Intelligenza Artificiale

Colognese, Rossini

aprile 2018

Rational Agent

Agente razionale: è un'entità che percepisce dall'ambiente attraverso i sensori, entra in uno stato e agisce attraverso gli attuatori (in maniera ciclica). È una funzione che lega la *storia delle percezioni* (per non ripetere azioni inutilmente) alla corrispondente *azione*.

$$f : \mathcal{P}^* \to \mathcal{A}$$

Ogni stato evolve attraverso una o più azioni possibili, generando nuovi stati futuri che vengono rappresentati attraverso un albero. Se l'agente ha $|\mathcal{P}|$ possibili percezioni, la funzione f avrà $\Sigma_{t=1}^{T}|\mathcal{P}|^t$.
Tipi di agente:

- *simple-reflex agent*: non ha stato, percepisce l'ambiente e agisce secondo una condizione;
- *reflex agent with state*: come il precedente ma tiene conto dello stato, di come evolve il mondo e che effetto avranno le sue azioni sull'ambiente;
- *goal-based agent*: come il precedente ma svolge delle azioni per raggiungere un determinato obiettivo (*goal*).
- *utility-based agent*: svolge le azioni non secondo un certo goal ma per massimizzare la performance measure;
- *learning agents*: tutti i precedenti sono convertibili in esso.

Performance measure: è un metodo di valutazione per le azioni svolte dall'agente (punti bonus oppure penalizzazioni). Un agente razionale scegli l'azione da svolgere con l'obiettivo di massimizzare il risultato. Questa soluzione permette di scegliere sempre l'azione migliore ma non garantisce il successo.

PEAS: per progettare un agente razionale bisogna definire i *task environment*.

- *Performance measure*: sicurezza, profitto . . . ;
- *Environment*: traffico, meteo . . . ;
- *Actuators*: sterzo, acceleratore, freno . . . ;
- *Sensors*: tastiera, accelerometro, videocamera, GPS. . . .

Ci sono diversi *tipi di environment*: osservabile, deterministico, episodico (una mossa non limita quelle future), statico (la variante è solo l'agente), discreto, sigle-agent.

Search

Simple-Problem-Solving-Agent(percept): in base ad una situazione del mondo percepita, esegue questi passi:

- inizialmente crea uno stato del mondo attraverso la percezione e da esso produce il goal, il problema e la sequenza di azioni da svolgere;
- esegue un'azione, la elimina dalla lista e ripete fino alla fine della sequenza ignorando ulteriori percezioni.

Tipi di problemi: *Non-Osservabili* (l'agente non ha sensori e ignora le percezioni), *Deterministico, Osservabile* (single-state problem, la soluzione è una sequenza scelta all'inizio), *Non-Deterministico* (ottiene informazioni dalle nuove percezioni e la soluzione viene aggiornata).

Single-State Problem formulation: è definito da uno stato iniziale, funzione successore (insieme di stato-azione), goal test, path cost (aggiuntivo) ed una soluzione. Lo *state space* è un'astrazione degli stati reali (complessi).

Tree Search Algorithms

È offline, simula l'esplorazione degli stati generando successori di stati già esplorati (espansi) secondo una strategia. La strategia è definita dall'ordine di espansione dei nodi. Ogni strategia è valutata riguardo: completezza (se c'è una soluzione la trova sempre), time-complexity (quanti nodi genera), space-complexity (quanti nodi tiene in memoria ad ogni istante), ottimalità (se la soluzione è la migliore).
Time e Space complexity sono misurate in termini di:

- **b**: numero massimo di successori per ogni nodo (*maximum branching factor*);
- **d**: profondità della soluzione minima (*depth*, dipende dallo stato iniziale);
- **m**: profondità massima (può essere infinita se ripeto una mossa).

Tree-Search(problem, fringe): crea una frontiera con i nodi da espandere, rimuove un nodo alla volta, ne fa il Goal-Test e se non è un terminale lo espande e inserisce i successori nella frontiera.

