Strategie di ricerca

- *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. Introduzione alla libreria aima.search
  4. Esempio di utilizzo

CERCARE SOLUZIONI

Alcuni concetti:

- *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.

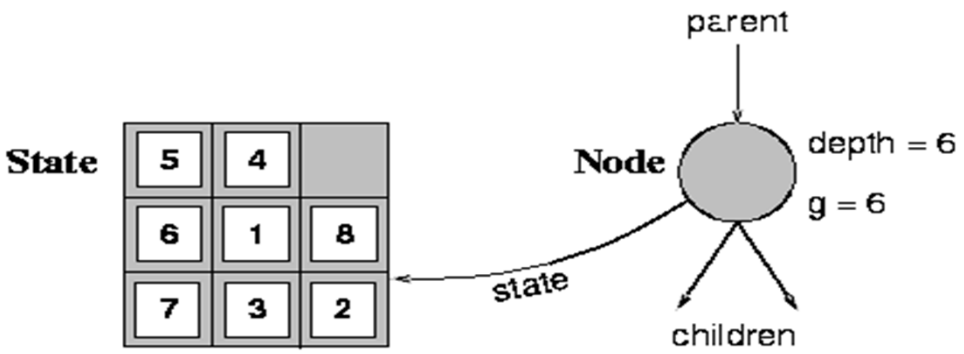
STRATEGIE DI RICERCA

- 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

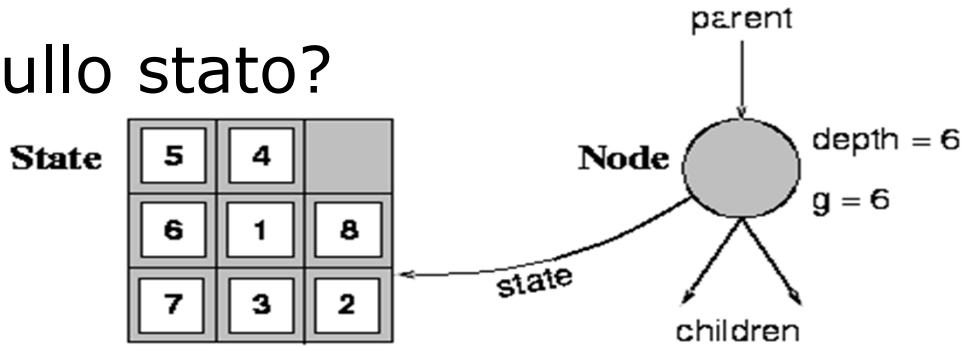
```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

  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
- E poi, quali altri metodi?
  - boolean **`isGoalTest(…)`** → mi dice se ho raggiunto il goal (metodo definito dall'interfaccia **`aima.search.framework.GoalTest`**)

La libreria **`aima`** non pone nessun vincolo su come rappresentare lo stato: richiede solo che sia una istanza di **`java.lang.Object`**

…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…

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

- La libreria **`aima.core.search.framework`** offre già la classe **`Successor.java`**
- lista_di_successori **`getSuccessors()`** → restituisce chi sono i possibili successori di questo stato applicando tutti gli operatori applicabili (interfaccia **`aima.search.framework.SuccessorFunction`**)
- Attenzione! Non confondete il concetto di successore con un nodo dell'albero di ricerca...

Riepilogo: fin qui abbiamo…

Dunque se costruisco una classe java che mi rappresenta uno stato, e che implementa rispettivamente le interfacce `GoalTest` e `SuccessorFunction`, allora posso applicare i seguenti metodi di ricerca:

1. Breadth-first
2. Depth-first
3. Depth-bounded
4. Iterated Deepening

E la strategia a Costo Uniforme?

Se voglio sapere il costo per giungere a tale soluzione, devo anche conoscere il **costo degli operatori** applicati per giungervi…

A tal scopo è specificata l'interfaccia

`aima.search.framework.StepCostFunction`

con il metodo:

`calculateStepCost(…)`

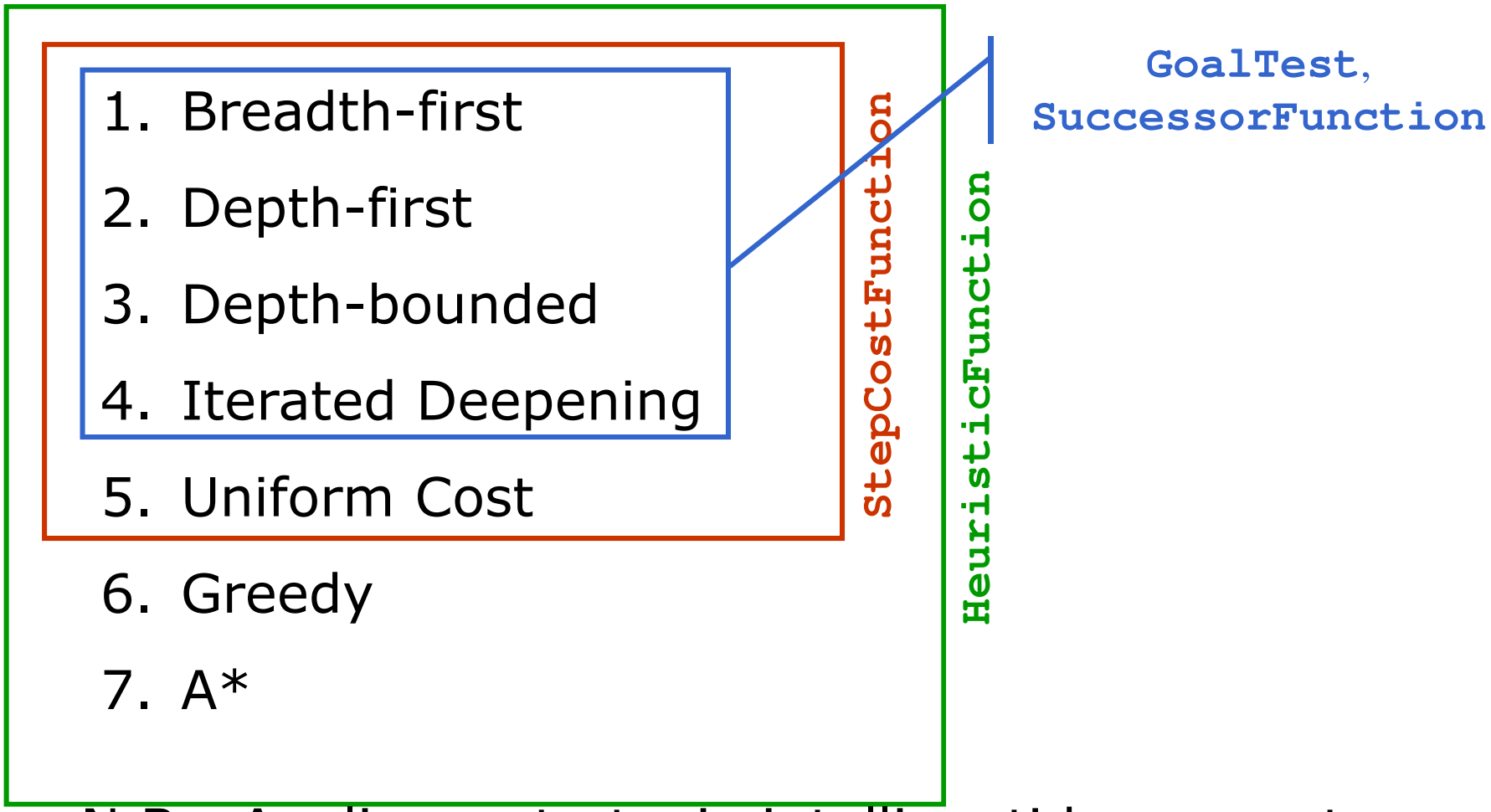
Strategie Informate

- 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 `aima.search.framework.HeuristicFunction` definisce quindi soltanto un nuovo metodo

