Intelligenza Artificiale

Riassunto breve

Indice

1. Algoritmi di ricerca

1.1. Ricerca non informata

1.1.1. Ricerca in ampiezza

Utile per: problemi in cui tutte le azioni hanno lo stesso costo.

Pseudocodice:

```
function Ricerca-In-Ampiezza(problema) returns un nodo soluzione o fallimento
  nodo ← NODO(problema.StatoIniziale)
  if nodo.Stato == problema.Obiettivo then return nodo
  frontiera ← una coda FIFO, con nodo come elemento iniziale
  raggiunti ← {problema.StatoIniziale}
  while not Vuota?(frontiera) do
    nodo ← Pop(frontiera)
    for each figlio in Espandi(problema, nodo) do
      s ← figlio.Stato
      if s == problema.Obiettivo then return figlio
      if s not in raggiunti then
        add s a raggiunti
        add figlio a frontiera
  return fallimento
```

Completezza e ottimalità: completa e ottimale.

- *completa* perché se c'è una soluzione, viene sempre trovata
- *ottimale* perché, tra più soluzioni, viene trovata quella a profondità minore (cioè anche a costo minimo)

Complessità: $O\left(b^d\right)$ (esponenziale) sia in termini di spazio che di tempo, dove b è il fattore di ramificazione dell'albero degli stati (b figli per ogni nodo) e d è la profondità della soluzione.

1.1.2. Ricerca a costo uniforme (*best-first* / *Dijkstra*)

Utile per: problemi in cui il costo di ogni azione è diverso.

Usa una metrica f che rappresenta il costo del cammino dal nodo radice fino al nodo considerato.

Pseudocodice:

```
function Ricerca-Costo-Uniforme(problema) returns un nodo soluzione o fallimento
  return Ricerca-Best-First(problema, Costo-Cammino)
```

dove `Ricerca-Best-First` è strutturata come segue:

```
function Ricerca-Best-First(problema, f ) returns un nodo soluzione o fallimento
  nodo ← Nodo(Stato=problema.StatoIniziale)
  frontiera ← una coda con priorità ordinata in base a f, con nodo come elemento
iniziale
  raggiunti ← una tabella di lookup, con un elemento con chiave problema.StatoIniziale e
valore nodo
  while not Vuota?(frontiera) do
    nodo ← Pop(frontiera)
    if nodo.Stato == problema.Obiettivo then return nodo
    for each figlio in Espandi(problema, nodo) do
      s ← figlio.Stato
      if s not in raggiunti or figlio.Costo-Cammino < raggiunti[s].Costo-Cammino then
        raggiunti[s] ← figlio
        aggiungi figlio a frontiera
  return fallimento
```

Completezza e ottimalità: completo e ottimale, ma solo in certe condizioni.

- sempre completo
- ottimale solo se la funzione f è non decrescente con la profondità

Complessità: variabile perché dipendente dal costo dei cammini e non dalla profondità. Potenzialmente maggiore di $O(b^d)$ a seconda dei casi.

1.1.3. Ricerca in profondità

Utile per: situazioni in cui la ricerca in ampiezza ha un costo eccessivo.

Pseudocodice:

Non disponibile. La ricerca parte dalla radice e si espande in profondità verso i nodi figli. In caso ce ne sia più di uno, si sceglie arbitrariamente un ordine da seguire. L'algoritmo risale e continua da altri nodi figli se nella discesa giunge ad un nodo foglia non soluzione. L'algoritmo termina se trova una soluzione o se esaurisce tutti i nodi.

Solitamente implementata come ricerca su albero (non grafo) senza memoria degli stati raggiunti.

Completezza e ottimalità: incompleto e non ottimale.

- incompleto perché in spazi infiniti l'algoritmo si può «incastrare» in una singola discesa senza esplorare mai gli altri percorsi
- non ottimale perché non è in grado di distinguere il costo delle soluzioni e si ferma alla prima trovata

Complessità: $O(b^m)$ nel tempo, $O(b \cdot m)$ nello spazio, dove b è il fattore di ramificazione dell'albero e m è la sua massima profondità.

1.1.4. Ricerca a profondità limitata

Variante della ricerca in profondità dove si impone una profondità massima L.

Pseudocodice:

```
function Ricerca-Profondità-Limitata(problema, L) returns un nodo soluzione o fallimento
o soglia
  frontiera ← una coda LIFO con Nodo(problema.StatoIniziale) come elemento iniziale
  risultato ← fallimento
  while not Vuota?(frontiera) do
    nodo ← Pop(frontiera)
    if nodo.Stato == problema.Obiettivo then return nodo
    if Profondità(nodo) > L then
      risultato ← soglia
    else if not È-Ciclo(nodo) do
      for each figlio in Espandi(problema, nodo) do
        aggiungi figlio a frontiera
  return risultato
```

Completezza e ottimalità: potenzialmente incompleto se L non è scelto correttamente.