Uninformed Search

Usano solo le informazioni disponibili nella definizione del problema: *Breadth-first search*, *Uniform-cost search*, *Depth-first search*, *Depth-limited search*, *Iterative-deepening search*.

Tree Search Algorithms

Breadth-First Search

Dato un albero, esso viene visitato in ampiezza e come frontiera si usa una coda FIFO.
Completezza: è completo se b è finito.
Tempo: esponenziale in d. $1 + b + b^2 + \cdots + b^d + b(b^d - 1) = \mathcal{O}(b^{d+1})$
Spazio: ogni nodo viene tenuto in memoria (può generare molti nodi). $\mathcal{O}(b^{d+1})$
Ottimale: sì se ogni step ha costo 1; non ottimale in generale.

Uniform-Cost Search

Espande i nodi non espansi col costo minore. La frontiera è una coda con ordine crescente sul path-cost.
È equivalente al *Breadth-first search* se gli step-cost sono tutti uguali.
Completezza: è completo se lo *step-cost* $\geq \epsilon$
Tempo: ♯ nodi con step-cost $\leq$ costo della soluzione ottimale, $\mathcal{O}(b^{\lceil C^*/\epsilon \rceil})$, con C^* = costo della soluzione ottimale.
Spazio: ♯ nodi con step-cost $\leq$ costo della soluzione ottimale, $\mathcal{O}(b^{\lceil C^*/\epsilon \rceil})$
Ottimale: sì, i nodi vengono espansi in ordine crescente.

Depth-First Search

Espande i nodi più profondi attraverso una visita in profondità. La frontiera è una coda LIFO.
Completezza: no, fallisce con spazi di profondità infinita e/o con cicli. È completo con stati finiti.
Tempo: $\mathcal{O}(b^m)$: terribile se $m \gg d$. Se la soluzione è densa è più veloce di *Breadth-first search*.
Spazio: $\mathcal{O}(bm)$: spazio lineare!
Ottimale: no!

Depth-Limited Search

DFS + *depth-limit* l: i nodi alla profondità l non hanno successori. La frontiera è una coda LIFO.
È come la DFS e per ogni nodo ne controlla la profondità per non superare il limite dato (se lo supera ritorna *cutoff*).

Iterative-Deepening Search

Per ogni livello di profondità da 0 a ∞ chiama la funzione *Depth-Limited-Search(problem, depth)* e, se il risultato è diverso da cutoff, lo ritorna. IDS è migliore di BFS perché non espande gli altri nodi a profondità d.
Però BFS può essere modificata per applicare il goal-test quando un nodo è generato.
Completezza: Sì.
Tempo: $(d+1)b^0 + bd^1 + (d-1)b^2 + \cdots + b^d = \mathcal{O}(b^d)$
Spazio: $\mathcal{O}(bd)$, genera i nodi più volte.
Ottimale: sì, se lo step-cost $= 1$

Graph Search

Si differenzia dal *tree-search* perché prima di espandere un nodo lo mette in **Closed** e lo toglie dalla frontiera per non espanderlo più. Può essere molto più efficiente del *tree-search*.
Si parte dal grafo e si risolve applicando un algoritmo di Tree-Search, duplicando gli stati raggiungibili in più modi.

Informed Search

Best First Search

Usa una ***funzione di valutazione*** per ogni nodo che ci dice quanto esso sia desiderabile per il cammino ottimo.
Espande il nodo più desiderabile. La frontiera è una coda ordinata in modo decrescente sull'ordine di desiderabilità.
Ci sono due casi speciali che sono: *Greedy Search* e *A* Search*.

Greedy Search

La funzione di valutazione è l'*euristica (h(n))* che è la stima del costo da n al nodo goal (distanza in linea d'aria).
Espande il nodo con l'euristica più bassa.
Completezza: no, può bloccarsi all'interno di certi loop. Sì, se c'è controllo sugli stati ripetuti.
Tempo: $\mathcal{O}(b^m)$: con una buona euristica può avere buoni miglioramenti.
Spazio: $\mathcal{O}(b^m)$: tiene tutti i nodi in memoria.
Ottimale: no.