```
double getHeuristicValue(Object state);
```

- 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`

Fornisce la classe:

- **`aima.search.framework.Problem`**,

rappresentante il problema, a cui è possibile fornire lo stato iniziale (una qualunque istanza di **`java.lang.Object`**), e le implementazioni delle interfacce (secondo necessità):

- **`GoalTest`**
- **`SuccessorFunction`**
- **`StepCostFunction`**
- **`HeuristicFunction`**

Com'è costruita la libreria
`aima.search`

Fornisce direttamente l'implementazione dei nodi di un albero (grafo) di ricerca, tramite le classi:

- **`aima.search.framework.Node`**,
- **`aima.search.framework.QueueSearch`**, che implementa l'algoritmo GeneralSearch, con le sottoclassi:
  - **`BreadthFirstSearch`**
  - **`DepthFirstSearch`**
  - **`DepthLimitedSearch`**
  - **`IterativeDeepeningSearch`**
  - **`GreedyBestFirstSearch`**
  - **`SimulatedAnnealingSearch`**
  - **`HillClimbingSearch`**
  - **`AStarSearch`**

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

Com’è costruita la libreria
aima.search

Per attuare le varie strategie, la libreria aima.search utilizza diverse queuening function:

- **aima.search.datastructures.FIFOQueue**
- **aima.search.datastructures.LIFOQueue**
- **aima.search.datastructures.PriorityQueue**

Com’è costruita la libreria

`aima.search`

Alla classe QueueSearch (cioè alle sue sottoclassi) è possibile specificare anche se si sta effettuando la ricerca su un albero o su un grafo (controlla di non ripassare per uno stato già visitato). A tal scopo fornisce anche le classi:

- **`aima.search.framework.TreeSearch`**
- **`aima.search.framework.GraphSearch`**

Com'è costruita la libreria

`aima.search`

Viene fornita infine la classe:

`aima.search.framework.SearchAgent`

che provvede direttamente a cercare la soluzione nello spazio degli stati.

Riceve come parametri di ingresso un problema (istanza della classe **`Problem`**), ed una strategia di ricerca (istanza dell'interfaccia **`Search`**).

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.

Esempio

