Intelligenza Artificiale e Machine Learning

Appunti delle Lezioni di Intelligenza Artificiale e Machine Learning
Anno Accademico: 2024/25

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeX disponibile al seguente link:
https://github.com/00Darxk/Intelligenza-Artificiale-e-Machine-Learning

Indice

1 Introduzione

1.1 Storia dell'Intelligenza Artificiale

2 Risoluzione dei Problemi e Ricerca

Si definisce un agente risolutore di problemi un agente con uno specifico obiettivo da raggiungere e che deve identificare una sequenza di azioni per raggiungerlo.

Bisogna determinare l'obiettivo, un insieme degli stati del mondo dove ci si trova, che si vuole raggiungere. Inoltre bisogna formulare il problema, ovvero le azioni e gli stati considerati dall'agente.

Un agente che ha a disposizioni diverse opzioni immediate di valore sconosciuto, può decidere quale scegliere la sua azione analizzando le diverse possibili sequenze di azioni, che portano a stati di valore conosciuto, per scegliere la sequenza di costo migliore.

Questo processo di selezione viene definito ricerca, un algoritmo di ricerca quindi prende un problema e restituisce una soluzione costituita da una sequenza di azioni.

Nella formulazione si definisce uno stato iniziale e dell'obiettivo, come un insieme di stati e si definiscono le azioni come transizioni tra stati. Dopo aver trovato la sequenza di azioni corrispondente alla soluzione, la esegue.

Si possono distinguere due tipi di problemi, i problemi giocattolo o "toy problems", sono ideati come illustrazione o esercitazione dei metodi risolutivi. Rappresentano delle astrazioni, anche semplificate, dei problemi del mondo reale in generale più difficili a cui si è effettivamente interessati.

I problemi del mondo reale possono essere la configurazione VLSI, la navigazione dei robot, la sequenza di montaggio, la ricerca dell'itinerario e generali problemi di viaggio come il commesso viaggiatore.

Si possono utilizzare due tipi diversi di formulazione di un problema. Si può partire da uno stato vuoto, ed utilizzando operatori si può estendere progressivamente la descrizione dello stato. Invece nella formulazione a stato completo consiste nel partire da uno stato iniziale completo, ed ogni operatore altera questo stato per cercare una soluzione.

Lo spazio degli stati è un grafo che rappresenta tutti i possibili stati, come nodi, collegati tra archi che rappresentano le possibili azioni. Il problema consiste nel trovare un percorso in questo spazio degli stati, dallo stato iniziale ad uno dei possibili stati soluzione. L'algoritmo deve decidere ad ogni stato quale azione prendere e quindi a quale nodo del grafo spostarsi. Per definire il costo della soluzione, si considera il costo del cammino delle azioni intraprese dall'agente.

Per definire formalmente un problema, sono necessarie quattro componenti. Uno stato iniziale in cui si trova l'agente. Una descrizione delle azioni possibili, questa può utilizzare una funzione successore che dato uno stato restituisce i suoi possibili successori. Tramite la funzione successore e lo stato iniziale si può costruire lo spazio degli stati. Oppure si può utilizzare un insieme di operatori. Un test obiettivo per determinare se un particolare stato è uno stato obiettivo. Come ultimo componente è necessaria una funzione di costo del cammino per determinare il costo di una data soluzione, assegnando un valore numerico ad ogni cammino.

Il tipo di dato problema è rappresentato da questi quattro componenti, le istanze di questo tipo di dato rappresentano gli input degli algoritmi di ricerca.

Il mondo reale è estremamente complesso, e lo spazio degli stati deve essere creato mediante un processo di astrazione. In questo processo, lo stato astratto rappresenta un insieme di stati reali più complessi. Analogamente per le azioni astratte, queste rappresentano combinazioni di azioni reali. Nei problemi giocattolo non è necessario effettuare questo processo di astrazione, poiché rappresenta un problema semplice.

