UNIVERSITÀ DEGLI STUDI DI MILANO

Algoritmi genetici con Mathematica: risoluzione del problema di Koza

Claudio Chiappetta
July 6, 2016

1 IL PROBLEMA

È stato scelto di risolvere un problema noto nel campo dell'intelligenza artificiale, la cui risoluzione ha contribuito a ...

Si ha a disposizione un insieme di lettere, che opportunamente combinate formano una parola data. Nel nostro caso, seguendo l'esperimento di Koza, è stata scelta la parola "universal".

Queste lettere ("blocchi")sono suddivise in un insieme liberamente accessibile (il "tavolo"), e un insieme che si comporta come una pile (la "pila"). Si suppone inoltre di avere un "robot" che possa compiere determinate operazioni sui blocchi. Queste operazioni sono:

- **CS** Ritorna l'elemento in cima alla pila.
- **TB** Ritorna l'ultimo elemento correttamente ordinato della pila.
- **NN** Ritorna la lettera successiva a **TB** nella parola "universal".
- **MS**(x) Se x è sul tavolo, sposta x in cima alla pila e ritorna x.
- **MT**(x) Se x è nella pila, sposta sul tavolo l'ultimo elemento della pila, e ritorna x.
- **DU**($esp1, esp2$) Valuta $esp1$ finchè $esp2$ non diventa TRUE.
- **NOT**(esp) Ritorna TRUE se esp è uguale a NIL, altrimenti NIL.
- **EQ**($esp1, esp2$) Ritorna TRUE se le due espressioni sono uguali, NIL altrimenti

Tutte le funzioni ritornano NIL se incontrano problemi (ad esempio, se l'elemenot che dovrebbero ritornare non esiste).

Il problema consiste nello scrivere un algoritmo genetico che componga queste istruzioni per creare un programma che, se dato da eseguire al nostro robot, consenta al robot di scrivere la parola "universal" sullo stack, per qualsiasi configurazione iniziale delle lettere. In questo linguaggio, un programma corretto consiste in un set di comandi annidati (con una sola radice, che per convenzione viene scelta essere il comando EQ), dove ogni comando contiene il numero e il tipo corretto di argomenti. I primi tre comandi (detti "sensori") non accettano argomenti, i restanti prendono come argomento i singoli sensori oppure un set annidato di altri comandi, nel numero oportuno. Si nota che per forza di cose, i comandi "in fondo" al programma sono sensori.

2 L'APPROCCIO

Il problema è stato impostato in questo modo nel linguaggio di programmazione Mathematica. La pila e il tavolo sono stati rappresentati come variabili globali, nello specifico come liste contenenti le lettere della parola "universal".

I comandi del robot sono stati implementati come funzioni di Mathematica.

Per quanto riguarda gli individui dell'algoritmo genetico, sono state percorse due strade.

La prima, la più semplice da implementare e sensibilmente più performante, sfrutta le potenzialità di Mathematica, ossia la possibilità di trattare ogni programma (un set di comandi annidati) come una lista annidata. Questo è vantaggioso perchè Mathematica mette a disposizione comandi ad alto livello per la manipolazione di liste ed alberi.

In questo modo un individuo non è altro che un espressione di Mathematica del tipo $C1[C2[], C3[..]]$, dove $C1, C2, C3...$ sono le funzioni corrispondenti ai comandi del robot. È tuttavia necessario trovare un modo per bloccare la valutazione di questa espressione fino al momento opportuno, dal momento che Mathematica valuta le espressioni appena ne ha la possibilità.

Mathematica mette a disposizione dei comandi per controllare la valutazione delle espressioni, come ad esempio $Hold$, gli attributi $HoldAll$ e simili, e altri comandi introdotti nelle versioni più recenti. È stato scelto di procedere in maniera meno ortodossa: ogni individuo è stato scritto utilizzando non le funzioni del robot definite in precedenza, ma funzioni "vuote" (non definite) con nomi simili; in questo modo l'espressione viene lasciata immutata (l'interprete non può valutare niente, perché le funzioni chiamate non sono definite), finché non viene usato il comando "ReplaceAll" per sostituire i nomi fittizi con le funzioni corrette, il che consente a Mathematica di procedere con la valutazione del programma

Si nota che i comandi specifici per il controllo della valutazione sono stati usati in altri punti del programma.

Come secondo approccio, per scopi puramente didattici, si è scelto di costruire e manipolare gli individui senza ricorrere alle potenzialità di Mathematica, ossia rinunciando alla possibil-

ità di accedere ai livelli di una lista annidata e perfino senza conoscere a priori le dimensioni degli insiemi usati. Nonostante si sia cercato di costruire algoritmi efficienti, questi sono sensibilmente meno efficienti dei corrispondenti creati sfruttando Mathematica.

Un individuo di questo tipo è una lista con struttura ad albero, dove gli elementi al livello più basso (foglie) sono le funzioni rche realizzano i sensori, ad esempio `Tb[] &`; l'applicazione di una funzione ad una espressione si realizza mettendo nella prima posizione di una lista la funzione, e nelle restanti gli argomenti, ad esempio: `EQ(CS NN) -> {Eq[#1,#2] &, Cs[] &, Nn[] & }` .