Mantenendo una memoria anche limitata dei nodi visitati, è possibile limitare il rischio di cicli e ricerche ridondanti.

Complessità: $O(b^L)$ nel tempo, $O(b \cdot L)$ nello spazio.

1.1.5. Ricerca ad approfondimento iterativo

Utile per: situazioni in cui si voglia automatizzare la determinazione di L nella ricerca a profondità limitata

Pseudocodice:

```
function Ricerca-Approfondimento-Iterativo(problema) returns un nodo soluzione o
fallimento
  for profondità = 0 to ∞ do
    risultato ← Ricerca-Profondità-Limitata(problema, profondità)
    if risultato ≠ soglia then return risultato
      # se risultato è diverso da soglia allora è la soluzione oppure fallimento
```

Completezza e ottimalità: potenzialmente sia completo che ottimale.

- *completo* se lo spazio degli stati è finito e aciclico
- *ottimale* se le azioni hanno tutte lo stesso costo

Complessità:

- $O\left(b^d\right)$ se c'è una soluzione a profondità d o $O(b^m)$ nel tempo per il caso peggiore (se non esistono soluzioni)
- $O(b \cdot d)$ o $O(b \cdot m)$ nello spazio a seconda del caso

1.2. Ricerca informata

Differenza con la ricerca non informata: invece di espandere sistematicamente gli stati, gli algoritmi di ricerca informata si lasciano guidare da un'euristica per ridurre il costo della ricerca.

Le euristiche sono specifiche per il problema considerato e sono difficilmente generalizzabili.

1.2.1. Ricerca *best-first greedy*

Utile per: situazion in cui è disponibile una metrica di distanza dall'obiettivo.

Pseudocodice:

```
function Ricerca-Best-First-Greedy(problema) returns un nodo soluzione o fallimento
  return Ricerca-Best-First(problema, h(n))
```

dove $h(n)$ è la funzione che rappresenta il valore dell'euristica per il nodo n.

Completezza e ottimalità: incompleta e non ottimale in generale.

Ha gli stessi problemi della ricerca in profondità. Diventa completa in spazi degli stati finiti.

Complessità: $O(b^d)$ nel tempo nel caso peggiore, $O(b^d)$ nello spazio.

1.2.2. Ricerca A^*

Utile per: contesti simili a quelli della *best-first-greedy*.

Oltre alla distanza dall'obiettivo, considera anche la lunghezza del cammino dalla radice.

Pseudocodice:

```
function Ricerca-Best-First-Greedy(problema) returns un nodo soluzione o fallimento
  return Ricerca-Best-First(problema, f(n))
```

$f(n)$ è un'euristica combinata:

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

dove $g(n)$ è la lunghezza del cammino dal nodo iniziale al nodo n e $h(n)$ è l'euristica dell'algoritmo *best-first greedy*.

Completezza e ottimalità: completa in stati degli spazi finiti. L'ottimalità dipende dalle condizioni.

A^* è ottimale per euristiche *ammissibili*.

Data $h(n)$ la vera distanza tra il nodo n e l'obiettivo, un'euristica $h'(n)$ è detta ammissibile se

$$\forall n h'(n) \leq h(n)$$

ovvero se l'euristica è *ottimistica*.

Complessità: $O(b^d)$ sia nello spazio che nel tempo nel caso peggiore, con b fattore di ramificazione e d profondità.

La complessità può essere ridotta rinunciando all'ottimalità.

1.3. Ricerca in ambienti complessi

1.3.1. Hill-climbing

Utile per: situazioni di ricerca locale.

Pseudocodice:

```
function Hill-Climbing(problema) returns uno stato che è un massimo locale
  corrente ← problema.StatoIniziale
  while true do
    vicino ← lo stato successore di corrente con valore più alto
    if Valore(vicino) ≤ Valore(corrente) then return corrente
    corrente ← vicino
```

1.3.2. Simulated annealing

Utile per: ridurre il rischio di convergenza ad un *plateau* locale nell'hill-climbing.

Usa una temperatura T, decrescente con le iterazioni, che rappresenta l'entità di uno spostamento stocastico che altera il movimento per gradiente.