Per individuare queste possibili sequenze di azioni l'algoritmo si può costruire un albero di ricerca, dove il nodo radice corrisponde allo stato iniziale, ed i rami rappresentano le azioni possibili, ed i nodi figli rappresentano gli stati successori di un certo stato. Tuttavia i nodi dell'albero di ricerca e gli stati dello spazio degli stati sono differenti, poiché è possibile che più nodi condividano lo stesso stato, mentre ogni stato nello spazio è univoco. Generalmente si vuole evitare di ripetere stati all'interno di un cammino, questo rappresenta il problema degli stati ripetuti, e sarà analizzato successivamente.

Ad ogni nodo si possono inserire altre informazioni utili, oltre allo stato, un riferimento al genitore, l'operatore che ha generato lo stato, la profondità, il costo del cammino parziale fino a questo stato, etc.

```
NODO = <stato, genitore, operatore, profondità, costo parziale, ...>
```

Il processo di ricerca comporta la stessa sequenza di azioni. Deve scegliere tra le foglie dell'albero corrente un nodo da "espandere", secondo un certo criterio o strategia. In seguito bisogna determinare se questo nodo rappresenta un'obiettivo del problema, altrimenti vengono generati i suoi nodi figli, ed i corrispondenti stati successori, e tutte le componenti dei nodi figli. La collezione dei nodi in attesa di essere espansi viene chiamata in vari modi: confine, frontiera, frangia o lista aperta.

2.1 Algoritmo di Ricerca Generale: Tree Search

Un algoritmo generale di ricerca può essere chiamato `TREE-SEARCH`. Quest'algoritmo prende un problema ed una strategia come input e restituisce una soluzione oppure un fallimento. Viene realizzato semplicemente ad un ciclo che ripete le operazioni precedentemente descritte, fino a quando non identifica una soluzione o viene sollevato un problema, e quindi restituisce un fallimento. Il primo passo è la generazione dell'albero di ricerca del problema, se la frontiera è vuota, ovvero non esistono nodi candidati per l'espansione viene riportato un fallimento, poiché non è stato ancora trovato uno stato obiettivo. Se si arriva ad un nodo corrispondente ad uno stato obiettivo viene restituita rappresenta la sequenza di nodi ottenuta come soluzione.

La frontiera può contenere un nodo relativo ad uno stato obiettivo, ma l'algoritmo non termina fino a quando non viene scelto per essere espanso.

Una strategia di ricerca rappresenta un criterio per decidere quale nodo da espandere. Può essere definita come una funzione per la scelta di un elemento tra un insieme di nodi, la frontiera. Oppure si può considerare come una funzione di inserimento di un elemento in una sequenza. Se già nella fase di inserimento si analizza tramite una metrica il valore di ognuno di questi stati, allora l'elemento in prima posizione in questa struttura dati rappresenta il nodo migliore.

2.1.1 Operazioni su Frontiera, Nodi e Problemi

La frontiera viene implementata con una struttura dati chiamata coda, ma non necessariamente segue la disciplina FIFO. Su questa coda sono definite una serie di operazioni:

- `MAKE-QUEUE`: prende come input un nodo `n` e restituisce una coda `q`, realizza una coda contenente solo il nodo `n`;

- `EMPTY`: prende come input una coda `q` e restituisce un booleano, verifica se la coda è vuota;
- `REMOVE-FRONT`: prende come input una coda `q` e restituisce il primo nodo `n` della lista;
- `QUEUING-FN`: prende come input una coda `q` ed una lista di nodi `n`, e restituisce la coda con aggiunti tutti questi nodi.

L'ultima funzione si utilizza quando si producono tutti i nodi successori e si vogliono aggiungere alla lista. Questa funzione non inserisce generalmente in coda, ma dipende dalla strategia di ricerca utilizzata.

Assumendo che esistano le operazioni sul tipo di dato problema, e sul tipo di dato nodo. Le operazioni sul tipo di dato problema sono:

- `INITIAL-STATE`: prende come input un problema `p` e restituisce lo stato iniziale del problema `n`;
- `GOAL-TEST`: prende come input un problema `p` ed uno stato `n` e verifica se questo rappresenta una soluzione, restituendo un booleano;
- `OPERATORS`: prende come input un problema `p` e restituisce una lista con tutti gli operatori del problema `ops`. Ogni operatore `op` applicato ad uno stato `n` restituisce una lista di stati `ss`.

