Lezione 15

RICERCA II

1. L'algoritmo generico di ricerca completo

- L'algoritmo semplificato risponde solo YES o NO
- Per trovare il cammino di soluzione occorre adattarlo; bisogna memorizzare, per ogni nodo n, il cammino che arriva ad n.
- Nel caso di costo, è bene memorizzare anche il costo g(n)

1.1. I costi e i cammini

- Se vi sono dei costi, vanno associati agli archi.
- Oltre al predicato neighbors(N, L), che definisce il grafo dello spazio di ricerca, occorre specificare il predicato:

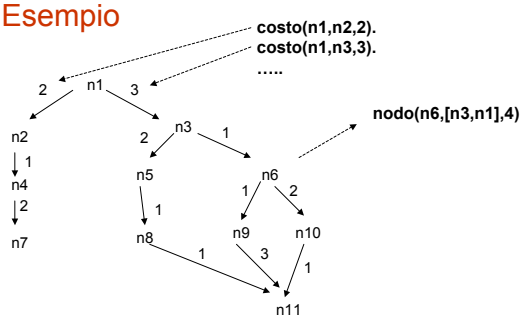
  cost(N,M,C) : C è il costo dell'arco (N,M)

- Per i cammini si usa la struttura:

  nodo(N, Cammino, Costo)

  Significato:
  [N|Cammino] è il cammino all'indietro da N alla radice e
  Costo = costo([N|Cammino]) = g(N)

Esempio

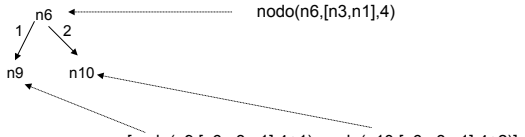


1.2. L'algoritmo generico completo psearch (path search)

psearch(F,Path): Path = cammino da un goal g ∈ G ad un nodo iniziale s ∈ S

```
psearch(F0, [N|P]) :- select(node(N,P,_),F0,_),
                      is_goal(N).
psearch(F0,Path) :-  select(node(N,P,C),F0,F1),
                     neighbors(N, NL),
                     add_paths(NL, node(N,P,C), NL1),
                     add_to_frontier(NL1,F1,F2),
                     psearch(F2,Path).
```


add_paths:



add_paths([n9,n10], nodo(n6,[n3,n1], 4),
[nodo(n9,[n6,n3,n1],5), nodo(n10,[n6,n3,n1],6)])

```
add_pats([ ], _, [ ]).

add_paths([M | R], node(N,P,C), [node(N, [N|P], NC) | RF])  :-
        cost(N, M, CN),
        NC is C + CN,
        add_paths(R, node(N,P,C), RF).
```


- L'algoritmo completo ha come predicati aperti:
  - cost spazio di ricerca
  - neighbors spazio di ricerca
  - select strategia di ricerca
  - add_to_frontiers strategia di ricerca
- Il fatto di usare l'algoritmo semplificato o quello completo non cambia nulla circa il modo di rappresentare lo spazio di ricerca e di implementare le strategie di ricerca;
- in particolare, quanto visto su depth e breadth first non cambia

Strategie di ricerca

- Vediamo alcune strategie, classificandole in base a completezza, ottimalità e complessità
  - Strategie non informate o blind (forza bruta)
    - Depth-First
    - Breadth-First
    - Lowest-Cost-First
  - Strategie informate o euristiche (funzione euristica)
    - Best-First (algoritmi gready)
    - Heuristic Depth-First
    - A*

Strategie blind

- Esaminano lo spazio di ricerca in un dato ordine (determinato dalla strategia) in modo completamente indipendente dalla natura e dalle conoscenze del problema.
- Parametri usati nella complessità:
  - b branching factor
  - m profondità massima dell'albero esaminato alla fine

