Università degli Studi di Verona

Intelligenza Artificiale

Riassunto dei principali argomenti

Davide Bianchi

4 giugno 2018

Indice

1 Agenti razionali

Agenti. Un agente è semplicemente un'entità che riceve percezioni e produce una risposta con delle azioni. Formalmente un agente è una funzione

$$f : P^* \to A$$

dove P^* è lo storico delle percezioni e A è un insieme di azioni.

Notare che se un agente ha $|P|$ possibili percezioni in ingresso, dopo T unità di tempo la funzione agente avrà il seguente numero di entries:

$$\sum_{t=1}^{T} |P|^t$$

Un agente è in generale una struttura formata da un'architettura fisica e un programma, e prende in input una percezione attuale e ritorna in output l'azione successiva da svolgere.

Esistono principalmente 4 tipi di agenti (ordinati per generalità crescente):

- agenti *simple-reflex* $\to$ agiscono in base alla percezione dell'ambiente;
- agenti *reflex* con stato (model based agents) $\to$ agiscono in base allo stato, le azioni sono condizionate da regole;
- agenti *goal-based* $\to$ le azioni sono condizionate in base al goal prefissato, anche qui è presente uno stato che influisce sul raggiungimento del goal. La soluzione al problema viene eseguita ignorando le percezioni;
- agenti *utility-based* $\to$ le azioni sono condizionate in base ad una utility che rappresenta un valore, si cerca di arrivare nello stato che massimizza questo valore.

Performace measure. La *performance-measure* costituisce una sorta di punteggio che misura il comportamento dell'agente nell'ambiente in cui opera. Quindi, data una performance measure e le percezioni attuale dell'agente, questo sceglie la sequenze di azioni che la massimizzano.

Ambienti. Un ambiente, ovvero lo spazio in cui l'agente opera, è caratterizzato dai seguenti tratti:

- Osservabilità (ho sensori con cui osservo);
- Determinismo;
- Episodicità (non ho bisogno dello storico delle percezioni, mi basta la percezione corrente);
- Staticità (l'ambiente non cambia indipendentemente dall'azione dell'agente);
- Discretezza;
- Presenza di altri agenti (Multi o Single Agent).

Il tipo di ambiente ovviamente condiziona il design degli agenti che vi operano.

2 Problemi di ricerca

Dividiamo i problemi in due macro-categorie:

- Deterministici e completamente osservabili, richiedono un singolo stato;
- Non osservabili, in tal caso gli agenti non sanno dove la soluzione possa risiedere;
- Non deterministici o parzialmente osservabili; problema di contingenza/eventualità (??);
- Lo spazio degli stati è sconosciuto (problemi di esplorazione).

2.1 Formulazione di problemi a stato singolo

Un problema a stato singolo è definito da 4 elementi:

1. uno stato iniziale;
2. una funzione successore (insieme di coppie azione-stato successivo);
3. un test di *goal*;
4. consto del percorso (costo dei singoli step).

Una soluzione è quindi una sequenza di azioni che portano dallo stato iniziale allo stato di goal.

2.2 Ricerca non informata

Le strutture dati utilizzate nelle ricerche su alberi, oltre alla struttura dati contenente l'albero, sono le seguenti:

- una lista `fringe` (una coda FIFO), contenente la *frontiera*, ovvero i nodi foglia disponibili;
- una lista `closed`, contente i nodi di frontiera espansi in passi precedenti.

In generale ogni algoritmo di ricerca su alberi funziona come segue:

```
function treeSearch(problem, strategy)
  inizializza l'abero di ricerca usando lo stato iniziale del problema;
  loop:
    se non ci sono candidati per l'espansione:
      return failure
    scegli un nodo foglia per l'espansione rispettando strategy;
    se il nodo contiene uno stato goal:
      return solution;
    altrimenti:
      espandi il nodo e aggiungi il nodo risultante all'albero
```

Nota: un nodo è una struttura dati, uno stato è una rappresentazione fisica di un nodo, non ha stati padre, ecc.