A* Search

Evita di espandere percorsi già troppo costosi. La funzione di valutazione è: $f(n) = g(n) + h(n)$, dove $g(n)$ è il costo per raggiungere n dalla partenza. A^* usa un'**euristica ammissibile** che non sovrastima il costo reale.
Completezza: sì, a meno che non ci siano infiniti nodi con $f \leq f(Goal)$.
Tempo: esponenziale.
Spazio: tiene tutti i nodi in memoria.
Ottimale: sì, non espande $f(i+1)$ finché non ha finito $f(i)$. Espande solo i nodi con $f(n) \leq C^*$.

Heuristics

L'**euristica** è **consistente** se $h(n) \leq c(n, a, n') + h(n')$, dove il path che passa da n' non è ottimo.
Un'euristica consistente è sicuramente ammissibile; se è ammissibile non è necessariamente consistente.
Tree Search + Euristica Ammissibile $\rightarrow$ A* ottimale
Graph Search + Euristica Ammissibile $\nrightarrow$ A* ottimale (può scartare path ottimali su nodi ripetuti)
Graph Search + Euristica Consistente $\rightarrow$ A* ottimale

Se $h_2(n) \geq h_1(n)$ (entrambe ammissibili), allora $h_2(n)$ domina $h_1(n)$ ed è migliore per la ricerca.
Date due euristiche ammissibili $h_1(n), h_2(n)$: prendiamo $h(n) = max(h_1(n), h_2(n))$.

Processamento dei Vincoli

Dato un set di possibili soluzioni, trova la migliore (non si possono provare tutte).
Dato un problema decisionale, se non è risolvibile diventa un *problema di ottimizzazione* (numero minimo di conflitti).
L'ottimizzazione può essere multi-obiettivo, come ad esempio: minimizzare i rischi e massimizzare gli obiettivi.

Graphical Model: Reti a Vincoli

Una rete a vincoli è caratterizzata: un *insieme di variabili*, un *insieme di domini* (uno per ogni variabile), un *set di funzioni locali* (regolano le relazioni tra alcune variabili). Una *funzione globale* è un'aggregazione di funzioni locali.
L'obiettivo è trovare dei valori per le variabili tali che valga una certa relazione tra loro.
Le relazioni non binarie sono più complesse.
La combinazione di due soluzioni parziali consistenti può non essere consistente (***backtracking*** necessario).

Constraint Graph

Ci sono due tipi di grafi:

- *Grafo Primale*: i nodi sono le variabili; gli archi sono i vincoli tra di esse.
- *Grafo Duale*: i nodi sono le variabili coinvolte nei vincoli; gli archi sono le variabili condivise dai nodi.

La rappresentazione dei vincoli può essere di tre tipi: tabellare, aritmetica ($a \neq b$) e booleana ($a \ \& \ b$).
Le *relazioni* sono il prodotto cartesiano delle variabili alle quali assegno tutte le combinazioni che rispettano i vincoli.

Node and Arc Consistency

Ci sono due tecniche risolutive per le Reti a Vincoli:

- *inferenza*: derivare nuovi vincoli basati su quelli esistenti; eliminazione di valori delle variabili che non rispettano i vincoli.
- *ricerca*: cercare una soluzione provando i diversi valori delle variabili; backtracking.

Backtracking: scegliere una variabile x, per ogni valore aggiungere un vincolo $x = v$ e valutare ricorsivamente il resto del problema. Viene fatto quando gli assegnamenti parziali violano i vincoli (*local consistency*).
Consistenza: quando si forza la consistenza locale, si potrebbe scoprire che il problema è in realtà inconsistente.
Data una soluzione parziale di lunghezza $i - 1$, può essere estesa ad un'altra variabile connessa (*i-consistency*).
Arc-Consistency: vale da 1 a 2 variabili.
Global-Consistency: è una rete i-consistente per $i = 1, \dots n$. Più alto è i, migliore sarà il risultato; però il costo in termini di tempo e spazio è esponenziale in i.
Node-Consistency: data una variabile x_i ed il dominio D_i, se ogni valore del suo dominio del soddisfa ogni vincolo. Posso forzarla rimuovendo i valori dal dominio che non soddisfano tutti i vincoli (se vuoto, non c'è soluzione).

