Relazione sul progetto "Algoritmi Genetici" con Mathematica

Luigi Torino
Matricola n.° 955396

Anno accademico 2019-2020

Indice

Introduzione - Algoritmi genetici

Un algoritmo genetico è un algoritmo di ottimizzazione euristico appartenente alla classe degli algoritmi evolutivi. Tali algoritmi si ispirano al principio della selezione naturale e dell'evoluzione e attuano dei meccanismi concettualmente simili a quelli dei processi biologici, utilizzando elementi come geni, DNA, cromosomi, selezione, crossover e mutazioni.

Un algoritmo genetico non garantisce in tutti i casi di trovare una soluzione globale ottimale, ma può arrivare ad una soluzione accettabile entro certi limiti in un tempo relativamente breve rispetto ad altri algoritmi di ricerca. In alcuni casi può essere l'unica via percorribile per trovare una soluzione a problemi non adattabili ai classici metodi matematici di ottimizzazione, per esempio nei casi in cui lo spazio delle soluzioni è troppo grande per essere analizzato in modo esauriente, se esistono molti massimi/minimi locali nella funzione che valuta le soluzioni, o se si ha un'idea qualitativa di "soluzione ottimale" ma non una procedura nota a priori per determinarla.

Il punto di partenza di ogni algoritmo genetico è una popolazione iniziale di *organismi* o *individui*, che vengono combinati a coppie per formare dei *figli*, ossia una nuova *generazione*. Nel processo di riproduzione i *cromosomi* / il *DNA* dei genitori vengono mescolati e passati ai figli (*crossover*), e a loro volta i figli possono ulteriormente venire modificati da *mutazioni* casuali.

I nuovi individui ereditano le caratteristiche dei genitori ma sviluppano anche nuovi tratti; successivamente si ha un processo di selezione basato sulla *fitness* (funzione a valori reali definita sullo spazio delle soluzioni), che classifica la popolazione in modo da poter scegliere per la riproduzione i migliori individui. Nel caso la fitness abbia una forma funzionale complicata, un AG può avere delle chances più alte di trovare soluzioni soddisfacenti e in modo efficiente rispetto ad altri metodi.

Nel nostro caso di studio sono state sfruttate la potenza e la versatilità degli algoritmi genetici per la scrittura automatizzata di programmi adatti a risolvere il problema del block-stacking.

Analisi del problema

Il problema del block-stacking è stato illustrato in dettaglio da Koza (1992). L'obiettivo è trovare un programma che riceve in input una certa configurazione iniziale di blocchi (lettere) distribuiti tra due variabili/liste dette *stack* e *table* e facenti parte di una parola (da me assegnata all'interno dell'algoritmo alla variabile *wanted*). Il programma deve riordinare tutti i blocchi, spostandoli sullo stack e nell'ordine corretto per formare *wanted*. Nell'esempio affrontato da Koza la parola cercata era "universal"; ho scelto invece di utilizzare la parola "universale" per testare l'algoritmo su una parola contenente lettere ripetute. Le funzioni utilizzate sono un set di sensori e comandi. I sensori (tre) restituiscono una lettera:

- CS ("current stack") restituisce il contenuto del blocco in cima alla stack, se la stack è vuota restituisce Null;
- TB ("top correct block") restituisce il blocco dello stack posizionato più in alto tale che esso stesso e tutti i blocchi sottostanti sono in ordine corretto. Se nessun blocco corrisponde a questi criteri, TB restituisce Null;
- NN ("next needed") restituisce il blocco immediatamente successivo a TB nella parola cercata. Se non servono ulteriori blocchi restituisce Null.

I comandi (cinque) invece sono:

- MS(X) ("move to stack") muove il blocco X in cima alla stack se X è nella table, e restituisce x;
- MT(X) ("move to table") muove il blocco in cima alla stack sulla table se il blocco X è presente nello stack (in qualsiasi posizione), e restituisce X;
- DU(expression1, expression2) ("do until") valuta ripetutamente il primo argomento finché il secondo argomento non restituisce True;
- NOT(expression1) restituisce True se l'argomento è uguale a Null, altrimenti restituisce Null;
- EQ(expression1,expression2) restituisce True se gli argomenti sono uguali (hanno lo stesso valore/restituiscono lo stesso risultato).

AG vs Random Search

Ci si potrebbe chiedere: qual è il vantaggio di utilizzare un algoritmo genetico per la ricerca della soluzione invece di una **random search**? Una ipotetica random search consisterebbe nel generare una popolazione iniziale ampia almeno quanto popolazione iniziale dell'AG × generazione media valutata su vari test di AG, e composta da individui con una Depth (quindi con un numero di comandi) fissata a un certo valore (anche qui, riprendendolo da quello di un ipotetico AG). É opportuno fissare una certa profondità limite per evitare di generare individui eccessivamente lunghi e prolungare i tempi della random search in modo eccessivo. Si devono quindi testare i programmi generati e verificare se almeno uno di essi rappresenta la soluzione del problema, utilizzando approssimativamente gli stessi criteri di valutazione dell'AG.

Un altro metodo potrebbe essere generare un solo individuo e testarlo, continuando con questo *trial and error* fino a che non si trova una soluzione o si arriva a un certo numero di iterazioni massimo (serve sempre impostare un limite per evitare cicli fuori controllo).