Depth-First

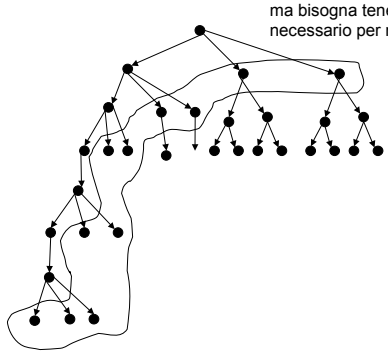
- Implementazione di select, add_to_frontiers già vista
- Completezza:
  - NO se lo spazio è infinito, SI se è finito
- Ottimalità:
  - NO
- Complessità
  - In tempo: $O(b^m)$ se termina con profondità massima m
  - In spazio: $O(b*m)$ se termina con profondità massima m

Non ottimalità:

- la prima soluzione trovata nell'ordine imposto da depth-first non è detto sia ottimale

INCOMPLETEZZA CON CAMMINI INFINITI

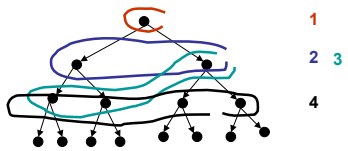
COMPLESSITA' IN SPAZIO:
lunghezza della frontiera: (b-1)*m + 1;
ma bisogna tener conto dello spazio necessario per memorizzare i cammini



Breadth-First

- Implementazione di select, add_to_frontiers già vista
- Completezza:
  - SI (anche con spazio infinito)
- Ottimalità:
  - SI a costo unitario, NO a costo non unitario
- Complessità
  - In tempo: $O(b^m)$ se m è la profondità dell'ultima frontiera
  - In spazio: $O(b^m)$ se m è la profondità dell'ultima frontiera

LUNGHEZZE DELLE FRONTIERE nel caso binario



$2^{k-1} < lung <= 2^k$

Complessità in tempo e spazio:

- Una frontiera ha profondità k-1 (prima parte della lista) o k (ultima parte); sia m la profondità dell'ultima frontiera
  - quella con la soluzione o quella massima dell'albero se è finito e non vi è soluzione
- Tempo:
  - O(dimensione dell'albero di profondità m) = $O(b^m)$
- Spazio
  - O(dimensione frontiera prof. M) = $O(b^m)$

Ottimalità/non ottimalità:

- la prima soluzione trovata si trova a profondità m-1 nella frontiera e le frontiere precedenti avevano profondità minore:
  - è con profondità minimale = costo minimale con costo 1
  - se il costo non è 1, può essere non ottimale

Lowest-Cost-First

- Se il costo di un cammino non è unitario, breadth first non è ottimale.
- Con costo non unitario, si può riadattare breadth first ordinando la frontiera un base al costo:

```
add_to_frontier(Neighbors,F1,F3) :-
      append(Neighbors,F1,F2),
      sort_by_g(F2,F3).
```

Nell'ipotesi di costo di arco >= ε > 0:

Completezza:

SI (con ε > 0 ; con ε = 0 diventa incompleto)

Complessità in tempo e spazio:

- Sia C il costo massimo dell'ultima frontiera; la profondità massima dei suoi nodi è m <= C/ε
- Tempo:
  - O(dimensione dell'albero di profondità m) = $O(b^{C/\varepsilon})$
- Spazio
  - O(dimensione frontiera prof. m) = $O(b^{C/\varepsilon})$

Ottimalità:

- SI: la prima soluzione trovata si trova a costo minimo nella frontiera e le frontiere precedenti avevano costi minori

RICERCHE INFORMATE O EURISTICHE

- Usano una funzione euristica h (da heuristic):
  - h(n) = stima della distanza (o costo) da n ad una soluzione che sia ottimale a partire da n
  - h(n) <= distanza effettiva(n) (sottostimata)
- select, append_to_frontier sono influenzate da h

Esempi di euristiche

- Ricerca del percorso più breve, ad esempio, da una città ad un'altra: si prende
  - h(n) = distanza_linea_aria(n, goal)
- Il gioco delle tessere:
  - h(n) = numero di mosse che farei se potessi spostare le tessere non al posto giusto senza vincoli (con varianti)
- I missionari e cannibali:
  - Numero di mosse non considerando il vincolo che il i cannibali non devono superare i missionari

