Strategie di ricerca in Prolog

Ricerca nello spazio degli stati

Un problema di ricerca può essere definito dai seguenti componenti:

- Lo **stato iniziale**.
- L'insieme delle **azioni**. Ogni azione fa passare da uno stato ad un altro.
- La specifica degli stati obiettivo (**goal**).
- Il **costo** di ogni azione.

Lo stato iniziale e le azioni definiscono implicitamente lo **spazio degli stati** del problema, ossia l'insieme di tutti gli stati raggiungibili a partire da quello iniziale. Un **cammino** è una sequenza di stati collegati da una sequenza di azioni. Il **costo** di un cammino normalmente è definito come la somma dei costi delle azioni che lo compongono.
La **soluzione** di un problema è un cammino dallo stato iniziale ad uno stato goal. Una **soluzione ottima** è una soluzione che ha il costo minore di tutte.

Rappresentazione in Prolog

In Prolog gli stati possono essere rappresentati come termini, che, naturalmente, dipendono dal problema che si vuole rappresentare. Ad esempio nel mondo dei blocchi gli stati possono essere liste di fatti [*on(a,b), clear(c)*, …], mentre nel caso del puzzle dell'8 uno stato dovrà specificare la posizione degli otto tasselli e dello spazio vuoto.
Le azioni verranno descritte specificando le loro precondizioni (che indicano in quali stati una azione può essere eseguita) ed i loro effetti. Nei nostri esempi useremo i seguenti predicati:

applicabile(AZ, S): l'azione *AZ* è eseguibile nello stato *S*;

trasforma(AZ, S, NUOVO_S): lo stato *NUOVO_S* è il risultato dell'esecuzione dell'azione *AZ* nello stato *S* (assumendo che l'azione sia applicabile).

Nel seguito faremo l'ipotesi che gli stati abbiano una rappresentazione univoca, in modo che sia immediato determinare se due stati sono uguali. Ad esempio, nel caso dei blocchi, si possono rappresentare gli stati come *insiemi ordinati* di fatti, invece che come generiche liste.
In caso contrario si deve fornire il predicato:

uguale(S1, S2): *S1* e *S2* rappresentano lo stesso stato (anche se non sono identici).

I predicati elencati sopra servono per la descrizione generale di un problema. Per definire una particolare istanza useremo i predicati:

iniziale(S): *S* è lo stato iniziale

finale(S): *S* è **uno** stato finale (goal).

Esempi:

Cammini (labirinto)
Il file *cammino* contiene la rappresentazione del problema di trovare un cammino in una griglia rettangolare in cui alcune celle possono contenere degli ostacoli.
Il numero di righe e colonne della griglia è specificato, mediante i predicati *num_righe* e *num_col*, per una istanza del problema. Lo stato è costituito dal termine *pos(Riga,Colonna)*, che rappresenta la posizione dell'agente. La posizione degli ostacoli è specificata con il predicato *occupata*.
Le azioni sono 4, *est, sud, ovest* e *nord*, e consistono nello spostarsi di un passo in una delle 4 celle adiacenti.
Una azione è *applicabile* quando la sua esecuzione non porta l'agente a uscire dalla griglia, o a muoversi in una cella occupata. Il predicato *trasforma* determina la cella adiacente in cui muoversi a seguito dell'esecuzione di una azione.
Il file *esempi cammini* contiene due esempi, uno 10 x 10 e uno 20 x 20.

Blocchi
Il file *blocchi* contiene la rappresentazione del mondo dei blocchi. Assumiamo che le azioni siano specificate alla STRIPS, ossia dando la *lista delle precondizioni*, la *delete list* e la *add list*.
Lo stato è rappresentato da un *insieme ordinato* di termini che rappresentano dei fatti, come *on(a,b), ontable(c), clear(a), ...* In questo modo, due stati uguali hanno esattamente la stessa rappresentazione. Le operazioni sugli insiemi ordinati sono fornite dal modulo *ordsets*.

Il predicato *applicabile(AZ, S)* verifica che le precondizioni di *AZ* siano contenute nello stato *S*. Il predicato *trasforma(AZ, S1, S2)* cancella da *S1* i fatti contenuti nella *delete list* di *AZ* e aggiunge quelli contenuti nella *add list*.

Il file *esempi blocchi* contiene alcuni esempi, in cui si fornisce uno stato iniziale ed un goal. Il goal è costituito da una lista di fatti che devono valere nello stato finale.
Il file *altri esempi blocchi* contiene altri esempi del mondo dei blocchi.