Il vantaggio di questo approccio potrebbe essere dato dal fatto di trovare casualmente una soluzione in un tempo breve; ma sarebbe appunto un caso, mentre altre molteplici prove potrebbero non trovare nessuna soluzione. Gli svantaggi invece sono molteplici:

- Non si conosce a priori il numero di individui da generare nè il numero massimo di iterazioni da impostare, si può solo andare a tentativi;
- Non è garantito trovare sempre una soluzione;
- Il tempo impiegato è paragonabile a quello di un AG che genera e valuta lo stesso numero di individui;
- Rispetto a un algoritmo genetico si deve espressamente impostare tutta una serie di accorgimenti per generare una eventuale soluzione ottimizzata, mentre nell'algoritmo genetico tutto ciò è automatizzato semplicemente includendo tali criteri nella fitness.

L'approccio genetico è quindi molto più indicato per la risoluzione di questo problema in quanto con pochi semplici criteri permette di trovare una soluzione molto più facilmente rispetto ad una random search, a parità di prove e date le stesse condizioni iniziali, e con tempi computazionali in media più brevi.

La scelta di Mathematica

Mathematica si presta particolarmente bene alla risoluzione di questa task. Si possono generalizzare e automatizzare molte parti del problema grazie alle caratteristiche intrinseche di Mathematica come la riscrittura di espressioni e il supporto svariati paradigmi di programmazione; tra di essi, la programmazione funzionale, la programmazione basata sul riconoscimento di schemi (pattern-matching) e sulle regole di sostituzione, la programmazione procedurale. L'approccio procedurale per un problema di questo tipo, strettamente legato alla manipolazione di espressioni annidate / in forma di albero e costituite da elementi simbolici, è più complicato e meno efficiente delle alternative

implementate di default in Mathematica (basta pensare alla sola funzione Map o alle regole di sostituzione).

La struttura dati fondamentale di Mathematica è l'**espressione**. Ogni individuo è quindi rappresentato in Mathematica da un'espressione con una sua struttura ben definita; ogni sua parte è classificabile grazie all'intestazione (**Head**) e dalla sequenza di argomenti racchiusi tra parentesi quadre e separati da virgole successivi alla Head. Questa classificazione è molto utile nell'approccio genetico nel momento in cui si devono attuare le operazioni di generazione degli individui, crossover e mutazione.

Controllo dei loops e della valutazione

Ritornando al problema vero e proprio, si può notare che EQ funge da connettivo logico tra altre due espressioni, mentre NOT può essere utilizzato come controllo per verificare che una certa condizione sia rispettata o meno. Il comando DU rappresenta invece il vero e proprio cuore di ogni individuo/programma, in quanto è l'unico esempio di iterazione disponibile all'interno del set di funzioni date. Poiché Mathematica esegue la valutazione di un'espressione non appena la incontra, ho assegnato al comando DU l'attributo **HoldAll**; viene così bloccata la valutazione degli argomenti (che altrimenti sarebbero stati passati al DU con il loro valore e non in forma simbolica). La valutazione viene in seguito effettuata in maniera dinamica tramite la funzione **ReleaseHold** presente nel ciclo While interno al DU.

Oltre al controllo sulla valutazione è opportuno inserire un limite al numero massimo di cicli che possono essere svolti sia all'interno di un unico DU sia da un singolo individuo. Si evitano così cicli infiniti e valutazione completa di individui troppo lunghi, con un conseguente risparmio di tempo e aumento dell'efficienza dell'AG. Mi è sembrata una scelta ragionevole porre il limite di iterazioni per un singolo DU, definito come ***maxciclenumber***, pari a 3 × (lunghezza di *wanted*); ho fatto invece variare il limite di iterazioni svolte da un singolo individuo come un multiplo di *maxciclenumber* compreso tra 4 e 7, a seconda del caso di studio, trovando nel valore 4 la scelta più soddisfacente. In caso il numero di iterazioni superi uno o entrambi i limiti, il ciclo While del DU viene interrotto, i *side effects* (ossia le azioni già svolte su stack e table) vengono mantenuti ma il DU restituisce Null.

Popolazione iniziale, correttezza sintattica

La popolazione di partenza deve essere generata in modo tale che i suoi individui, alla pari degli argomenti del DU, non vengano valutati da Mathematica nel momento in cui vengono creati e siano "statici" rispetto a operazioni quali mutazione e crossover. Ho quindi scelto di inizializzare qualsiasi individuo partendo dallo schema *Hold[Evaluate[...]]*, che permette di costruire il resto dell'individuo estendendo la sua struttura ad albero con nuovi comandi fino a una certa profondità iniziale fissata. Nel conteggio della profondità iniziale ho tenuto conto della depth aggiuntiva dovuta alla stessa espressione *Hold[Evaluate[...]]*.

Nella costruzione dell'individuo ho inoltre impiegato comandi "dummy", ossia dei comandi senza implementazione che avevano la funzione di segnaposto per i comandi veri e propri; ad esempio, al posto di NN ho usato xNN, xCS al posto di CS e così via. Tali comandi verranno poi sostituiti dagli originali quando gli individui vengono

valutati nella fitness: ho infatti implementato una funzione *Esegui* che sostituisce i comandi "dummy" con i comandi normali e rimuove gli Hold presenti nell'individuo (partendo dalle "foglie", ossia dai livelli più interni).

Un altro aspetto di cui ho tenuto conto è la cosiddetta **'correttezza sintattica'**, vale a dire il controllo di quali argomenti vengono inseriti dentro ciascun tipo di funzione. Grazie a questo accorgimento è possibile generare programmi sensati, evitando di produrre espressioni che non hanno alcuna utilità proprio per il modo in cui sono state scritte. Per esempio, non avrebbe senso inserire come secondo argomento del DU un sensore; il secondo argomento del DU viene infatti valutato finché non restituisce True, e un sensore restituisce unicamente una lettera o Null. In questo modo vengono migliorati i tempi computazionali, la correttezza e l'utilità dell'intero processo genetico.