Il metodo generale per eseguire una ricerca sugli alberi è il seguente:

```
function treeSearch(problem, fringe):
  fringe = insert(makeNode(initialState[problem]), fringe)
  loop:
    if fringe is empty
      return failure
    if goalTest(problem,state(node))
      return node
    fringe = insertAll(expand(node, problem), fringe)
```

Metodo di espansione dei nodi:

```
function expand(node, problem):
  successors = {}
  for each action, result in successorFn(problem, state[node]) do
    s = nuovo nodo
    parentNode[s] = node
    action[s] = action
    state[s] = result
    pathCost[s] = pathCost[node] + stepCost(state[node], action, result)
    depth[s] = depth[node] + 1
    aggiungi s ai successors
  return successors
```

Una strategia possibile è quella di scegliere l'ordine dei nodi da espandere.
Le strategie sono valutate a seconda delle seguenti dimensioni:

- *Completezza*: trova sempre una soluzione se essa esiste?
- *Complessità* in termini di *tempo*: numero di nodi generati/espansi
- *Complessità* in termini di *spazio*: massimo numero di nodi in memoria
- *Ottimalità*: trova sempre la soluzione a costo minore?

La complessità in termini di tempo e spazio fa affidamento sui seguenti termini:

- b: massimo fattore di ramificazione del search tree
- d: profondità della soluzione a costo minimo
- m massima profondità dello spazio degli stati (può essere ∞)

I problemi di ricerca non informata utilizzano solo le informazioni presenti nella definizione del problema.

Uniform-cost search. È l'algoritmo più semplice: espande il nodo con il costo di percorso minore. La frontiera è quindi una coda ordinata per costo. Non guarda al numero di nodi espansi ma unicamente al loro costo.

Breadth-first search (BFS). Espande il nodo non espanso più in superficie. La frontiera è una coda FIFO. Il problema di questo algoritmo è lo **spazio usato**. Infatti, dal momento che deve tenere ogni nodo in memoria, con grandi alberi occupa molto spazio; inoltre ha complessità $O(b^{d+1})$, sia spazialmente che temporalmente. In particolare la complessità in termini di tempo è data dalla seguente formula:

$$1 + b + b^2 + b^3 + \cdots + b^d + b(b^d - 1)$$

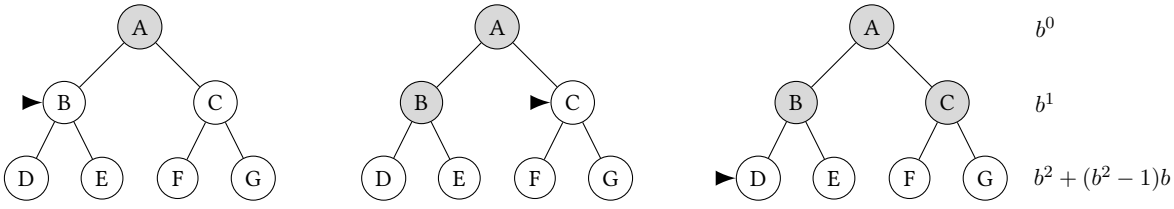


Figura 1.1: BFS

Depth-first search (DFS). Mentre BFS lavora in ampiezza, DFS lavora in profondità, andando ad espandere il nodo non espanso più a fondo possibile. La complessità spaziale è $O(bm)$, che sarebbe ideale se non per il fatto che fallisce in spazi infiniti oppure in spazi con cicli. Temporalmente ha complessità $O(b^m)$, una complessità peggiore quanto più m è maggiore di d.

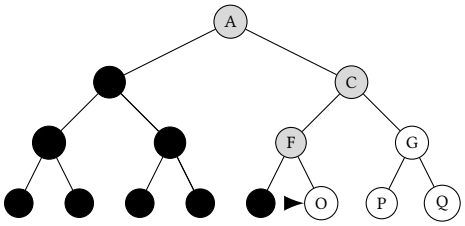


Figura 2: DFS

Depth-limited search. In realtà è solo una variante della DFS, alla quale viene imposto un limite di profondità da raggiungere. La profondità limite, oltre a ridurre lo spazio utilizzato, risolve anche il problema dei cammini infiniti, che nella DFS standard erano il problema più grande che si potesse avere. È anche vero che viene introdotto un altro punto di debolezza, ovvero quello in cui il goal è oltre la profondità limite.