Metropolitana di Londra
Il file *metropolitana di Londra* contiene una descrizione ridotta delle linee della metropolitana nel centro di Londra. Lo stato di un agente che si muove con la metropolitana viene descritto come *[at(Stazione), Posizione]*, dove *Stazione* è il nome di una stazione della metropolitana, e *Posizione* può essere *in(Linea, Dir)*, se l'agente di trova su un treno della linea *Linea* e viaggia nella direzione *Dir*, oppure *ground*, se l'agente è a terra. La direzione di un treno può essere 0 o 1.
Le azioni possibili sono: *sali(Linea, Dir), scendi(Stazione), vai(Linea, Dir, StazionePartenza, StazioneArrivo)*.

NOTA: Nei primi due domini si assume che tutte le azioni abbiano costo unitario. Nel caso della metropolitana ha poco senso considerare azioni di costo unitario. Si chiede quindi di definire un costo per le azioni. Per questo sono fornite le coordinate di ogni stazione (l'unità di misura corrisponde a circa 625 metri).

Altri domini interessanti sono quelli della logistica (vedi capitolo 11 del Russell e Norvig o esempi CLIPS), o il puzzle dell'8, o i percorsi fra le città, ecc.

Strategie di ricerca non informate

Ricerca in profondità

La ricerca in profondità espande sempre per primo il nodo più profondo, ovvero quello più lontano dalla radice dell'albero di ricerca. Può essere facilmente realizzata in Prolog sfruttando il non determinismo del linguaggio.
Il file "ricerca in profondità" contiene la definizione della procedura *ric_prof(S, Act_list)*, dove *S* è lo stato iniziale e *Act_list* è una lista di azioni che porta dallo stato *S* ad uno stato finale. La procedura è formulata in modo non deterministico: se *S* non è uno stato finale e *Az* è una azione applicabile ad *S*, trasformare *S* in *Nuovo_S* applicando *Az*, e richiamare *ric_prof* su *Nuovo_S*. Dato che l'interprete Prolog utilizza una strategia di ricerca in profondità con *backtracking*, l'esecuzione della *ric_prof* realizza una ricerca in profondità.
Si noti il *cut* alla fine della prima regola. Supponiamo che non ci sia. Quando l'interprete arriva a chiamare la *ric_prof* con uno stato *S* finale, si ferma restituendo una soluzione. Se noi chiediamo un'altra soluzione, l'interprete passa alla seconda regola, cercando una soluzione che parte da *S*. Quindi la seconda soluzione avrebbe come stato intermedio *S*, che è uno stato finale. Viceversa, con il *cut*, quando si chiede un'altra soluzione, l'ultima chiamata di *ric_prof* fallisce e l'interprete fa *backtracking* all'ultimo punto di scelta.

Questa procedura può non terminare se l'albero di ricerca contiene cammini infiniti. Una semplice modifica consiste nel verificare che non ci siano cicli nel cammino che va dallo stato iniziale allo stato corrente. La procedura *ric_prof_cc* nel file "ricerca in profondità" ha un parametro in più che rappresenta la lista degli stati visitati lungo il cammino corrente: prima di espandere un nodo si verifica che questo non sia già stato incontrato. Questa procedura non garantisce di trovare il cammino di lunghezza minima, ma termina sempre se lo spazio di ricerca è finito.

Per trovare la soluzione di lunghezza minima, la ricerca in profondità può essere modificata come segue.

Ricerca a profondità limitata
Una procedura *ric_prof_lim*, che limita la profondità della ricerca, può essere facilmente ottenuta aggiungendo alla *ric_prof* un parametro che rappresenta il limite massimo di profondità.

Ricerca ad approfondimento iterativo (iterative deepening)
La ricerca ad approfondimento iterativo esegue ripetutamente una ricerca a profondità limitata, incrementando ad ogni passo il limite, fino a quando trova una soluzione.
Può essere facilmente implementata definendo una procedura iterativa (realizzata con la ricorsione) che ad ogni passo chiama la *ric_prof_lim* con un certo limite (inizialmente 1), e, se questa esecuzione fallisce, incrementa il limite di 1 e richiama la *ric_prof_lim*.
Questa strategia trova la soluzione ottima nel caso di azioni di costo unitario.