Parametri delle run

L'avanzare delle generazioni fino al ritrovamento della soluzione costituisce una 'run' dell'algoritmo genetico. I parametri iniziali della run sono:

- $Npop$ = numero di individui della popolazione iniziale, fissa il pool di candidate soluzioni all'interno dell'algoritmo per ogni generazione;
- $Ngen$ = numero di massimo di generazioni fino alla quale ogni run si può spingere, in quasi tutte le prove è fissato a 200;
- pc = probabilità di crossover, ho variato i valori tra 0.7 e 0.8 durante le varie prove;
- pm = probabilità di mutazione, ho alternato i valori 0.1 e 0.15 durante le varie prove.

Correttezza delle soluzioni

Trovata la soluzione, ho deciso di testarla su un certo numero di stack e table generate casualmente (solitamente 200). Ho verificato che alcune soluzioni garantiscono di risolvere qualsiasi coppia di stack e table venga loro passata; la loro correttezza percentuale, a prescindere dal numero di test, è 100.

Per altre soluzioni la correttezza è inferiore al 100%; ciò potrebbe essere dovuto ad aver trovato una soluzione specifica per le coppie di stack e table iniziali, contenente istruzioni non adatte a risolvere una certa classe di configurazioni sulla quale la soluzione non era stata testata nel corso della run. La correttezza media delle soluzioni trovate all'interno di un gruppo di run può essere aumentata a discapito del tempo di esecuzione delle stesse, impostando un numero più ampio di stack e table di test.

Fitness

La fitness traduce le caratteristiche di ogni candidato in un numero reale, valutandolo in base a certe caratteristiche generali: patrimonio genetico, correttezza dei singoli stack, correttezza globale. In questa sezione mi riferirò, per brevità e utilizzo frequente, alla lunghezza della parola cercata come $Lenght_W$.

Voti

Il **patrimonio genetico** non è altro che il conteggio del numero di comandi differenti presenti nell'individuo; si suppone che un programma con una certa quantità di funzioni al suo interno sia favorito nel trovare la soluzione rispetto ad un altro programma molto più scarno, o che quantomeno un programma del primo tipo riesca a passare ai programmi figli dei blocchi di comandi abbastanza vari e utili nel raggiungere l'obiettivo. Ho scelto di assegnare a ogni individuo un voto per il patrimonio genetico pari a:

$$\mathrm{v}_{patr} = 5 \times (\text{numero di comandi diversi}) \times (\text{numero di stack iniziali})$$

Per questo problema, anche per parole da riordinare molto corte, vi sono tantissime **configurazioni possibili** di stack e table iniziali. É quindi necessario scegliere all'inizio del programma un numero di configurazioni definito, campionato tra tutte le possibili disposizioni, per valutare la fitness su di esse in un tempo ragionevole. La scelta dovrebbe ricadere su almeno 4-5 stack iniziali (e conseguenti table iniziali) di tipo diverso:

- Una o due stack composte solo da $N \leq Lenght_W$ lettere ordinate nel modo in cui compaiono in *wanted* (eventualmente si può anche scegliere una stack completamente ordinata);
- Una stack composta da $N < Lenght_W$ lettere in ordine e altre $M < (Lenght_W - N)$ lettere in disordine;
- Una stack composta da $N < Lenght_W$ lettere in disordine;
- Una stack composta da $N = Lenght_W$ lettere in disordine;
- Una stack vuota.

Ogni individuo va valutato sulle coppie stack-table di volta in volta; per ogni valutazione ho assegnato un voto corrispondente alla **correttezza del singolo stack** dato da:

$$\mathrm{v}_{corr}\,(\text{coppia}) = 10 \times \text{indiceTCB}$$

in cui l'indice TCB corrisponde all'indice dell'ultimo blocco corretto dello stack dopo che l'individuo ha agito su stack e table (corrispondente in un certo senso a ciò che fa il sensore TB). Il voto totale per la correttezza $\mathrm{v}_{corr\ (\mathrm{tot})}$ è la somma dei voti sulle coppie; vengono quindi premiati gli individui che generano più stack corretti.

Per ottimizzare la soluzione ho inserito anche un voto da assegnare agli individui che riescono a costruire più stack corrette 'in parallelo'; così facendo si evita di premiare in modo eccessivo individui che ricostruiscono solo certi tipi di stack (solo con lettere già in ordine, solo con lettere in disordine, ecc.) e si tende ad arrivare a una soluzione che risolve la maggior parte delle stack che le vengono sottoposte. Avendo calcolato $\mathrm{v}_{corr\ (\mathrm{tot})}$ conosciamo già gli indiciTCB di tutti gli L stack; il voto che ho assegnato a questa **'correttezza parallela'** è:

$$\mathrm{v}_{parall} = 20 \times \frac{\sum_{i=1}^{L} \text{indiceTCB}_i}{max - min + 1};$$

dove max corrisponde all'indiceTCB$_i$ più grande, min al più piccolo. Se una candidata soluzione ricostruisce soltanto uno stack correttamente e gli altri stack in misura minore, allora il denominatore $max - min + 1$ sarà grande; se lo scarto tra max e min è piccolo allora il denominatore sarà vicino a 2 e il voto assegnato sarà maggiore.