Algorithm 1: Recursive implementation of DLS

```
function DLS(problem, limit):
  recursiveDLS(makeNode(initialState[problem]), problem, limit)

function recursiveDLS(node, problem, limit):
  cutoffOccurred = false
  if goalTest(problem, state[node]) then return node
  else if depth[node] = limit then return cutoff
  else foreach successor in expand(node, problem) do
    result = recursiveDLS(successor, problem, limit)
    if result = cutoff then cutoffOccurred = true
    else if result != failure then return result
  if cutoffOccurred then return cutoff
  else return failure
```

Iterative-deepening search (IDS). È una tecnica usata in combinazione con la DLS per trovare il limite minimo necessario al raggiungimento del goal. Lavora su una profondità variabile chiamando ad ogni iterazione la DLS con il limite corrente. Le complessità sono $O(b^d)$ (temporale) e $O(bd)$ (spaziale). La complessità in termini di tempo in particolare è:

$$(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d$$

Algorithm 2: IDS

```
function IDS(problem)
  for depth = 0 to inf do
    result = DLS(proble, depth)
    if result != cutoff then return result
  end
```

Confronto tra gli algoritmi. Presentiamo di seguito un confronto riepilogativo dei vari algoritmi di ricerca su alberi.

Criterio	BF	UC	DF	DL	ID
Completezza	Si*	Si*,†	No	Si, se $l \geq d$	Si*
Tempo	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Spazio	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Ottimale	Si*	Si	No	Si*, se $l \geq d$	Si*

dove:

- $*$: completo se il branching factor è finito;
- $\dagger$: completo se uno step ha costo $\geq \epsilon$;
- $\star$:ottimale se tutti i costi dei singoli step sono uguali.

2.3 Esempi

BFS vs IDS. prendiamo un albero che abbia le seguenti caratteristiche:

- Sia un albero di ricerca ben bilanciato
- Lo stato di goal è l'ultimo ad essere espanso nel suo livello (rightmost)

Prendiamo, dato l'albero, i seguenti casi

$b = 3,\ d = 3$
BFS

$$b^0 + b^1 + \cdots + b^d + b(b^d - 1) =$$
$$3^0 + 3^1 + 3^2 + 3^3 + 3(3^3 - 1) = 118$$

$b = 4,\ d = 3$
IDS

$$b^0(d+1) + b^1 d + \cdots + b^d =$$
$$4^0(3+1) + 4^1 \cdot 3 + 4^2 \cdot 2 + 4^3 \cdot 1 = 112$$

Cosa succede se il controllo del goal viene effettuato quando si inserisce nella frontiera invece che quando si rimuove? (come avviene in tree search)
Per l'IDS non cambia nulla, il numero di nodi sarà sempre 112.
Per la BFS, ci si ferma al nodo goal, senza espandere i nodi sottostanti, quindi il costo cala drasticamente: $b^0 + b^1 + b^2 + b^3 = 40$

2.4 Ricerca informata

Le strategie di ricerca informata utilizzano conoscenze specifiche riguardanti il problema oltre alla definizione dello stesso, pertanto sono più efficienti. Gli approcci generali sono 2:

- euristiche greedy best-first;
- euristiche su A^* ;

Greedy best-first. Questo approccio cerca di espandere il nodo più vicino al goal, usando una funzione di valutazione. La funzione di valutazione è detta **euristica** ($h(n)$).

La ricerca greedy non è completa (può fallire in caso di cicli). La sua complessità temporale è $O(b^m)$, ma si può migliorare utilizzando euristiche migliori. La complessità spaziale è la stessa, in quanto è necessario tenere in memoria tutti i nodi.

Ricerca con A^* . L'idea è quella di evitare di espandere percorsi che sono già costosi di suo. La funzione di valutazione in tal caso è così composta:

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

dove:

- $f(n)$ è il costo complessivo del percorso attraverso n al goal;
- $h(n)$ è il costo stimato fino al goal a partire da n;
- $g(n)$ è il costo per raggiungere n.

La ricerca con A^* usa un'euristica ammissibile, ovvero un euristica in cui $h(n) \leq h^\star(n)$, dove $h^\star(n)$ è il costo vero da n. Inoltre si richiede che $h(n) \geq 0$, quindi si ha che $h(G) = 0$ per ogni goal.

A^* è completo, però deve tenere tutti i nodi in memoria e ha complessità esponenziale nell'errore relativo di h per la lunghezza della soluzione. Inoltre si può dimostrare che A^* è ottimale.

2.5 Caratteristiche delle funzioni euristiche

Un'euristica si dice consistente se si ha che $h(n) \leq c(n, a, n') + h(n')$.