Le operazioni sul tipo di dato nodo sono:

- `MAKE-NODE`: prende come input uno stato `s` e costruisce un nodo su di esso `n`;
- `STATE`: prende come input un nodo `n` e ne restituisce lo stato contenuto `s`;
- `EXPAND`: prende come input un nodo `n` ed una lista di operatori `ops` e restituisce una lista di nodi successori `ns`.

2.1.2 Implementazione

Date queste operazioni, si può rappresentare in pseudocodice l'algoritmo di `TREE-SEARCH` in modo più semplice:

```
function TREE-SEARCH(problem) returns a solution or failure

fringe <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE(problem)))
loop do
    if EMPTY(fringe) then return failure
    node <- REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return SOLUTION(node)
    fringe <- QUEUING-FN(fringe, EXPAND(node, OPERATOR(problem)))
end
```

In questa variazione non viene conservato l'intero albero, ma solamente la coda con i nodi della frontiera.

2.2 Criteri di Valutazione

Per valutare questi algoritmi oltre alla complessità temporale e spaziale, si utilizzano altre due criteri, la completezza e l'ottimalità. Un'algoritmo si definisce completo, se quando esiste una soluzione è garantito sia in grado di trovarla. Un algoritmo si dice ottimo se dato un problema con diverse soluzioni, individua sempre la migliore, quella a costo minimo. La complessità dell'algoritmo dipende dal fattore di ramificazione dello spazio degli stati e dalla profondità della soluzione più superficiale.

3 Algoritmi

3.1 Problema di Ricerca Globale

Nei problemi precedenti quando l'algoritmo risolutivo raggiungeva uno stato obiettivo, il cammino verso quello stato rappresenta una soluzione del problema. Tuttavia in alcuni problemi lo stato obiettivo contiene tutte l informazioni rilevanti per la soluzione, dove il cammino è irrilevante. Come esempio si consideri il problema dell otto regine, è indifferente il cammino attraverso gli stadi intermedi, solamente lo stato finale, la disposizione delle regine nello stato finale.

Algoritmi di ricerca locale si utilizzano per risolvere questo tipi di problemi. In questi problemi è sempre presente uno spazio degli stati ed uno spazio degli stati aventi ciascuno una sua valutazione. Si può immaginare questi stadi su una superficie del territorio, uno spazio dove l'altezza di questo stato rappresenta la sua valutazione. L'algoritmo quindi itera su ognuno di questi stati per cercare quello di altezza maggiore, o minore, identificando quindi la soluzione al problema, indipendentemente dal cammino preso per raggiungerla. Questi punti di massimo rappresentano dei picchi, i cui punti adiacenti sono strettamente minori dello stato di massimo. Quindi l'algoritmo che parte da uno stato iniziale deve cercare un massimo globale in questo spazio, determinando quale sia tra i vari massimi locali ed i massimi locali piatto, e le "spalle" massimi locali "piatti", prima di un massimo globale.

Questi algoritmi chiamati anche di miglioramento iterativo, si muovono sulla superficie cercando questi picchi, senza tenere traccia del cammino effettuato, tenendo solamente traccia dello stato attuale e dei suoi vicini o successori, gli stati immediatamente adiacenti. Bisogna formulare il problema in modo che l'algoritmo non rimanga bloccato tra due massimi locali.

3.1.1 Algoritmo di Hill-Climbing

Questo algoritmo segue sempre le colline più ripide, si muove sempre verso l'alto nella direzione dei valori crescenti, e termina quando raggiunge uno stato per il quale si ha un picco che non ha vicino stati di valore maggiore. Tuttavia questo algoritmo può rimanere intrappolato su massimi globali.

Non viene memorizzato lo stato corrente, solamente il valore attraverso nodi che contengono lo stato ed il suo valore: `NODO=<STATO, VALORE`.

Esistono diversi tipi di algoritmi di questo genere per evitare di rimanere bloccati su picchi locali, utilizzando diverse tecniche.