Il voto finale assegnato ad ogni individuo dopo la valutazione tramite la fitness è:

$$\mathrm{v}_{TOT} = \mathrm{v}_{patr} + \mathrm{v}_{corr\ (\mathrm{tot})} + \mathrm{v}_{parall}$$

Ottimizzazione

Un'ennesima valutazione / ottimizzazione svolta all'interno della funzione di fitness è la **limitazione degli individui eccessivamente lunghi**; una soluzione di tipo iterativo non dovrebbe contenere troppi comandi o comandi annidati in un numero di livelli molto grande. Per gli individui con una Depth superiore a 8 la fitness non viene valutata, ma viene loro assegnato un voto pari al voto per il patrimonio genetico; in questo modo non vengono totalmente gettati via nel processo di selezione ma possono contribuire a passare pezzi del loro codice alle generazioni successive.

Infine, se un individuo viene valutato su tutti gli stack iniziali e li ordina correttamente, il processo di valutazione non viene più svolto per la generazione in corso e l'individuo viene salvato come **soluzione**; la run si interrompe.

Risultati

Fitness proportionate vs Boltzmann selection

Una delle scelte da compiere durante la scrittura dell'algoritmo è il tipo di selezione da implementare per decidere quali individui far riprodurre. Il tipo di selezione più semplice è la *fitness-proportionate*, un metodo probabilistico in cui la *raw fitness* di ogni individuo è usata per associare ad esso la probabilità di essere estratto come genitore. Se f_i è la fitness dell'individuo i della popolazione, la probabilità di essere selezionato è data dalla fitness normalizzata sul totale delle fitness:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Con la selezione fitness-proportionate la probabilità che gli individui più 'deboli' sopravvivano al processo di selezione è minima ma non nulla; ciò può essere un vantaggio poiché anche gli individui peggiori possono avere delle caratteristiche utili nei successivi processi di ricombinazione.

Un altro metodo di selezione è la cosiddetta 'Boltzmann selection', un approccio simile al simulated annealing che abbiamo incontrato nel corso di Laboratorio di Simulazione Numerica. Implementando tale selezione si ha una pressione selettiva adattiva, meno stringente per le generazioni iniziali e gradualmente più alta con l'avanzare del tempo. L'idea è di variare in modo continuo e automatico un parametro simile alla "temperatura del sistema", il quale controlla il rate di selezione. La temperatura varia da un massimo iniziale, corrispondente a una situazione in cui la pressione riproduttiva è bassa e ogni individuo ha una probabilità di riprodursi comparabile a quella degli altri, a valori sempre più piccoli. Con la decrescita di T la pressione viene gradualmente aumentata e l'algoritmo conserva soltanto gli individui migliori pur mantenendo un certo grado di varietà nella popolazione.

Data la fitness f_i di un individuo i a un certo tempo t, in cui il sistema è a temperatura T, la sua probabilità di estrazione sarà:

$$p_i = \frac{e^{f_i/T}}{\langle e^{f_i/T} \rangle_t}$$

Il denominatore corrisponde alla media sulla popolazione della quantità tra parentesi al tempo t. Al diminuire di T, la differenza tra le p_i di individui con fitness alta e le p_i di individui con fitness bassa è significativamente maggiore rispetto alle generazioni

iniziali, dove la temperatura era più elevata. La variazione di T deve essere graduale nel tempo, per riuscire a selezionare in modo progressivo gli individui migliori ma senza arrivare a una convergenza prematura dell'algoritmo.

Approcciandomi per la prima volta agli AG e a Mathematica, ho utilizzato prevalentemente il criterio fitness-proportionate nelle varie prove. In ultima istanza ho provato a implementare una selezione mista tra fitness-proportionate e Bolztmann selection. Mentre la convergenza per l'approccio fitness-proportionate deve essere opportunamente garantita calibrando i parametri della run, con la selezione mista basta impostare una decrescita lenta di T e una temperatura iniziale adeguata. Di seguito sono riportati alcuni grafici significativi per entrambi i criteri di selezione; i parametri delle run sono uguali per i due casi.

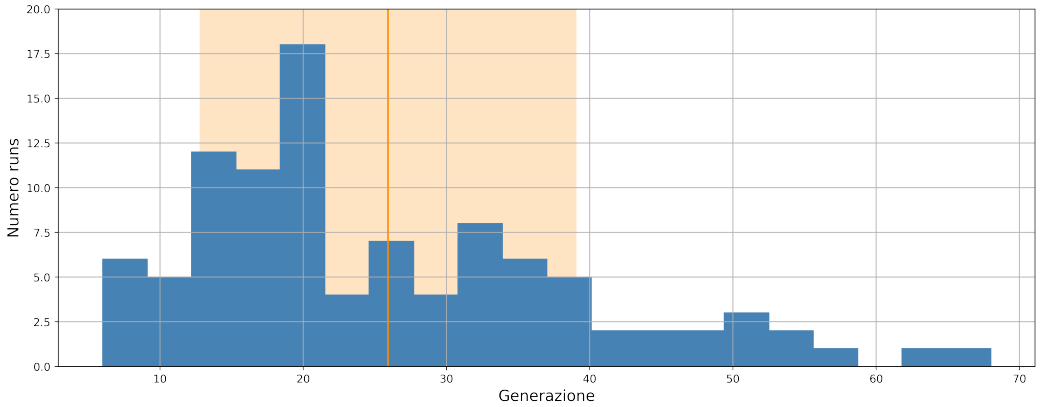

Figura 1: Istogramma con 20 bins delle generazioni raggiunte da 100 run con un approccio fitness-proportionate; in evidenza generazione media e deviazione standard.

