Algoritmi di ricerca informati

Capitolo 4, sezioni 1–2, 4

Sommario

- ♢ Ricerca best-first
- ♢ Ricerca greedy, Ricerca A*
- ♢ Hill-climbing
- ♢ Simulated annealing

Ripasso: ricerca generale

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

Una strategia è definita scegliendo *l'ordine di espansione dei nodi*

Ricerca Best-first

Idea: usare una *funzione di valutazione* per ogni nodo
– stima di "desiderabilità"

⇒ Espandere il nodo non espanso più desiderabile

Implementazione:
QUEUEINGFN = inserire i successori in ordine decrescente di desiderabilità

Casi speciali:
ricerca greedy
ricerca A*

Ricerca greedy

Funzione di valutazione $h(n)$ (euristica)
$=$ stima del costo dal nodo n al $goal$

E.g., $h_{\mathrm{SLD}}(n) =$ distanza in linea d'aria da n a Bucharest

La ricerca greedy espande il nodo che $appare$ essere il più vicino al goal

Ricerca A*

Idea: evitare di espandere cammini che sono gia' costosi

Funzione di valutazione $f(n) = g(n) + h(n)$

$g(n) =$ costo gia' avuto per raggiungere n
$h(n) =$ costo stimato da n al goal
$f(n) =$ costo totale stimato del cammino attraverso n al goal

La ricerca A* usa un'euristica $ammissibile$
cioe': $h(n) \leq h^*(n)$ dove $h^*(n)$ e' il costo $vero$ da n.

Es.: $h_{\mathrm{SLD}}(n)$ non sovrastima mai la reale distanza

Teorema: la ricerca A* e' ottima

Algoritmi di miglioramento iterativo

In molti problemi di ottimizzazione, il *cammino* e' irrilevante;
la soluzione e' lo stato di goal stesso

Allora lo spazio degli stati e' l'insieme delle configurazioni "complete";
trova una configurazione *ottima*, es.: TSP
o trova una configurazione che soddisfa dei vincoli, es.: n-regine

In tali casi, si possono usare gli algoritmi di *miglioramento iterativo*;
Mantiene un singolo stato corrente, e tenta di migliorarlo

Spazio costante

Hill-climbing (o discesa/ascesa di gradiente)

```
function HILL-CLIMBING( problem) returns a solution state
    inputs: problem, a problem
    local variables: current, a node
                     next, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        next ← a highest-valued successor of current
        if VALUE[next] < VALUE[current] then return current
        current ← next
    end
```

Simulated annealing

Idea: evitare i massimi locali permettendo delle mosse cattive
ma gradualmente decrementare la loro grandezza e frequenza

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"
  local variables: *current*, a node
    next, a node
    T, a "temperature" controlling the probability of downward steps

  current $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  for $t \leftarrow 1$ **to** ∞ **do**
    $T \leftarrow$ *schedule*[t]
    if T=0 **then return** *current*
    next $\leftarrow$ a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    if $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
    else *current* $\leftarrow$ *next* only with probability $e^{\Delta E/T}$

Se T e' alta, le mosse "cattive" sono piu' probabili.
T scende a mano a mano che il tempo passa, secondo schedule.