi su $A1 = \{agent\_location, agent\_orientation\}$ e sull'insieme M. Ecco l'implementazione della query `check_condition` eseguita ad ogni *situazione*:

```
check_condition:-
        agent_location(L),
        agent_orientation(O),
        no(state_rec(L,O)),
        asserta(state_rec(L,O)),
        !.

check_condition:-
        agent_location(L),
        no(agent_in_cave),
        L = [1,1],
        !.

check_condition:-
        fail.
```

Si noti che $P1$ in I_1 non termina poiché continuerá a ciclare nelle celle **[4,6]** $\rightarrow$ $[3,6] \rightarrow [2,6] \rightarrow [3,6] \rightarrow [4,6] \rightarrow [5,6] \rightarrow$ **[4,6]** (v. Figura 3). Nella traccia di esecuzione del test su $P1$ e I_1 l'agente visita per la seconda volta lo stato [[4,6],180] (`check_condition` fallita). La procedura $P1$ per I_1 non é corretta infatti l'agente continuerá a ciclare tra gli stati: $\mathbf{[[4,6],180]} \rightarrow [[3,6],180] \rightarrow [[2,6],180] \rightarrow [[2,6],270] \rightarrow [[2,6],0] \rightarrow [[3,6],0] \rightarrow [[4,6],0] \rightarrow [[5,6],0] \rightarrow [[5,6],90] \rightarrow [[5,6],180] \rightarrow \mathbf{[[4,6],180]}$.

Passando all'esecuzione di $P1$ in I_2 (v. Figura 4) l'agente $P1$ termina correttamente in [1,1] seguendo il cammino $[5,4] \rightarrow [6,4] \rightarrow [6,5] \rightarrow [6,6] \rightarrow [5,6] \rightarrow [4,6] \rightarrow [3,6] \rightarrow [2,6] \rightarrow [1,6] \rightarrow [1,5] \rightarrow [1,4] \rightarrow [1,3] \rightarrow [1,2] \rightarrow$ **[1,1]**. Il test identifica sempre un nuovo stato s_i (`check_condition` soddisfatta) e la computazione termina in [1,1] (`check_condition` soddisfatta): $[[5,4],0] \rightarrow [[6,4],0] \rightarrow [[6,4],90] \rightarrow [[6,5],90] \rightarrow [[6,6],90] \rightarrow [[6,6],180] \rightarrow [[5,6],180] \rightarrow [[4,6],180] \rightarrow [[3,6],180] \rightarrow [[2,6],180] \rightarrow [[1,6],180] \rightarrow [[1,6],270] \rightarrow [[1,5],270] \rightarrow [[1,4],270] \rightarrow [[1,3],270] \rightarrow [[1,2],270] \rightarrow \mathbf{[[1,1],270]}$. La procedura $P1$ per I_2 é corretta.

Si consulti il file `strategy1.pl` per il codice completo.

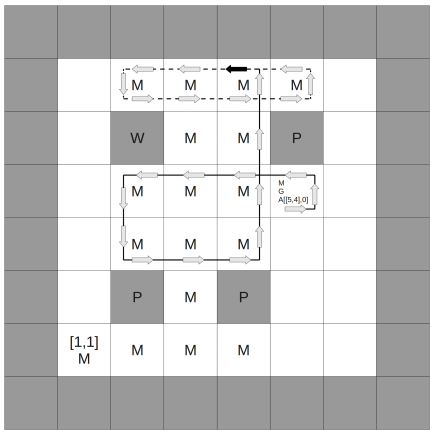

Figura 3: Esecuzione della procedura $P1$ sull'input I_1, le frecce indicano l'orientamento dell'agente. La freccia nera indica il primo stato ripetuto.

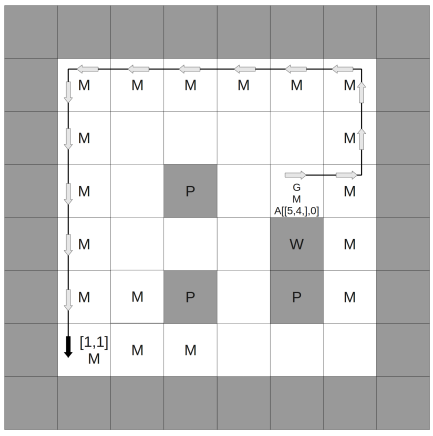

Figura 4: Esecuzione della procedura $P1$ sull'input I_2, le frecce indicano l'orientamento dell'agente.

