Strategie di ricerca

Esercitazione del 13 Maggio 2004

- *Scopo:*
  1. Assestare la comprensione delle strategie di ricerca viste a lezione
  2. Imparare ad utilizzare la libreria aima.search, che vi potrà essere utile per eventuali tesine

- *Com'è organizzata l'esercitazione:*
  1. Velocissimo ripasso delle strategie
  2. Un po' di osservazioni su come rappresentare lo stato
  3. Veloce introduzione alla libreria aima.search
  4. Esempio di utilizzo
  5. Implementazione di strategie di ricerca
  6. Utilizzo delle strategie su un problema determinato

CERCARE SOLUZIONI

Generare sequenze di azioni.

- *Espansione:* si parte da uno stato e applicando gli operatori (o la funzione successore) si generano nuovi stati.

- *Strategia di ricerca:* ad ogni passo scegliere quale stato espandere.

- *Albero di ricerca:* rappresenta l'espansione degli stati a partire dallo stato iniziale (la radice dell'albero).

- Le foglie dell'albero rappresentano gli stati da espandere.

STRATEGIE DI RICERCA

- STRATEGIE DI RICERCA **NON-INFORMATE**:
  - breadth-first (a costo uniforme);
  - depth-first;
  - depth-first a profondità limitata;
  - ad approfondimento iterativo.

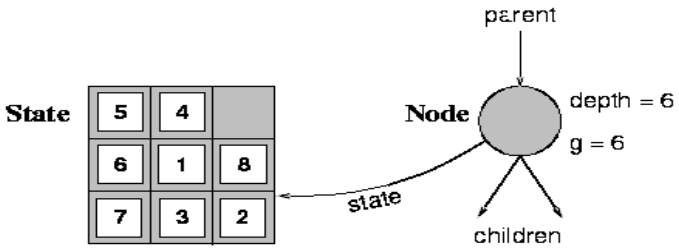
STRATEGIE DI RICERCA

- STRATEGIE DI RICERCA **INFORMATE**:
  - Best first
    1. Greedy
    2. A*
    3. IDA*
    4. SMA*

IDA* e SMA* non sono stati visti a lezione… li potete trovare sul Russell-Norvig nel capitolo dedicato alle ricerche informate…

Strutture dati per l' albero di ricerca (struttura di un nodo)

- Lo stato nello spazio degli stati a cui il nodo corrisponde.
- Il nodo genitore.
- L'operatore che è stato applicato per ottenere il nodo.
- La profondità del nodo.
- Il costo del cammino dallo stato iniziale al nodo