```
MCState initState = new MCState();
Problem problem = new Problem(
      initState,
      new MCSuccessorFunction(),
      new MCGoalTest());

Search search =
new BreadthFirstSearch(new TreeSearch());

SearchAgent agent =
new SearchAgent(problem, search);


// semplici metodi di stampa dei risultati…
printActions(agent.getActions());
printInstrumentation(agent.getInstrumentation());
```

Un secondo problema : Riempimento di Quadrato

- 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 : Riempimento di un Quadrato

- 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)

Un terzo problema :
Il ponte degli U2 (compito d'esame del 16 Dicembre 2005)

- Il complesso degli U2 sta per fare un concerto a Dublino.
- Mancano 17 minuti all'inizio del concerto ma, per raggiungere il palco, i membri del gruppo devono attraversare un piccolo ponte che è tutto al buio, disponendo di una sola torcia elettrica. Sul ponte non possono andare più di due persone per volta. La torcia è essenziale per l'attraversamento, per cui deve essere portava avanti e indietro (non può essere lanciata da una parte all'altra) per consentire a tutti di passare. Tutti sono dalla stessa parte del ponte.
- Ciascun componente del complesso cammina a una velocità diversa. I tempi individuali per attraversare il ponte sono:
- Bono, 1 minuto
- Edge, 2 minuti
- Adam, 5 minuti
- Larry, 10 minuti

Un terzo problema :
Il ponte degli U2 (compito d'esame del 16 Dicembre 2005)

- Se attraversano in due, la coppia camminerà alla velocità del più lento.
- Ad esempio: se Bono e Larry attraversano per primi, quando arrivano dall'altra parte saranno trascorsi 10 minuti. Se Larry torna indietro con la torcia saranno passati altri dieci minuti, per cui la missione sarà fallita.
- Si stabilisca una funzione euristica ammissibile per questo problema e lo si risolva tramite l'algoritmo A* per i grafi, mostrando il grafo generato. Per limitare la dimensione dello spazio di ricerca, si supponga che i componenti si muovano sempre in 2 in una direzione e sempre in 1 nella direzione opposta (quando devono riportare indietro la torcia per prendere gli altri componenti della band).

Un terzo problema :
Il ponte degli U2 (compito d'esame del 16 Dicembre 2005)

- Per definire la funzione euristica *h'* si usi la seguente idea:
- Possono spostarsi solo 2 persone alla volta, per cui raggruppiamo le persone sulla riva di partenza (sinistra) in gruppi da 2. Ordino i tempi di percorrenza dal più alto al più basso e metto
- nel primo gruppo i primi due
- nel secondo gruppo il terzo ed il quarto.
- Ciascun gruppo si muoverà alla velocità del più lento, per cui per ogni gruppo prendo il massimo. La funzione *h'* è data dalla somma dei tempi di percorrenza di ciascun gruppo. In altre parole, il tempo di percorrenza totale sarà almeno il tempo di percorrenza del più lento, più quello del terzo più lento (il secondo più lento potrebbe essersi mosso con il più lento, per cui il suo tempo non viene calcolato).
- Nell'algoritmo A* per i grafi si considerano cammini alternativi per arrivare ad uno stesso stato; in caso di cammini alternativi viene tenuto il costo minore. Rappresentiamo con una freccia piena il cammino migliore e con una freccia tratteggiata i cammini peggiori.
- La funzione *h'* cosi' definita e' ammissibile? L'algoritmo trovera` la strada migliore?

In laboratorio

1. Lanciare Eclipse e creare un nuovo progetto
2. Scaricare dalla home page del corso il file aima.0.93,jar ed includerlo nelle librerie del progetto
3. Creare nuove classi al fine di risolvere uno dei problemi proposti
   - Missionari e Cannibali
   - U2
   - Riempimento di un quadrato 10x10