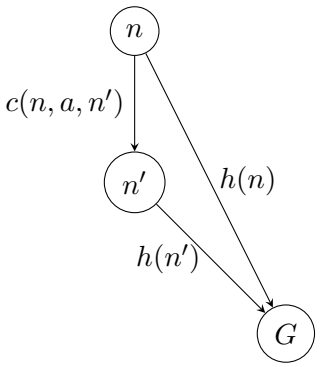
È importante notare che

$$\begin{aligned} \text{consistenza} &\implies \text{ammissibilità} \\ \text{ammissibilità} &\not\implies \text{consistenza} \end{aligned}$$

Inoltre si definisce un'euristica dominante h_2 se vale che, $\forall n$

$$h_2(n) \geq h_1(n)$$

L'euristica dominante è sempre migliore dal punto di vista prestazionale.



3 Constraint reasoning

3.1 Problemi combinatori

Portiamo come principale categoria di problemi risolvibili tramite reti a vincoli i problemi combinatori, ovvero quei problemi dei quali si deve scegliere la soluzione migliore tra le molti possibili. Alcuni di questi problemi sono i problemi di decisione e di ottimizzazione.

Ogni problema combinatorio può essere formulato come, dati un insieme di variabili e domini, la ricerca dei valori delle variabili per i quali vale una certa relazione.

3.2 Reti a vincoli

Una rete a vincoli è una struttura rappresentata come segue:

- un insieme di variabili $X = \{x_1, x_2, \ldots, x_n\}$;
- un insieme di domini $D = \{D_1, D_2, \ldots, D_n\}$;
- insieme di vincoli (S_i, R_i)

Si noti che un vincolo è una coppia (S_i, R_i) dove S_i è una o più variabili $S_i \subseteq X$, mentre R_i è un sottoinsieme del prodotto cartesiano in S_i.

La soluzione di una rete a vincoli è l'assegnamento di tutte le variabili in modo tale che tutti i vincoli siano soddisfatti.

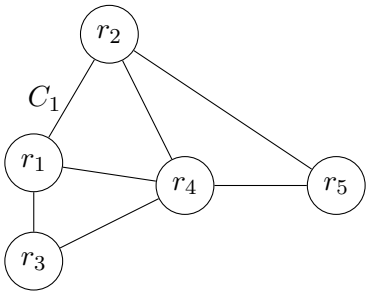
Nota: una soluzione parziale (ovvero un parziale assegnamento delle variabili) potrebbe non essere un sottoinsieme dell'assegnamento facente parte della soluzione.

Esempio. Il problema del *map-colouring* è l'esempio più indicato. In poche parole in una mappa ogni regione deve avere colore diverso dalle regioni adiacenti.

Supponendo di avere 5 regioni $r_1, ..., r_5$, costruiamo il grafo dei vincoli. Le regioni confinanti dovranno essere collegate tra loro da un arco per rappresentare il vincolo del *non avere lo stesso colore*.

Ogni vincolo sarà così formato (ne diamo solo uno di esempio, gli altri sono analoghi):

$$C_1 = (\{r_1, r_2\}, r_1 \neq r_2)$$



Definiamo due nuovi tipi di grafo, ovvero il grafo **primale** e quello **duale**. Il grafo primale ha come nodi le variabili e come archi i vincoli, mentre quello duale è esattamente il contrario: ha come nodi i vincoli, mentre come archi ha le variabili comuni ai vincoli collegati.

3.3 Forzatura di vincoli e propagazione

Quando si parla di soluzioni di reti a vincoli, le tecniche risolutive più utilizzate sono quelle basate su:

- *inferenza*: generazione di nuovi vincoli a partire da quelli già esistenti;
- *ricerca*: ricerca della soluzione per tentativi/backtracking.

L'idea del backtracking è quella di, scelta una variabile, aggiungere un nuovo vincolo su quella variabile e tentare di risolvere il resto del problema. L'assegnamento parziale di una variabile può comunque violare alcuni vincoli del problema, violando la consistenza del vincolo. In tal caso si fa backtracking per tornare alla situazione precedente e procedere per un'altra via.

3.3.1 Consistency

Lo scopo della *constraint inference* è quello di generare reti più forti con uno spazio di ricerca più piccolo, quindi di migliorare prestazionalmente la ricerca della soluzione. Grazie alla propagazione di nuovi vincoli è possibile ridurre lo spazio di valori accettabili da una sola variabile che a sua volta riduce lo spazio di un'altra variabile ecc.