A titolo di esempio, viene riportata la realizzazione di una possibile soluzione al prolema.

Questa struttura si presta maggiormente ad una generalizzazione, sia nel linguaggio (è più facilmente implementabile utilizzando linguaggi con meno funzionalità di Mathematica), sia nelle funzionalità, in quanto è possibile rappresentare un generico albero e risolvere così una ampia classe di problemi.

È stata creata una variabile globale per contenere il numero di passi effettuati dall'esecuzione di un individuo. In questo modo, oltre a misurare l'efficienza di un programma, è possibile bloccare l'esecuzione dei programmi che impieghino troppi passi, ed evitare loop infiniti, nonché garantire la creazione di una soluzione ragionevolmente efficiente.

Si è scelto di affidare l'incremento di questa variabile e il controllo sul numero di passi alle singole funzioni elementari. Questo perché, come viene mostrato nella sezione riguardante l'esecuzione degli individui, a causa dell'istruzione DU, è l'unico modo per escludere con certezza la possibilità di loop infiniti.

2.1 Generazione casuale degli individui iniziali

In entrambi i casi, gli individui vengono generati tramite ricorsione. Prendiamo in considerazione il solo caso del secondo approccio (liste di funzioni).

La funzione generatrice parte creando una lista di tre elementi, dove nella prima posizione viene messa la funzione `Eq[] &`, e nelle altre due viene chiamata una funzione che, in maniera random decide se ritornare un sensore, oppure ritornare una delle funzioni, e chiamare di nuovo se stessa per decidere l'argomento della funzione.

In questo modo, è possibile creare uniformemente programmi random, senza però vincoli sulla lunghezza. Se non si è interessati ad avere programmi particolarmente brevi, per controllarne la lunghezza è sufficiente scartare gli individui che superino un certo numero di elementi totale (il numero dato da `Length[Flatten[individuo]]`

2.2 Esecuzione di un individuo

Per gli individui del tipo 1, è sufficiente effettuare un `ReplaceAll`: `/.{ nomefintoTB -> Tb, ...}`

A questo punto il programma diventa un'espressione valutabile da Mathematica

Gli individui del tipo 2 vengono convertiti in individui del tipo 1 tramite una funzione ricorsiva `TreeToExp`, che funziona in questo modo: presa una lista che corrisponde all'esecuzione di una funzione `{funzione , argomento1, argomento2 }`, ritorna il primo elemento applicato all'esecuzione di `TreeToExp` agli elementi successivi: `funzione[TreeToExp[argomento1], TreeToExp[argomento2]`. `TreeToExp` Applicata ad una foglia ritorna `nomefintosensore`

È interessante notare che, se non fosse per la presenza dell'istruzione DU, sarebbe immediato valutare un individuo di questo tipo senza dover passare per questa conversione; è sufficiente modificare la funzione `TreeToExp` in modo che valuti i sensori invece che sostituirli con espressioni da valutaere. In presenza di istruzioni che richiedono di valutare più di una volta le espressioni passate come argomento, non c'è un modo sempice per fare questo, neanche utilizzando le funzioni per il controllo della valutazione.

Di fatto, l'istruzione DU crea problemi anche nell'approccio 1; per garantire un'esecuzione corretta sono state necessarie alcune accortezze, fra cui l'uso dell'attributo `HoldAll` nella funzione DU.

2.3 Fitness

In maniera simile all'implementazione originaria di Koza, abbiamo scelto, per ogni esecuzione dell'algoritmo di generare un insieme di configurazione iniziali per le lettere (dell'ordine del centinaio di elementi), quasi tutte random tranne due casi particolari fissati per tutte le esecuzioni (una configurazione in cui le lettere sono già nell'ordine desiderato e una in cui tutte le lettere sono sul tavolo).

Ogni programma viene eseguito per tutte le configurazioni iniziali, e la funzione di fitness ritorna il numero di configurazioni per cui il programma.

È stata implementata anche un'altra funzione di fitness che, dopo un esecuzione del programma con una delle configurazioni iniziali, assegnava un punto per ogni lettera ordinata nella pila, più un bonus se tutte le lettere venivano ordinate correttamente. Il punteggio totale è la somma di questo per tutte le configurazioni iniziali, come nel caso precedente.

Questa funzione è stata introdotta perchè, con un campione di configurazioni e/o individui limitato, sarebbe stato ragionevole supporre che la fitness originaria non fosse in grado di evidenziare differenze fra gli individui (generando un programma random, è improbabile che questo riesca ad ordinare una configurazione).

Tuttavia, l'utilizzo della fitness alternativa non ha migliorato il funzionamento del programa.

2.4 Crossover e mutazione

Il metodo di Crossover usato, lo stesso in entrambi gli approcci, è lo stesso usato da Koza.

Una mutazione viene realizzatta generando un albero casuale e sostituendolo ad un ramo

dell'individuo scelto casualmente.