7 Caso di studio: $P2$ verificata su I_1 e I_2

L'agente che segue la strategia `P2` considera in ogni *situazione* le celle `M` adiacenti e si muove nella cella con coordinata `Y` minima e se non é unica sceglie tra queste quella con coordinata `X` minima. La scelta dell'azione di basa quindi su $A2 = \{agent\_location\}$. Di seguito il codice della strategia:

```
act(strategy_go_out,climb) :-
        agent_location([1,1]),
        !.

act(strategy_go_out,up) :-
        agent_location([XL,YL]),
        minL([XM,YM]),
        YM is YL + 1,
        !.

act(strategy_go_out,down) :-
        agent_location([XL,YL]),
        minL([XM,YM]),
        YM is YL - 1,
        !.

act(strategy_go_out,left) :-
        agent_location([XL,YL]),
        minL([XM,YM]),
        XM is XL - 1,
        !.

act(strategy_go_out,right) :-
        agent_location([XL,YL]),
        minL([XM,YM]),
        XM is XL + 1,
        !.

minL(L) :-
        % from the list of locations Loc[X,Y] such that
        % m_ret(Loc)=true select L with minimal Y value
        % and if there isn't a unique L, select among these
        % what has minimal X.
        % Note that there is always a unique L such that
        % minL(L)= true.

m_ret([X,Y]) :-
        agent_location(L),
        adjacent(L,[X,Y]),
        medium([X,Y]),
        no(is_wall([X,Y])).
```

Per semplicitá sono state aggiunte le azioni `up, down, left, right` che possono essere espresse combinando le azioni native `forward, turnleft` e `turnright` dunque l'agente ha le stesse capacitá motorie dell'agente visto in $P1$. L'implementazione di `check_condition` segue lo schema di quanto fatto per $P1$ ma risulta leggermente semplificata ricordando infatti che per $P2$ l'insieme $A2 = \{agent\_location\}$:

```
check_condition:-
        agent_location(L),
        no(state_rec(L)),
        asserta(state_rec(L)),
        !.

check_condition:-
        agent_location(L),
        no(agent_in_cave),
        L = [1,1],
        !.

check_condition:-
        fail.
```

Applicando $P2$ a I_1 l'agente termina correttamente in [1,1] seguendo il cammino $[5,4] \rightarrow [4,4] \rightarrow [4,3] \rightarrow [3,3] \rightarrow [3,2] \rightarrow [3,1] \rightarrow [2,1] \rightarrow$ **[1,1]** (v. Figura 5).

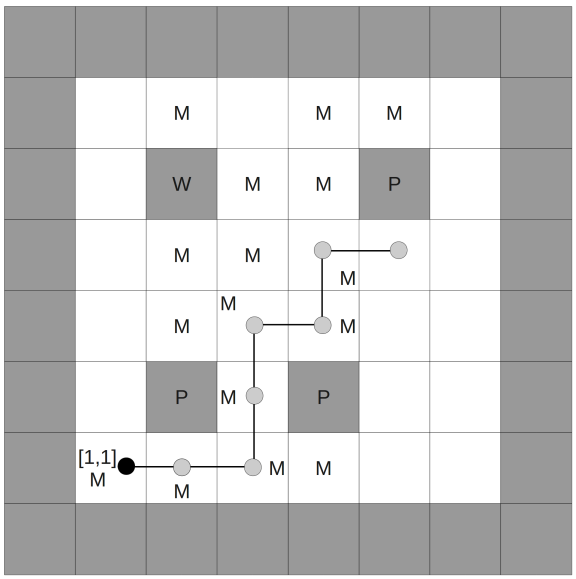

Figura 5: Esecuzione della procedura $P2$ sull'input I_1. Il pallino indica la posizione dell'agente.