Node consistency. Una variabile x_i del dominio D_i è *node-consistent* se ogni suo valore all'interno del dominio soddisfa ogni vincolo unario. In generale, se una variabile non è *node-consistent*, si possono rimuovere tutti i suoi valori nel dominio che non soddisfano tutti i vincoli unari per quella variabile.

Il nuovo dominio della variabile diventa un dominio D'_i tale per cui:

$$D'_i = D_i \setminus \{v | \exists C = \{\langle x_i \rangle, R_{x_i}\} \wedge v \notin R_{x_i}\}$$

In tal modo il nuovo dominio conterrà solo valori che soddisfano tutti i vincoli, e quelli esclusi non potranno far parte della soluzione.

Arc consistency. Date due variabili x_i e x_j, diciamo che x_i è *arc-consistent* rispetto a x_j se vale che

$$\forall a_i \in D_i \; \exists a_j \in D_j | (a_i, a_j) \in R_{x_i, x_j}$$

Da questo si conclude che R_{x_i,x_j} è *arc-consistent* se x_i è arc-consistent rispetto a x_j e viceversa.

Diamo qui di seguito gli algoritmi possibili per verificare l'*arc-consistency*.

AC1. L'algoritmo **AC1** ha complessità $O(nek^3)$, dove n sono i nodi, e gli archi e k è il massimo numero di valori in un dominio. La procedura di revising è la rimozione dei valori in un dominio per i quali non ha arc-consistency rispetto ad un secondo dominio.

```
repeat
  for all Pairs x_i, x_j that participate in a constraint do
    Revise((x_i), x_j);
    Revise((x_j), x_i);
  end for
until no domain is changed
```

Nota: AC1 termina sempre, e mantiene l'arc-consistency. Se la rete è vuota, non esiste alcuna soluzione.

AC3. L'algoritmo **AC3** ha complessità $O(ek^3)$ e procede come segue:

```
for all every pairs x_i, x_j that participate in a constraint R_{x_i,x_j} ∈ R do
  Q = Q ∪{(x_i, x_j), (x_j, x_i)}
end for

while Q ≠ {} do
  pop (x_i, x_j) from Q
  REVISE((x_i), x_j)
  if D_i changed then
    Q = Q ∪{(x_k, x_i), k ≠ i, k ≠ j}
  end if
end while
```

Domini vuoti. Nel caso in cui un dominio sia vuoto, si ha un problema inconsistente (ovvero non si ha soluzione). Al contrario, il fatto che non tutti i domini siano vuoti non implica che il problema sia consistente. Il principale punto debole dell'arc-consistency sta nel fatto che lavora solamente su vincoli binari e vincoli con un singolo dominio.

Path consistency. È un'evoluzione dell'arc-consistency che utilizza triple di variabili. Una coppia di variabili si dice *path-consistent* in relazione ad una terza variabile quando, per ogni assegnamento $x_i = a, x_j = b$ esiste un valore per x_m tale per cui sono soddisfatti i vincoli (x_i, x_m) e (x_m, x_j). Se i valori cercati violano i vincoli sulla terza variabile, allora tali valori non possono far parte della soluzione, ma **non possono essere rimossi dai loro domini**.

Quando una coppia di valori $(x = a, y = b)$ non è path-consistent rispetto a z non viene rimossa alcuna soluzione: infatti viene rimossa la *coppia* di valori dalla soluzione, ma non vengono rimossi i valori dai rispettivi domini, in quanto questi potrebbero far parte di altre soluzioni, seppur in coppie differenti da quella scartata.

Per forzare la path-consistency si usa l'algoritmo PC-2 (molto simile a AC-3), che però è computazionalmente più complesso ($O(n^3k^5)$) in quanto lavora su triple.

i-consistency. La *i-consistency* è un'estensione del concetto nato con l'arc-consistency e la path-consistency che si estende a reti di $i - 1$ variabili.

Diciamo che una rete è *i-consistent* quando esiste un valore per un assegnamento per le variabili per il quale un valore consistente può sempre essere assegnato alla i-esima variabile.

Inoltre si definisce una rete come *strongly-consistent* se vale che la rete R è j-consistent per ogni $j \leq i$.

3.4 Tree decomposition