Constraint Propagation *(algoritmo)*: disegnare il grafico con nodi (scrivere dominio della variabile) e archi (scrivere i vincoli su di essi). Scegliere variabile e vincolo: eliminare dal dominio i valori non presenti nel vincolo e propagare.
Revise((x), y): rimuove gli elementi del dominio di x che non hanno relazioni con elementi del dominio di y.

AC-1 *(algoritmo)*: si fa il *Revise* di tutte le coppie da entrambi i versi (se $D_i = \emptyset$, la rete è inconsistente). Il procedimento si ripete fino a quando non ci sono più variazioni nei domini (termina sempre). Complessità: $\mathcal{O}(nek^3)$
Nel caso peggiore si elimina un elemento dal dominio ad ogni ciclo (ci possono essere al più nk cicli).
AC-3 *(algoritmo)*: è come *AC-1*, però quando analizzo (x_i, x_j) modifico D_i devo aggiungere alla coda tutte le coppie (x_k, x_i) con $k \neq i$ e $k \neq j$. Complessità: $\mathcal{O}(ek^3)$.
Arc-Consistent + Domains_Not_Empty $\nrightarrow$ Consistent Problem (tranne per i problemi trattabili che sono polinomiali).

Inconsistenza: quando si forza la consistenza locale, si può scoprire che il problema è inconsistente (ad esempio l'Arc-Consistenza e il dominio vuoto). Però non vale sempre il contrario.
Dato un albero con ogni nodo arc-consistente col figlio, allora il problema è globalmente consistente. Questo è dovuto al fatto che i nodi fratelli non introducono inconsistenza. Il problema sono i *cicli*.
Un problema arc-consistente è anche globalmente consistente se: non ha domini vuoti, i vincoli sono binari, il grafo primale non contiene cicli.

Search Strategies: Lookahead

Binary CSP *(Constraint Satisfaction Problem)*: ogni vincolo collega al massimo due variabili. C'è un arco tra due nodi quando vi è un vincolo tra essi. Gli stati sono definiti dai valori assegnati nel seguente modo: *initial state* (vuoto), *funzione successore* (se possibile, assegna un valore ad una variabile libera senza creare conflitti altrimenti *fail*), *goal test* (assegnamento completo).
Backtracking: lo scopo è ridurre la dimensione dello spazio di ricerca esplorato. È necessario ordinare le variabili, consistenza locale (arc oppure path consistency), *look-ahead* (predirre future inconsistenze), *look-back* (dove fare backtrack) e *tree-decomposition.*

MRV: scelgo la variabile con il minor numero di valori possibili.
Degree Heuristics: scelgo la variabile con più vincoli (nodo con più archi).
Scelta delle variabili - Valore meno vincolante: scelgo il valore che limiterà meno le altre variabili.
Backtrack Free Search: una rete è *backtrack free* se ogni foglia è un nodo goal.

Forward Checking: propaga l'effetto di un valore ad una futura variabile (vincolata). Se il dominio della variabile futura diventa vuoto, si prova un altro valore per la variabile corrente.
Arc-Consistency Look-Ahead *(algoritmo)*: disegno il grafico e riporto tutti i vincoli. Devo eliminare i cicli e lo faccio rimuovendo una o più variabili (quelle con più archi). Nel grafo rimanente rappresento anche i vincoli con le variabili rimosse (archi tratteggiati). Assegno i valori alle variabili rimosse e propago sul grafo rimanente.

Tree Decomposition Methods

Si può sempre trasformare un grafo ciclico in un albero aciclico.

Acyclic Network

Ipergrafo: i nodi raggruppati formano una clicca. Si può trasformare in grafo primale e duale.
Nel grafo duale possono esserci archi ridondanti che possono sembrare cicli (si toglie l'arco con grado minore).
Running Intersection property *(per la Dual Based Recognition)*: un vincolo tra due nodi può essere rimosso se le varibili che etichettano l'arco sono contenute in altri path attraverso questi due nodi (***Maximum Spanning Tree***).
Join Tree: grafo duale senza cicli in cui vale la *Running Intersection property*.
Tree Solver: scelgo una clique come radice, rappresento l'albero con i vincoli e faccio il *Revise* tra la radice e i figli (rimane una o più soluzioni).