Best-First (o gready)

```
select(N,[N|F],F).
add_to_frontier(Neighbors,F1,F3) :-
        append(Neighbors,F1,F2),
        sort_by_h(F2,F3).
```

- E' un algoritmo goloso: mangia subito la cosa che più gli piace, senza curarsi d'altro.
- Ottimalità e completezza NO, come depth first
- Complessità in tempo e spazio esponenziali, come breadth first

Heuristic Depth-First

```
select(N,[N|F],F).
add_to_frontier(Neighbors,F1,F2) :-
        sort_by_h(Neighbors,N),
        append(N,F1,F2).
```

- E' una variante golosa di depth first.
- Ha le caratteristiche (completezza, ottimalità, complessità) di depth-first, che non ripetiamo

A*

Variante euristica di Lowest-Cost-First.

Usa sia il costo calcolato g(n), sia l'euristica h(n):

f(n) = g(n) + h(n)

```
select(N,[N|F],F).
add_to_frontier(Neighbors,F1,F3) :-
        append(F1,Neighbors, F2),
        sort_by_f(F2,F3).
```

Condizioni di ammissibilità:

- h(n) è una sottostima
- b è finito
- costo arco >= ε > 0

Se valgono tali condizioni:

Completo: SI

Ottimale: SI

Complessità in spazio e tempo: esponenziali, come in lowest cost first

Dimostrazione dell'ottimalità:

- $f(n_{ott})$ <= costo(ott) < costo(nonott).
- f(nonott) = costo(nonott) perché è una soluzione
  - f(nonott) = g(nonott) + 0 (h(nonott)=0)
- Assumiamo per assurdo che sia selezionata nonott; la frontiera conterrebbe ancora un nodo n_{ott} che porta ad ott; assurdo perché la frontiera è ordinata secondo f, mentre $f(n_{ott})$ < costo(nonott).

Iterative deepening

- Si procede seguendo la strategia depth first, ma si pota l'albero ad una profondità k;
- facendo variare k a partire da 1, con incrementi successivi, sino a trovare la prima soluzione, si ottiene un comportamento sostanzialmente simile alla ricerca in larghezza:
  - ad ogni k, viene visitata la frontiera di profondità k
  - incrementando k ogni volta, se non si è trovata una soluzione, la prima soluzione trovata sarà ottimale
  - è chiaro che, se una soluzione esiste, viene trovata

- Completezza: SI
- Ottimalità: SI
- Complessità:
  - in spazio: quella di depth first, essendo questa la strategia adottata
  - in tempo: pur ripetendo ogni volta la visita dell'albero per le profondità minori, rimane $O(b^k)$
    - la frontiera di profondità k viene generata con un fattore moltiplicativo pari a $(b / (b-1))^2$
      - con b = 2 il fattore è 4, con b=3 è 2,25, ecc.

Considerazioni finali

- L'uso di PROLOG: ci è servito per dare in modo compatto l'algoritmo generico; ma
  - Non abbiamo scelto le implementazioni migliori per lavorare con le frontiere ordinate: vedere esercizio 4.9 sul libro
  - L'interprete Prolog è in generale lento (rispetto a linguaggi tipo C) e non si ha un controllo diretto dell'occupazione di memoria (come con un uso appropriato dello Heap con i puntatori in C)
- Dunque Prolog è stato usato più come un linguaggio di spiegazione che di implementazione

- Si può usare convenientemente Prolog quando si voglia sfruttare il backtracking in esso implicito; sostanzialmente, prolog sviluppa la ricerca depth-first, nell'albero in cui i rami sono dati dalla applicazione delle clausole nel programma, nell'ordine in cui esse si trovano nello stesso.
- Un esempio di uso del Prolog per realizzare una ricerca è l'esercizio del programma che esegue la procedura top-down; per evitare loop, si può aggiungere un limite sulla profondità ed usare l'iterative deepening.
- **Vedere programma di esercizio relativo all'esercizio 4.9 sulle code di priorità**