L'algoritmo generale di ricerca

function GENERAL-SEARCH(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

L'algoritmo generale di ricerca

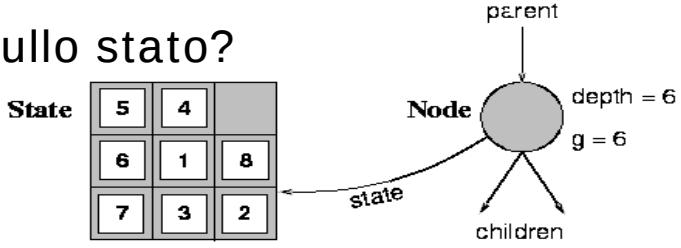
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
  loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
  end
```

Tramite l'argomento **Queuing-Fn** viene passata una funzione per accodare i nodi ottenuti dall'espansione

Primo passo: Definizione del problema

- Come rappresento un problema?
  1. In generale utilizzo una rappresentazione a stati (ho quindi degli operatori che mi permettono di operare sugli stati)
  2. Ho uno stato iniziale
  3. Ho un goal da soddisfare
- Come rappresento uno stato?
  - Strutture dati che rappresentano lo stato
- E gli operatori sullo stato?



Primo passo: Definizione del problema

- Usando un approccio “object-oriented”, rappresento uno stato tramite una classe
- Gli operatori sullo stato vengono rappresentati tramite metodi della classe stessa
- Allora, quali metodi?
  1. boolean isGoal() → mi dice se ho raggiunto il goal
  2. lista_di_successori successors() → mi restituisce chi sono i possibili successori di questo stato applicando tutti gli operatori applicabili…
- Implementato in questa versione con l’interfaccia **`aima.search.State.java`**

…operatori sullo stato… bastano questi?

- Con questi metodi, quali strategie posso applicare?
  1. Breadth-first
  2. Depth-first
  3. Depth-bounded
  4. Iterated Deepening
- Ma se non tengo traccia della “strada” percorsa per giungere alla soluzione, posso solo dire che esiste una soluzione ma non posso dire come generarla
  - Es:nel gioco del filetto, so che c’è una soluzione, ma se non conosco le mosse per giungervi…

… cosa manca?

- Se voglio sapere come giungere ad una certa soluzione, devo sapere quali sono gli operatori che sono stati applicati
- Inoltre, se voglio sapere il costo per giungere a tale soluzione, devo anche conoscere il costo degli operatori applicati per giungervi
- → oltre allo stato, devo tenere traccia anche di come ci sono arrivato, e di quanto mi è costato passare dallo stato precedente a quello attuale

… introduciamo il concetto di "successore"

- Il successore è una struttura dati che tiene traccia di:
  1. Lo stato
  2. L'operatore applicato per giungere in tale stato
  3. Il costo di tale operatore
- La libreria **`aima.search`** offre già la classe **`Successor.java`**
- Attenzione! Non confondete il concetto di successore con un nodo dell'albero di ricerca…

Secondo passo: l'interfaccia

`aima.search.Traversable`

- La sola interfaccia State non è più sufficiente… dobbiamo conoscere esplicitamente quali sono gli operatori e qual è il loro costo → Traversable
- Definisce appunto tre metodi, che si applicano ad un certo stato:
  1. Quali sono gli operatori che posso applicare per generare nuovi stati… → `validOperators()`
  2. Qual è il costo se applico un certo operatore… →

     `float costOf(String op)`
  3. Una funzione che riceva come argomento l'operatore che voglio applicare, e mi restituisca il nuovo successore → `applyOperator(String op)`

Riepilogo: fin qui abbiamo…

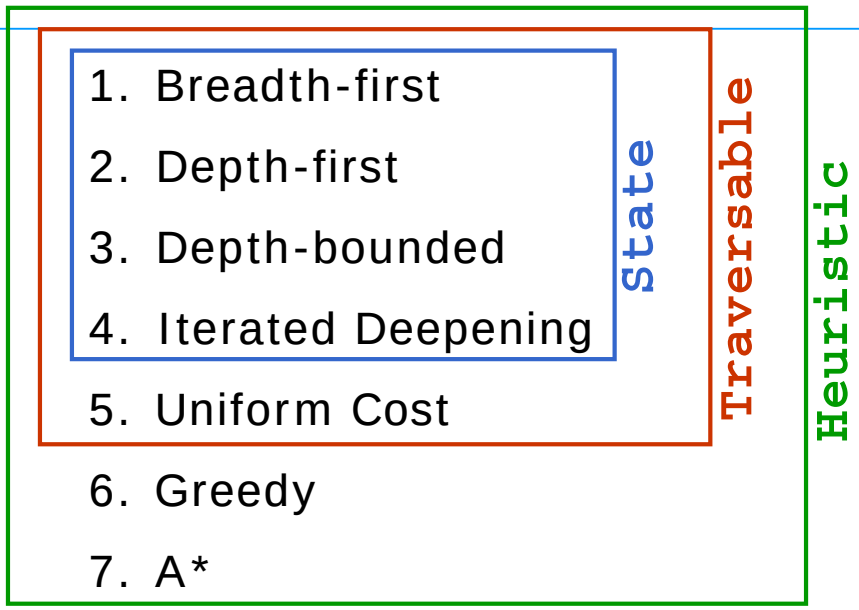
- Dunque se costruisco una classe java che mi rappresenta uno stato, e che implementa rispettivamente le interfacce `State` e `Traversable`, allora posso applicare i seguenti metodi di ricerca:
  1. Breadth-first
  2. Depth-first
  3. Depth-bounded
  4. Iterated Deepening
  5. Uniform Cost

Terzo passo: l'interfaccia

`aima.search.Heuristic`

- E se voglio applicare delle strategie informate?
- Ricordiamoci la definizione di euristica: una funzione che mi restituisce una stima (più o meno esatta) di quanto mi costa giungere fino al goal a partire da un certo stato…
- L'interfaccia **`Heuristic`** definisce quindi soltanto un nuovo metodo, `float h()`
- Ricordiamoci che poi, a seconda della strategia, si può scegliere di considerare soltanto l'euristica, o una qualche funzione più elaborata…

Riepilogo: come implementare uno stato



N.B.: Applicare strategie intelligenti ha un costo in termini di "conoscenza" che devo fornire…

Com'è costruita la libreria `aima.search` (`aima.cs.util`)

- Fornisce direttamente l'implementazione dei nodi di un albero di ricerca, tramite le classi `SearchNode` (per ricerche non informate) ed `HeuristicSearchNode` (per ricerche informate).
- Entrambe fanno riferimento alla classe `Successor`, che a sua volta contiene internamente la classe `MyState` che andrete a definire voi.
- Implementa anche l'algoritmo "Generalized Search", tramite la classe `GeneralQueueSearch`
- Fornisce anche l'implementazione di diverse "liste"

L'algoritmo generale di ricerca

function GENERAL-SEARCH(*problem*, QUEUING-FN) **returns** a solution, or failure