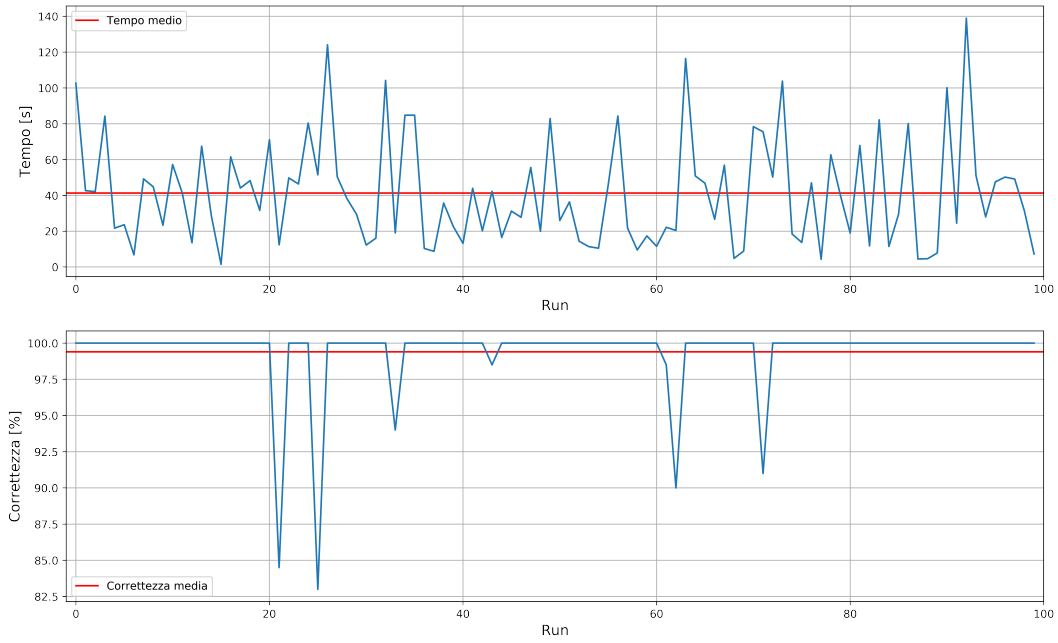

Figura 2: Tempi computazionali e correttezza media delle soluzioni per ogni run utilizzando un approccio fitness-proportionate.

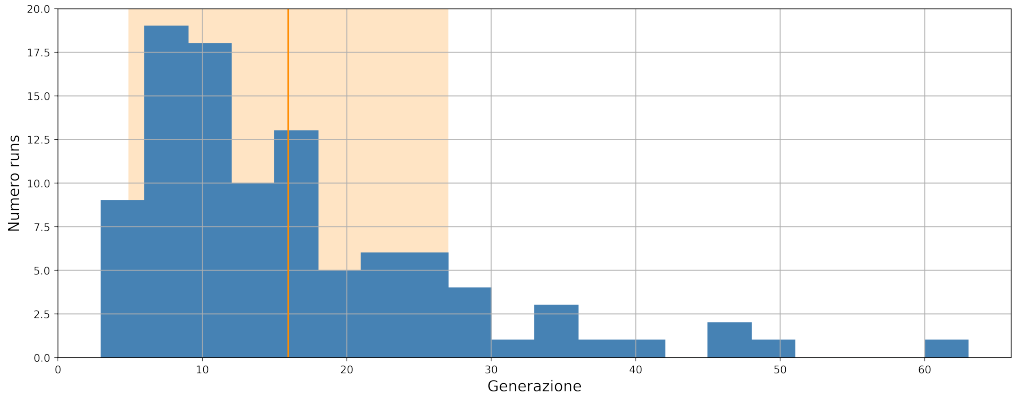

Figura 3: Istogramma con 20 bins delle generazioni raggiunte da 100 run con un approccio misto fitness-proportionate / Boltzmann selection; in evidenza generazione media e deviazione standard.