Per generare una coppia di figli a partire da due individui, si seleziona un ramo casuale dal primo individuo, e lo si sostituisce ad un ramo casuale del seconde; viceversa per il secondo figlio.

Tuttavia, l'implementazione di questo meccanismo è radicalmente diversa nei due approcci.

Nel primo approccio, ottenere casualmente un ramo da un individuo è immediato; in Mathematica, l'ultimo di una lista contiene, nel nostro caso, tutti i rami dell'albero, presi una sola volta. È suficiente scegliere un elemento casuale di questo insieme.

...

Nel caso del secondo approccio, la manipolazione degli alberi è meno immediata, dal momento che l'unica funzione che abbiamo a disposizione è la funzione `Part`, che consente di accedere agli elementi di una lista. Se si conosce il numero di nodi dell'albero, un modo per affrontare il problema potrebbe essere di decidere un modo di indicizzare l'albero, e trattare l'albero come una lista, costruendo opportune funzioni per accedere agli elementi con l'indicizzazione prescelta.

Nel nostro caso è stato scelto un algoritmo del tipo *Reservoir Sampling* che non richiede di conoscere la dimensione dell'albero, che garantisce in ogni caso un tempo di esecuzione $O(N)$.

Per selezionare un ramo casuale da un individuo, l'algoritmo percorre ricorsivamente tutti i nodi dell'albero, mettendo il nodo corrente in una fissata variabile temporanea (il deposito) con probabilità $1/n$, dove n è il numero corrente di passi. Alla fine viene restituita la variabile temporanea. Facendo i conti, si trova che ogni ramo dell'albero ha la stessa probabilità di essere scelto.

Si procede in maniera analoga per inserire un ramo in un punto casuale di un individuo dato, con la differenza che in questo caso bisogna avere qualche accortezza in più con la variabile di deposito, dal momento che tenere in memoria una posizione dell'albero (che non è indicizzato) non è un problema banale.

(eventuale ...)

2.5 Riproduzione

La prima generazione è composta da n_{ind} individui generati casualmente, di lunghezza contenuta. A partire da questa vengono scelti i genitori secondo il criterio *fitness proportionate*, ossia viene creata una lista di $n_{ind} - 2$, e viene riempita scegliendo elementi random della prima generazione, con probabilità proporzionale alla loro fitness. Si sarebbero potuti prendere in considerazione anche altri criteri per la selezione dei genitori, ma dati i ragionevoli tempi di esecuzione dell'algoritmo, non è stato ritenuto necessario.

Questa lista di elementi (i "genitori") viene scorsa a coppie, e ogni coppia con una certa probabilità p_c viene sostituita dalla coppia dei figli, ottenuti con il metodo descritto in precedenza.

Questo processo genera $n_{ind} - 2$ nuovi individui, per cui è necessatio agiungere altri due programmi. Questi vengono scelti essere i due individui con la fitness più alta nella generazione precedente. Questa modifica è molto importante per garantire una rapida convergenza dell'algoritmo. Esecuzioni del progamma senza questa caratteristica non convergevano in tempi ottimali (<100 generazioni)

Successivamente a tutti gli individui viene applicata una mutazione, ossia con una propabilità pari a p_m un ramo dell'individuo scelto casualmente viene sostituito con un ramo generato casualmente.

3 Risultati

Per prima cosa, è stato necessario determinare il valore dei parametri dell'algoritmo, p_c e p_m.

Questi sono stati determinati empiricamente, eseguendo il programma con diverse combinazioni dei parametri in un intervallo ragionevole (sappiamo che nella maggior parte dei casi, un algoritmo genetico funziona in maniera ottimale quando $p_c > 0.7$ e $p_m \sim 0.01$).

Il tempo di esecuzione del programma non è ovviamente deterministico, ma è soggetto a fluttuazioni statistiche. Non sono stati eseguite sufficienti esecuzioni da permettere una statistica rigorosa su queste fluttuazioni e di conseguenza una stima precisa delle performance del programma. Considerati i tempi di esecuzione (dell'ordine di $1 \div 10$ ore per ogni esecuzione), una statistica rigorosa avrebbe richiesto una quantità rilevante di tempo e risorse.

Dai dati raccolti, risulta che il programma converge più velocemente per bassi valori di p_c ($p_c \sim 0.6$) e alti valori di p_m ($p_m \sim 0.2$). Questo potrebbe significare che è il caso, più che l'evoluzione, a dettare la comparsa di soluzioni.

Tuttavia, la convergenza viene ottenuta dopo un numero di generazioni simile per tutti i parametri (~ 20 per le configurazioni ottimali, ~ 30 per quelle non ottimali). Inoltre in ogni singola esecuzione la fitness di ogni generazione è rigorosamente crescente, anche se non sempre convergente al massimo.

Per ogni valore dei parametri, l'algoritmo usualmente converge prima della quarantesima generazione.

4 Estensioni

Come già anticipato, la struttura del programma è molto generale, e facilmente estensibile a diverse classi di problemi.

Ad esempio, lo stesso Joza ha usato questo tipo di algoritmo per risolvere una classe di problemi detti problemi di *planning* . . . .