Ci sono due metodi per verificare se una rete è aciclica: *Dual Based Recognition* e *Primal Based Recognition*.
Primal Based Recognition: generare il grafo primale relativo all'ipergrafo (*conformalità*) e verificare la *cordalità* verificando se ogni nodo forma una clique con i suoi *anchestors*. Con le clique massimali genero poi il grafo duale e faccio Maximum Spanning Tree.
Max-Cardinality Order: scelgo un nodo arbitrariamente, scelgo un altro nodo che ha il numero massimo di connessioni ai nodi già scelti e ripetere.

Clustering

Consiste nel raggruppare sottoinsiemi di vincoli per formare una struttura ad albero.

Algoritmo: dato un ordine di variabili, creare il **grafo indotto** (analizzo gli *anchestor* e aggiungo vincoli per creare le clique). Faccio Spanning Tree sul grafo e aggiungo i vincoli corrispondenti alle clique massimali (mai doppi).
Si specificano i vincoli (disguaglianza) sul grafo primale e li riporto nelle clique (tabella, stando attento a quali vincoli ho assegnato alla clique) mantenendo solo le combinazioni valide (rimarranno solo le possibili soluzioni). Poi faccio il Revise del nodo/clique padre verso il figlio, cioè *Rev((p),f)*.

Il problema è polinomiale eccetto trovare le combinazioni valide (esponenziale).

Constraint Optimisation Problems

Combinatorial Auction: $S = \{a_1, \dots, a_n\}$ set di oggetti, $B = \{b_1, \dots, b_m\}$ insieme di offerenti (*bids*). Abbiamo vari $b_i = (S_i, r_i)$ che associano uno o più item, con un costo complessivo, ad un acquirente.
L'obiettivo è trovare un sottoinsieme di offerenti $B' \subseteq B$ tale che nessuna coppia di offerenti non condivida nessun item e $C(B) = \Sigma_{b_i \in B'} r_i$ è massimizzata.
Cost Network: Constraint Network + Cost Function
Cost Function: $F(\overline{a}) = \Sigma_{j=1}^{l} F_j(\overline{a})$

Variabili b_i, domini $D_i = \{0, 1\}$. $b_i = 1$ significa che il bid i (insieme di variabili) è stato selezionato.
Hard-Constraint: due *bids* selezionati non posso condividere nessun item.
Soft-Constraint: selezionare i *bids* che massimizzano la somma del loro costo.

Branch and Bound

Si rappresentano i bids in ordine con i rispettivi costi ed i vincoli (due bids collegati devono avere valori diversi).
Faccio la sommatoria di tutti i costi per avere il valore massimo. Sviluppo l'albero binario (es. $b_1 = 1, b_2 = 0 \dots$) facendo attenzione ai vincoli (se b_1 è collegato a b_2 non possono essere entrambi 1).
Il valore più alto in fondo all'albero sarà la soluzione L.

Bucket Elimination

Sono procedure di programmazione dinamica per risolvere i *Constraint Optimisation Problems*.
Un Bucket B_i è un insieme di vincoli che si riferiscono ad una certa variabile x_i.

- Si assegnano i vincoli al bucket seguendo un ordine dato. Si assegna un vincolo al primo bucket rispettivo ad una delle variabili contenute dal vincolo.
- Per ogni Bucket si genera una ***funzione di massimizzazione*** ($H^b(a, c)$) contenente tutte le variabili dei vincoli del bucket esclusa quella che lo identifica. Questa si inserisce nel primo bucket che fa riferimento ad una delle variabili della funzione. L'ultimo bucket il numero massimo M che posso ottenere dal problema.
- Per ogni Bucket si genera il valore massimizzato della variabile relativa: $a^* = argmax_a(F_0(a) + H^f(a))$