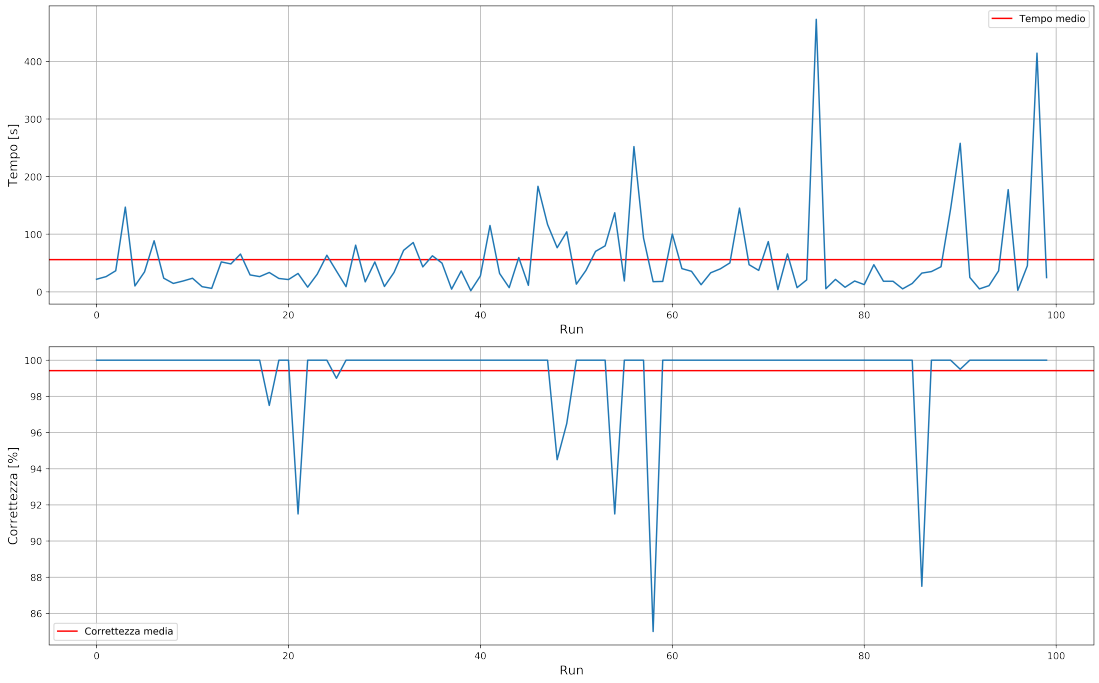

Figura 4: Tempi computazionali e correttezza media delle soluzioni per ogni run con un approccio misto fitness-proportionate / Boltzmann selection.

La convergenza entro 100 generazioni risulta sempre garantita per entrambi i casi (nella maggior parte delle prove non si supera la generazione 50); l'approccio misto converge decisamente prima di quello FP (la generazione media è 16 $\pm$ 11 rispetto a 26 $\pm$ 13 del secondo approccio) ma i tempi computazionali medi per run sono leggermente più lunghi di quelli ottenuti usando la selezione fitness-proportionate (55.81 $\pm$ 73.95 sec contro 41.26 $\pm$ 30.14 sec). La deviazione standard dei tempi è invece più alta, il suo valore è addirittura maggiore del valore medio, indicando un comportamento temporale fortemente variabile. La correttezza media delle soluzioni è pressoché uguale (99.42 $\pm$ 2.33 con approccio misto, 99.40 $\pm$ 2.67 con fitness-proportionate), indicando una forte dipendenza dal numero e dal tipo di stack e table di prova (in questo caso, appunto, identici).

Fitness della popolazione, generazione media

Andando avanti con le generazioni, la fitness media e la fitness massima della popolazione tipicamente aumentano (figura 5). I migliori individui di ogni generazione gestiscono correttamente sempre più blocchi dei vari stack di prova; allo stesso tempo la fitness media tende a crescere in quanto le successive popolazioni sono costituite in misura sempre maggiore da individui con fitness alta.

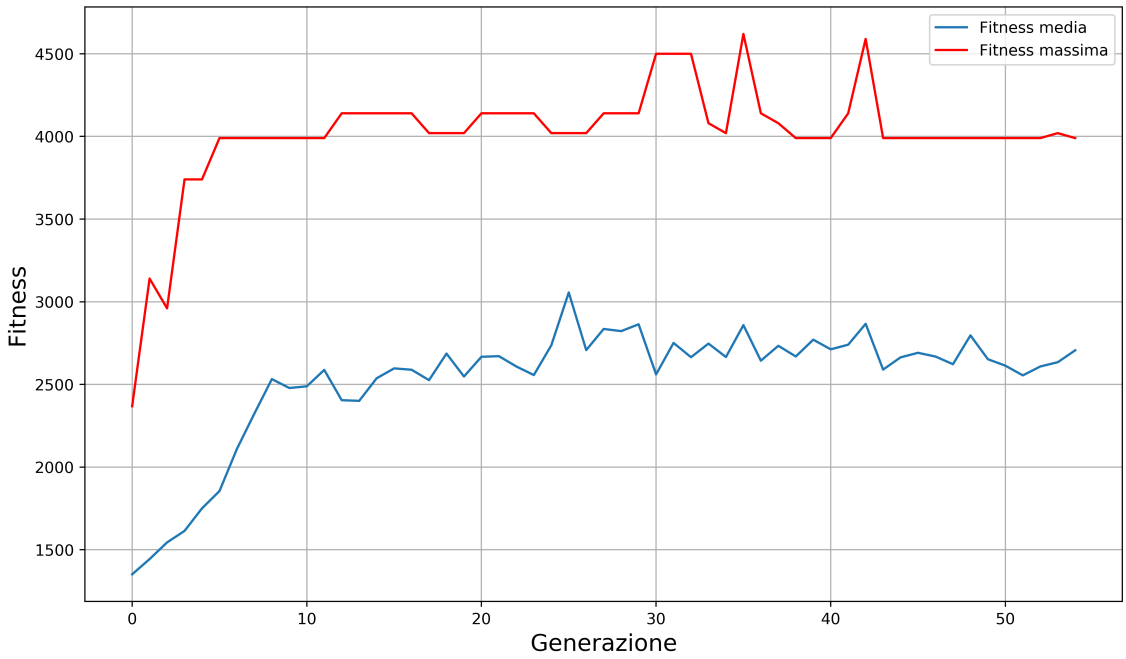

Figura 5: Fitness media e massima di una run.