Strategie di ricerca non informate

Ricerca in ampiezza

Il file "ricerca in ampiezza" contiene la definizione della procedura *ric_amp(Nodi, Act_list)* che implementa la ricerca in ampiezza. Il parametro *Nodi* è una lista che rappresenta una **coda** di nodi, in cui ogni nodo è rappresentato dal termine *nodo(S, ListaAz)*, dove *S* è uno stato e *ListaAz* la lista delle azioni che porta dallo stato iniziale ad *S*. Il secondo parametro è la soluzione. Ad ogni passo, la procedura espande il nodo in testa alla coda generando tutti i suoi successori, che vengono aggiunti in fondo alla coda.
Per trovare l'insieme di tutte le azioni applicabili in un certo nodo, si usa il predicato *findall* (v. Console pag. 213)
Dato che questa strategia garantisce di trovare il cammino di lunghezza minima, non è previsto che la procedura *ric_amp* sia usata per ottenere altre soluzioni.

L'esecuzione della procedura può esaurire rapidamente la memoria, perché ogni stato può essere considerato più volte (ricerca su un albero) e quindi la coda può raggiungere dimensioni molto elevate. Per ridurre la dimensione dello spazio di ricerca, è possibile modificare la *ric_amp* in modo che effettui una ricerca su un grafo, ossia considerando ogni stato una sola volta.

Ricerca in ampiezza su grafi
Per realizzare la ricerca su grafo occorre ricordarsi i nodi già espansi (nel Russell e Norvig si parla di **lista chiusa**): prima di espandere un nodo occorre verificare che questo non sia già chiuso. Se è chiuso, non lo si espande più.
Una soluzione consiste nell'aggiungere alla *ric_amp* un parametro che rappresenta la lista degli stati appartenenti ai nodi chiusi. Un'altra possibilità è quella di asserire il fatto *chiuso(S)* ogni volta che un nodo *nodo(S, ListaAz)* è tolto dalla coda ed espanso.
Questo algoritmo realizza la cosiddetta strategia del "sasso nello stagno", ossia la coda dei nodi ancora da espandere (frontiera) contiene stati a distanza via via crescente dallo stato iniziale, e può quindi essere immaginata come un'onda che si propaga nell'acqua partendo dallo stato iniziale.

Strategie di ricerca informate

Le strategie di ricerca informate utilizzano una funzione euristica *h(n)*:

h(n) = costo stimato del cammino più conveniente dal nodo *n* ad uno nodo finale

Assumiamo inoltre che ad ogni azione sia associato un costo, e che *g(n)* sia

g(n) = costo del cammino trovato dal nodo iniziale al nodo *n*.

Le precedenti strategie di ricerca possono essere modificate per tener conto della stima e per qualunque costo delle azioni.

Ricerca in profondità ad approfondimento iterativo con stima (IDA*)

Questo algoritmo è analogo a quello basato sull'approfondimento iterativo. La differenza principale sta nel valore della soglia che ad ogni iterazione viene fissata come limite della profondità della ricerca. Al primo passo la soglia è uguale a *h(si)*, dove *si* è lo stato iniziale. Successivamente, ad ogni iterazione il nuovo valore della soglia è dato dal minimo $f(n) = g(n) + h(n)$ per tutti i nodi *n* che hanno superato il valore della soglia nell'iterazione precedente (ossia quelli in cui la ricerca fallisce e si fa backtracking).

Nella implementazione in Prolog, quando si arriva ad un punto di fallimento, occorre salvare il valore di *f(n)* prima di fare backtracking, perché altrimenti questo valore andrebbe perso. Questo può essere fatto usando una *assert*.

Ricerca in ampiezza su grafi con stima (A*)

La procedura di ricerca in ampiezza su grafi può essere modificata in modo da tener conto anche della stima (procedura A*). A differenza della ricerca non informata, in questo caso ad ogni passo si estrae dalla coda per espanderlo il nodo il cui valore $f(n) = g(n) + h(n)$ è minimo. I nodi già espansi, nodi chiusi, non vengono più espansi[1].

Questo può essere realizzato facilmente aggiungendo i campi F e G ai nodi, *nodo(F, G, S ,ListaAz),* e implementando la coda dei nodi come un *ordset*. Se F è il primo argomento di *nodo*, i nodi vengono mantenuti in ordine crescente del valore di F, e quindi il primo nodo della coda è quello che ha costo minimo.

[1] In realtà A* prevede di poter riaprire un nodo già chiuso. Nel Russell e Norvig sono descritte le condizioni sotto le quali è garantito che la procedura trova una soluzione di costo minimo senza riaprire i nodi chiusi (stima ammissibile e consistente).