Pseudocodice:

```
function Simulated-Annealing(problema, velocità_raffreddamento) returns uno stato
soluzione
  corrente ← problema.StatoIniziale
  for t ← 1 to ∞ do
    T ← velocità_raffreddamento[t]
    if T = 0 then return corrente
    successivo ← un successore di corrente scelto a caso
    ΔE ← Valore(corrente) − Valore(successivo)
    if ΔE > 0 then corrente ← successivo
    else corrente ← successivo solo con probabilità e^(ΔE/T)
```

L'input `velocità_raffreddamento` determina il valore della temperatura T in funzione del tempo.

1.3.3. Algoritmi genetici

Utile per: situazioni complesse in cui è difficile procedere in modo deterministico.

Una serie di possibili soluzioni, inizializzate casualmente, vengono fatte competere e riprodurre in modo simile all'evoluzione degli organismi viventi.

Ogni soluzione è rappresentata come una stringa di testo per facilitare il processo di riproduzione.

Pseudocodice:

```
function Algoritmo-Genetico(popolazione, fitness) returns un individuo
  repeat
    pesi ← Pesato-Da(popolazione, fitness)
    popolazione2 ← lista vuota
    for i = 1 to Dimensione(popolazione) do
      genitore1, genitore2 ← Selezione-Casuale-Pesata(popolazione, pesi, 2)
      figlio ← Riproduzione(genitore1, genitore2)
      if (piccola probabilità casuale) then figlio ← Mutazione(figlio)
      aggiungi figlio a popolazione2
      popolazione ← popolazione2
  until esiste in popolazione un individuo con fitness sufficientemente alto, o è
passato
    abbastanza tempo
  return l'individuo migliore nella popolazione in base alla fitness
```

dove

```
function Riproduzione(genitore1, genitore2) returns un individuo
n ← Lunghezza(genitore1)
c ← numero casuale fra 1 e n
return Concatena(Sottostringa(genitore1, 1, c), Sottostringa(genitore2, c + 1, n))
```

Le stringhe che rappresentano i due genitori sono ricombinate ad ottenere un figlio. Il punto di *cross-over* c viene scelto casualmente.

2. Ottimizzazione

CSP: problemi di soddisfacimento di vincoli.

Si utilizza un sistema di ricerca negli stati, ma con euristiche generali.

2.1. AC-3

Utile per: controllare la consistenza d'arco dei vincoli in un CSP.

Pseudocodice:

```
function AC-3(csp) returns false se viene trovata una inconsistenza, o altrimenti true
  Arc ← un insieme di archi orientati, inizialmente tutti quelli nel csp
  while Arc non è vuota do
    (Xi, Xj) ← Estrai(Arc) #Estrai e rimuovi un arco da Arc
    if Rimuovi-Valori-Inconsistenti(csp, Xi, Xj) then
      if dimensione di Di = 0 then return false
      for each Xk in Xi.Adiacenti do
        aggiungi (Xk, Xi) ad Arc
  return true
```

dove

```
function Rimuovi-Valori-Inconsistenti(csp, Xi, Xj) returns true se e solo se viene
rimosso un valore
  rimosso ← false
  for each x in Di do
    if nessun valore y in Dj permette a (x, y) di soddisfare il vincolo tra Xi e Xj then
      rimuovi x da Di
      rimosso ← true
  return rimosso
```

Complessità: $O(cd^3)$ (polinomiale) nel tempo dove c è il numero di vincoli binari e d è la dimensione del più grande tra i domini delle variabili.

2.2. Ricerca con backtracking per CSP

Utile per: fare backtracking, ovvero determinare la soluzione di un CSP dopo aver imposto la consistenza d'arco.

L'algoritmo è basato sulla ricerca in profondità ricorsiva.

Pseudocodice:

```
function Ricerca-Backtracking(csp) returns una soluzione, o fallimento
  return Backtracking(csp, { })
```

dove

```
function Backtracking(csp, assegnamento) returns una soluzione, o fallimento
  if assegnamento è completo then return assegnamento
  var ← Scegli-Variabile-non-Assegnata(csp, assegnamento)
  for each valore in Ordina-Valori-Dominio(csp, var, assegnamento) do
    if valore è consistente con assegnamento then
      aggiungi {var = valore} ad assegnamento
      inferenze ← Inferenza(csp, var, assegnamento)
      if inferenze ≠ fallimento then
        aggiungi inferenze a csp
        risultato ← Backtracking(csp, assegnamento)
        if risultato ≠ fallimento then return risultato
        rimuovi inferenze da csp
```

```
    rimuovi {var = valore} da assegnamento
return fallimento
```