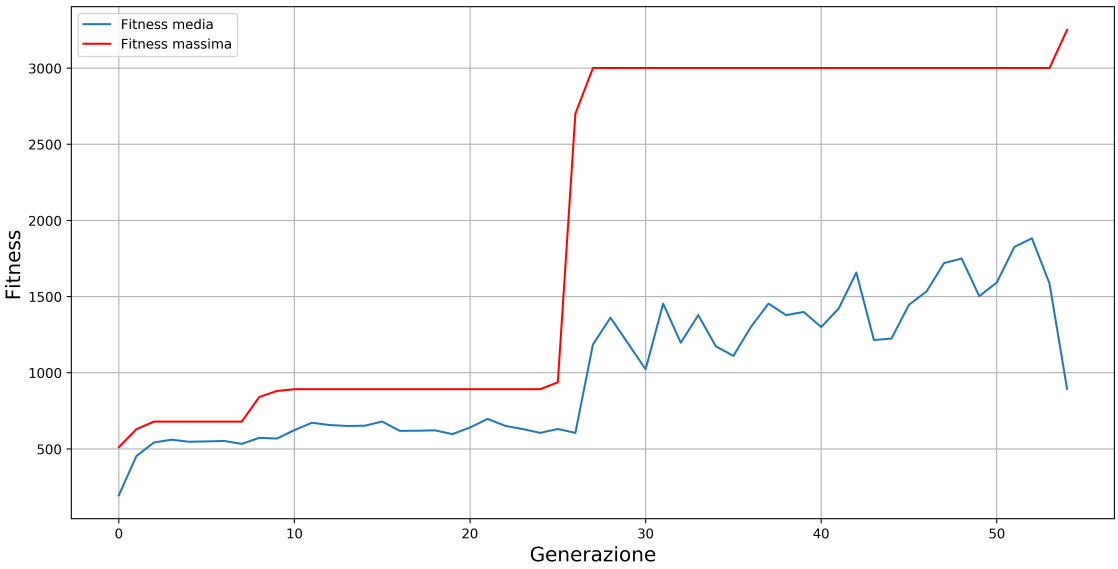

Figura 6: Fitness media e massima di una run per la Boltzmann selection.

La generazione media a cui viene trovata una soluzione dipende dalla dimensione della popolazione iniziale, dalla probabilità di crossover e dalla probabilità di mutazione.

Aumentando la popolazione iniziale fino a 150-200, l'algoritmo converge prima, in quanto è più facile che venga generata la soluzione in un numero minore di generazioni. Anche aumentare la probabilità di crossover fino a 0.8 e quella di mutazione fino a 0.15 ha dimostrato essere favorevole per trovare una soluzione più velocemente. Riporto alcuni dati significativi:

- Npop = 150, pc = 0.7, pm = 0.1 → generazione media: 43.3 ± 34.6;
- Npop = 200, pc = 0.8, pm = 0.15 → generazione media: 32.4 ± 24.6;
- Npop = 300, pc = 0.7, pm = 0.1 → generazione media: 34.4 ± 22.2.

I risultati migliori si hanno per un gruppo di 100 run svolte con l'approccio misto FP-Boltzmann selection, i cui individui sono stati valutati su 10 coppie di stack e table, con parametri:

- $T_{iniziale} = 1.0$, T dimezzata ogni 4 generazioni;
- Npop = 150, pc = 0.8, pm = 0.15, limite impostato a 200 generazioni per run (mai raggiunto).
- Generazione media: 14.3 ± 8.8.

Tempi computazionali e correttezza

La task più dispendiosa in termini di tempo all'interno di una generazione, prevedibilmente, corrisponde alla valutazione degli individui. Il grafico mostra la percentuale di tempo utilizzata per valutare la fitness rispetto al tempo computazionale totale di ogni generazione, per un gruppo di 10 run. Si nota che dopo circa 11 generazioni tale percentuale satura al 98% circa dell'intero tempo computazionale.

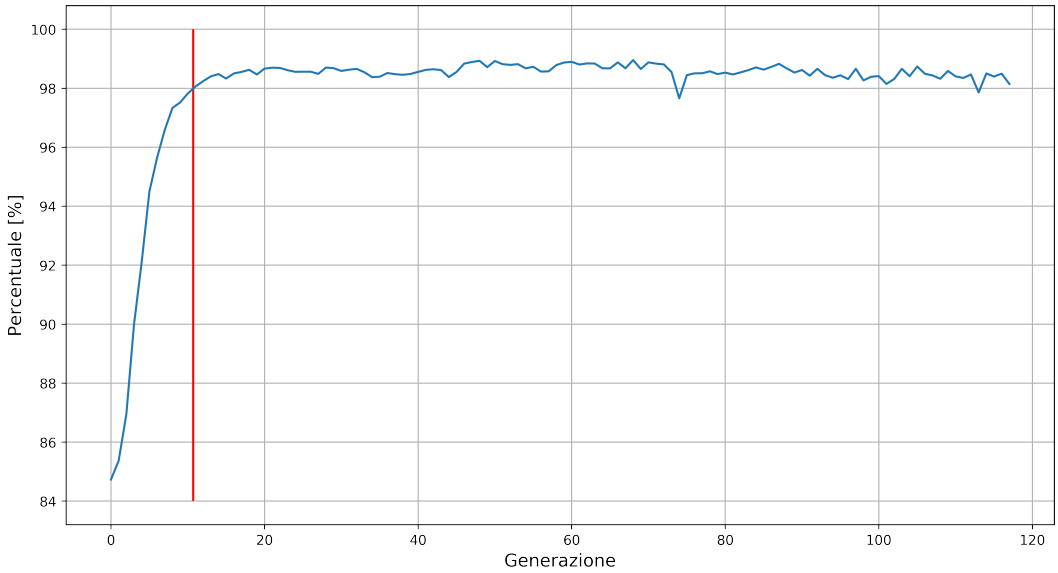

Figura 7: Percentuale del tempo utilizzato per la fitness rispetto al tempo computazionale totale della generazione.

Si può notare il lavoro che ho svolto per cercare di ottimizzare il processo di generazione delle popolazioni; ho di volta in volta sostituito cicli e altri comandi tipici di un approccio procedurale con funzioni *built-in* riducendo di molto i tempi. Ho cercato in egual modo di ottimizzare la fitness, anche qui eliminando tutti i cicli; ho interpretato la saturazione del tempo computazionale al 98% del tempo totale come una caratteristica intrinseca dell'algoritmo (la fitness è il 'luogo' dove effettivamente ogni individuo/programma viene eseguito e quindi impiega più tempo rispetto a tutte le altre operazioni).