Dallo stato corrente si ricava il suo valore, in seguito comincia un ciclo che prende in considerazione tutti i successori e si sceglie come `next` il nodo di valore più alto. Se tutti i nodi adiacenti

hanno un valore minore di `next` allora questo rappresenta la soluzione dell'algoritmo e l'algoritmo termina, tuttavia questo stato potrebbe corrispondere ad un massimo locale, invece se esiste uno stato adiacente di valore maggiore, questo diventa `next` e si passa alla nuova iterazione.

L'algoritmo contiene una componente di ripartenza casuale, questo infatti conduce una serie di ricerche di Hill-Climbing partendo da stati generati casualmente. Questo algoritmo da un punto di vista teorico è completo, poiché con una serie infinita di ripartenze, sicuramente l'algoritmo visita tutti gli stati del sistema, trovando sicuramente la soluzione ottima del problema.

Il ciclo interno ad ogni iterazione genera un ottimo locale, e prova ad evitare ottimi locali effettuando una nuova ricerca da un nuovo stato scelto casualmente.

L'algoritmo Stochastic Hill-Climbing si ottiene modificando la procedura normale dell'algoritmo. Invece di valutare tutti i vicini dallo stato corrente, l'algoritmo sceglie casualmente uno solo dei suoi successori da valutare per determinare se si tratta il successore, ed in caso diventa il nuovo stato corrente `next`, questo viene accetta con una probabilità che dipende dalla differenza della valutazione tra i due punti: $\Delta E =$`VALUE(current)-VALUE(next)`.

Il nuovo stato viene scelto con una probabilità p, calcolata come:

$$p = \frac{1}{1 + e^{\Delta E/T}} \tag{3.1.1}$$

In seguito dopo una serie di iterazioni l'algoritmo restituisce uno stato ottimo. L'algoritmo ha quindi un solo ciclo, e può scegliere un nuovo punto con una probabilità p, quindi anche di valore minore. Questa probabilità dipende da un parametro T costante durante l'esecuzione dell'algoritmo. Se vale 1, la probabilità di accettazione è sostanzialmente pari al 100%.

All'aumentare del valore i T la probabilità di accettazione tende al 50%, diventa quindi sempre meno importante la differenza della valutazione tra i due punti, effettivamente comporta una ricerca casuale, mentre al diminuire di T, la procedura rappresenta un semplice algoritmo Hill-Climbing.

In caso di stati di valore uguale, la probabilità è del 50%, se il valore dello stato `next` è minore, la probabilità diminuisce, mentre se il valore di `next` è maggiore dello stato corrente, la probabilità aumenta.

Bisogna trovare una "link function" tra l'intervallo $\Delta E/T$ e la probabilità p.

La caratteristica di poter scegliere come passo uno stato peggiore questo algoritmo potrebbe evitare massimi locali.

Questo algoritmo prende il nome dall'analogia con il processo di metallurgia per temprare un materiale, questo processo infatti raggiungere uno stato di struttura cristallina ad energia minima. La differenza principale con l'algoritmo stocastico, è la possibilità di variare il valore di T gradualmente durante l'esecuzione dell'algoritmo. Il valore di T parte da un valore elevato, per poi diminuire nel tempo, come se fosse la temperatura durante un processo di temperatura.

In questo algoritmo la probabilità di accettazione di un certo nodo viene implementata in modo differente rispetto all'algoritmo stocastico normale, poiché nel simulated annealing si considera solamente un semiasse dell'ascissa, dato che in caso `next` sia migliore non viene calcolata la probabilità di accettazione. Si accetta sempre uno stato di valore migliore, mentre l'accettazione di uno stato peggiore dipende da una probabilità.

Molte implementazioni dell'algoritmo seguono la stessa sequenza di passi. Si assegna la variabile `T`, e si sceglie uno stato corrente casuale al primo passo. Si determina un successore e si ripete per

un certo numero di cicli nel secondo, e come passo finale si diminuisce la temperatura e si ripete dal primo passo se la temperatura non ha raggiunto la temperatura minima. Quindi l'algoritmo termina solamente quando la temperatura ha raggiunto la temperatura minima.