Nella traccia di esecuzione del test su $P2$ e I_1 l'agente visita sempre un nuovo stato s_i (`check_condition` soddisfatta) e la computazione termina in [1,1] (`check_condition` soddisfatta): $[5,4] \rightarrow [4,4] \rightarrow [4,3] \rightarrow [3,3] \rightarrow [3,2] \rightarrow [3,1] \rightarrow [2,1] \rightarrow \mathbf{[1,1]}$. La procedura $P2$ per I_1 é corretta.

Si supponga ora che l'agente abbia trovato l'oro e si consideri la *situazione* I_2. La procedura non termina continuando a ciclare nelle celle **[6,3]** $\rightarrow [6,2] \rightarrow$ **[6,3]** (v. Figura 6).

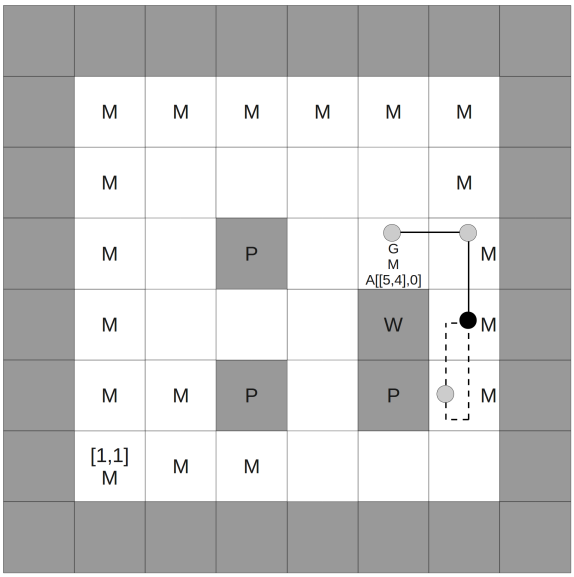

Figura 6: Esecuzione della procedura $P2$ sull'input I_2. Il pallino indica la posizione dell'agente. Il pallino nero indica il primo stato ripetuto.

Nella traccia di esecuzione del test su $P2$ e I_2 l'agente visita per la seconda volta lo stato [6,3] (`check_condition` fallita). La procedura $P2$ per I_2 non é corretta infatti l'agente continuerá a ciclare tra gli stati: $\mathbf{[6,3]} \rightarrow [6,2] \rightarrow \mathbf{[6,3]}$. La procedura $P2$ per I_2 non é corretta.
Si consulti il file `strategy2.pl` per il codice completo.

8 Conclusioni

Vengono proposte alcune idee di sviluppo per tematiche viste nel documento:

- la sezione 4 presenta una strategia per l'inferenza (non sempre possibile) della posizione dell'unico Wumpus immobile nella mappa. La strategia puó essere estesa considerando che la posizione del Wumpus possa cambiare e che esistano piú Wumpus nella mappa magari prevedendo la collaborazione multiagente.

- una volta definito A e I il test presentato in sezione 5 é in grado di dimostrare la correttezza di una procedura P su quel particolare I considerando una serie di assunti. Tuttavia come é stato mostrato nei casi di studio P puó non essere corretta su $I' \neq I$. Qual'ora si voglia testare la correttezza di P su ogni possibile I_i sembra sconveniente enumerare i possibili I_i ed eseguire il test per ogni I_i. Da qui la necessitá di studiare condizioni di correttezza piú potenti o lo studio di I rappresentativi generalizzando il risultato del test all'insieme di input rappresentati.

In generale ogni problema visto nel documento risolve esigenze applicative interessanti ma rimane la necessitá di adeguare la soluzione algoritmica al particolare contesto (multiagente e caratteristiche di comunicazione, diverse prioritá negli obbiettivi da raggiungere, morfologia ambientale etc.).

Riferimenti bibliografici

[1] *http://archives.limsi.fr/Individu/hernandz/resources/software/wumpus/wumpus.html*

[2] S. Russel, P. Norving, *Intelligenza artificiale: un approccio moderno. Vol.1 sez. 7.2 e 10.3*, 2005.