Il tempo computazionale medio per run è proporzionale al numero di individui iniziali (più individui da valutare corrisponde in media a più tempo impiegato) e alla generazione media. Risulta anche proporzionale alla fitness media della run; si può dedurre che oltre a dipendere dal numero di individui che vengono valutati, dipende dalla struttura degli stessi (cioè dal tipo di comandi che contengono) e dalla loro capacità di risolvere gli stack (quindi di compiere più iterazioni).

I migliori tempi per l'approccio fitness-proportionate sono stati registrati per 100 run con parametri Npop = 100, Ngen = 200, pc = 0.8, pm = 0.15 e sono mostrati nella figura 2; il tempo medio per run è 41.26 ± 30.14. Per quanto riguarda l'approccio misto, i risultati migliori sono quelli nel grafico 4 e sono stati ottenuti per 100 run con parametri Npop = 100, Ngen = 200, pc = 0.8, pm = 0.15, con un tempo medio per run pari a 55.81 ± 73.95.

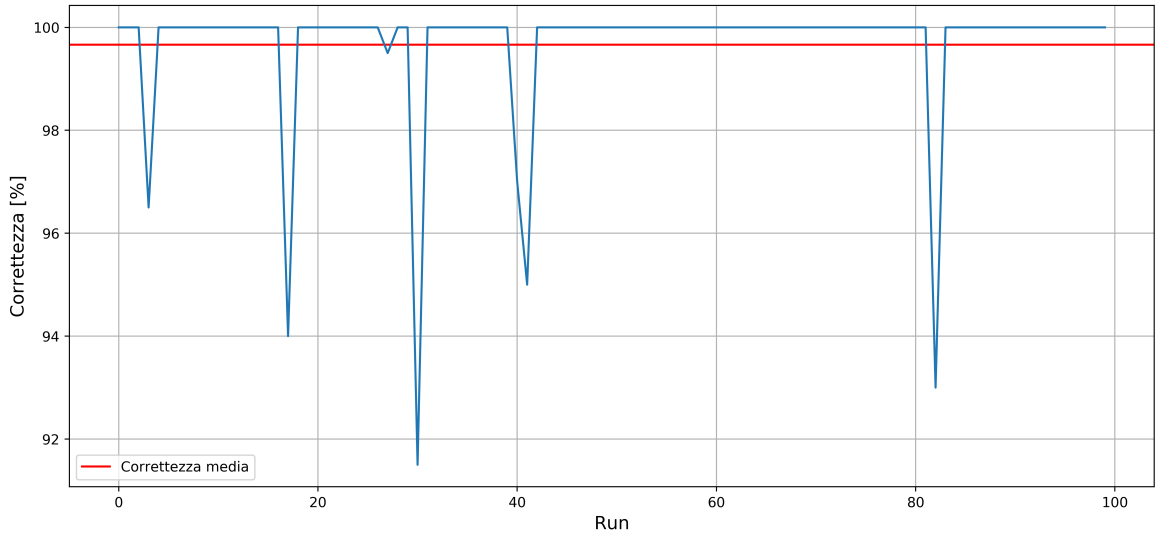

Figura 8: Correttezza delle soluzioni per un gruppo di 100 run con selezione mista.

La correttezza (per dettagli riferirsi alla sezione **Correttezza delle soluzioni**) più elevata è stata ottenuta con un gruppo di 100 run con parametri Npop = 200, Ngen = 200, pc = 0.8, pm = 0.15 e approccio fitness-proportionate; il numero di coppie stack-table di prova usato è 20. La correttezza si assesta sul valore 99.88 ± 1.00, a discapito però del tempo medio per run pari a 230.83 ± 234.08. La selezione mista apporta un notevole passo avanti in questa direzione in quanto, a fronte di una valutazione su 10 coppie di prova iniziali, sono riuscito a generare delle soluzioni con una correttezza media di 99.67 ± 1.39 (figura 8) con un tempo medio per run decisamente minore (57.08 ± 66.89). I parametri per quest'ultimo gruppo di 100 run sono Npop = 150, Ngen = 200, pc = 0.8, pm = 0.15.

Conclusione

L'approccio genetico si è dimostrato un metodo vincente per la risoluzione del problema del block-stacking; i tempi di esecuzione dell'algoritmo sono contenuti ed è stato anche facile arrivare a un certo grado di ottimizzazione della soluzione.

L'utilizzo di Mathematica ha reso la generazione degli individui estremamente semplice, così come le operazioni di crossover e mutazione: la manipolazione delle espressioni è molto più potente e flessibile rispetto a un approccio procedurale puro. La correttezza delle soluzioni (ovvero la loro generalità nel risolvere qualsiasi configurazione data di stack e table, formate a partire da qualsiasi parola) può essere ulteriormente migliorata campionando più stack e table di prova all'inizio del programma, a discapito del tempo impiegato per trovare la soluzione.

Mutuare il "simulated annealing" dal corso di Laboratorio di Simulazione Numerica ha permesso di sperimentare un nuovo metodo di selezione / progressione delle generazioni, con un miglioramento della generazione media a cui l'algoritmo trova la soluzione; i tempi sono invece leggermente peggiorati. Questo mix tra programmazione di ispirazione biologica e metodi di carattere fisico applicati all'informatica saranno senza dubbio per me fonte di futuri spunti e approfondimenti.