Le aree di applicazione di questo algoritmo sono molto vaste nell'informatica. Proposto negli anni '80 ha avuto un enorme successo, anche solo nella sua versione base.

4 Introduzione a Python

Python è un linguaggio di programmazione vastamente utilizzato nell'area dell'intelligenza artificiale e nel machine learning. Recentemente è diventato il linguaggio di programmazione più diffuso al mondo. Python è un linguaggio general-purpose, ideato da Guido van Rossum nel 1989, a più alto livello del C, poiché gestisce automaticamente le più fondamentali operazioni.

La versione di Python utilizzata nel corso è la versione 3, nell'ambiente Anaconda. Può essere avviato tramite un interprete. Alla creazione di una variabile non è necessario definirne il tipo, il nome identificativo è arbitrario e può contenere numeri, ma non cominciare con un numero, viene consigliato di utilizzare un carattere minuscolo come primo carattere del nome. Esistono 33 parole chiave, non utilizzabili come nomi di variabili. Si può assegnare un valore ad una variabile tramite l'operatore `=`, senza specificarne il tipo.

Esistono una serie di operatori aritmetici come `+`, `-`, `*`, \, `**` e \\ (floor division). Una differenza tra Python 2 consiste nella gestione del resto, infatti in Python 2 viene considerata solo la parte intera del resto.

Si possono inserire dati tramite la funzione `input`, e si può convertire in un tipo specifico come `tipo(var)`. I commenti vengono realizzati tramite il carattere #.

In Python sono presenti tutti gli operatori booleani di C, in aggiunta sono presenti altri operatori `is` ed `is not`.

Sono presenti gli stessi operatori logici di C, e come in C un qualsiasi valore diverso da zero corrisponde al booleano `true`.

In Python per identificare funzioni o istruzioni condizionali non si usano parentesi, ma si indenta di quattro posizioni. Dopo la condizione dell' istruzione condizionale vanno inserite dei due punti `:`.

Si possono gestire le eccezioni con il costrutto `try` ed `except`:

```
try:
  # corpo del try
except:
  # corpo dell'except
```

In Python sono integrate tantissime funzioni utili, per svolgere attività comuni, utilizzabili senza doverle definire, queste sono funzioni "built-in". Per invocare funzioni presenti in un certo modulo si utilizzata la notazione puntata `nomeModulo.nomeFunzione()`.

Dati gli algoritmi analizzati precedentemente, si nota la necessita di introdurre generatori di numero casuali. La maggior parte di generatori casuali, sono deterministici, ovvero dato lo stesso input, generano gli stessi numeri casuali. Si utilizzano quindi numeri pseudo-casuali, generati da

un calcolo deterministico, ma non è quasi possibile distinguerli da numeri generati casualmente. In Python esiste il modulo `random` contenente funzioni pertinenti alla generazione di numeri casuali.

Per definire nuove funzioni si utilizza la parola chiave `def`, specificando il nome, tra parentesi tonde gli argomenti ed i due punti, indentando di quattro posizioni per scrivere il corpo della funzione:

```
def nomeFunzione(nomeArgomenti):
  # corpo della funzione
# resto del codice
```

Dopo aver passato degli argomenti ad una funzione, questi vengono assegnati a delle variabili locali. Si può utilizzare anche una variabile come argomento. Inoltre tutte le aggiunte possibili alle funzioni built-in, si possono effettuare sulle funzioni definite dall'utente come `_twice`, per ripetere due volte la funzione.

Si dividono le funzioni in due tipi "fruitful function" e "void function", le prime restituiscono un valore, le secondo non restituiscono valore. Le prime quindi vengono usate per assegnare o inizializzare variabili. Se si tenta di assegnare il risultato di una void function ad una variabile, viene ottenuto un valore chiamato "None". Questo valore ha un suo proprio tipo.

Si possono realizzare cicli tramite il costrutto `while` o `for`, seguito da una condizione booleana e dai due punti `:`. Si può interrompere il ciclo con `break`, e si può saltare l'iterazione corrente con `continue`. Quando bisogna iterare su una collezione, si può realizzare un ciclo for-each:

```
for elem in Array
  # corpo del ciclo
```