```
nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
loop do
    if nodes is empty then return failure
    node ← REMOVE-FRONT(nodes)
    if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
    nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
end
```

Tramite l'argomento **Queuing-Fn** viene passata una funzione per accodare i nodi ottenuti dall'espansione

Come implementare una strategia di ricerca utilizzando la libreria `aima.search`

- La classe **`GeneralQueueSearch`** offre già tutto ciò di cui c'è bisogno. Tale classe implementa proprio l'algoritmo presentato nella slide precedente.
- Riceve come parametro d'ingresso nel costruttore:
  1. Una istanza dello stato iniziale del problema
  2. Una coda il cui inserimento è soggetto al tipo di coda che si sta utilizzando (fifo, lifo, a priorità)
- Si invoca quindi il metodo search(), e questo applica l'algoritmo visto fino a trovare una soluzione

Come usare la libreria `aima.search` `(aima.cs.util)`

- Si comincia con il scegliere una possibile strategia di ricerca per il problema che si vuole risolvere
- Si realizza una classe che implementi tale strategia di ricerca
- Si costruisce una classe per rappresentare lo stato del problema scelto
- Si testa il tutto tramite il codice già fornito in esempio

Esempio

- Il problema del “23”
- Il mio stato è rappresentato da un numero intero
- Le operazioni ammesse sono add2 (con costo 2) e add3 (con costo 4)
- Lo stato iniziale è 0
- Problema: quale sequenza di operatori devo applicare per poter avere stato che sia multiplo di 23, partendo dallo stato iniziale 0?
- Goal: soddisfatto se (stato % 23) == 0

Un primo problema semplice: Missionari e Cannibali

ESEMPIO:

- 3 missionari e 3 cannibali devono attraversare un fiume. C'è una sola barca che può contenere al massimo due persone. Per evitare di essere mangiati i missionari non devono mai essere meno dei cannibali sulla stessa sponda (stati di fallimento).
- Stato: sequenza ordinata di tre numeri che rappresentano il numero di missionari, cannibali e barche sulla sponda del fiume da cui sono partiti.
- Perciò lo stato iniziale è: (3,3,1) (nota l’importanza dell’astrazione).

Un primo problema semplice: Missionari e Cannibali

- Operatori: gli operatori devono portare in barca
  - 1 missionario, 1 cannibale,
  - 2 missionari,
  - 2 cannibali,
  - 1 missionario
  - 1 cannibale.
- Al più 5 operatori (grazie all'astrazione sullo stato scelta).
- Test Obiettivo: Stato finale (0,0,0)
- Costo di cammino: numero di traversate.

Un secondo problema : Quadrato Magico

- E' dato un quadrato di 10 caselle per 10 (in totale 100 caselle).
- Nello stato iniziale tutte le caselle sono vuote tranne la più in alto a sinistra che contiene il valore 1.
- Problema: assegnare a tutte le caselle un numero consecutivo, a partire da 1, fino a 100, secondo le seguenti regole:
- A partire da una casella con valore assegnato x, si può assegnare il valore (x+1) solo ad una casella vuota che dista 2 caselle sia in verticale, che orizzontale, oppure 1 casella in diagonale.

Un secondo problema : Quadrato Magico

- Se il quadrato è ancora vuoto, per una casella generica ci sono 7 possibili caselle vuote su dove andare
- Perché le caselle sono 7 e non 8 (4 in diagonale e 4 in orizzontale/verticale) ?

- Man mano che si riempe il quadrato, le caselle libere diminuiscono → diminuisce il fattore di ramificazione

- La profondità dell'albero è 100 (dobbiamo assegnare 100 numeri – 99 se il primo è già stato assegnato)