Relazione per progetto algoritmi genetici:
un robot universale

Adalberto Valsecchi

23 giugno 2020

1 Cos'è un algoritmo genetico?

I **problemi di ottimizzazione** sono una categoria di compiti che consistono nella ricerca di una soluzione ottima tra un numero tipicamente grande di possibili soluzioni. Una moltitudine di domande può essere formulata in termini di questi problemi e ciò rende molto importante sviluppare tecniche per risolverli. Queste possono essere di varia natura, per esempio analitica o computazionale. Ci concentreremo su queste ultime tecniche di ottimizzazione, che consistono di svariati algoritmi, i quali permettono di ottenere la soluzione desiderata (o meglio, una sua approssimazione). Per ogni problema di ottimizzazione si deve scegliere quale tecnica utilizzare per ottenere la soluzione nel modo che meglio concili l'efficienza computazionale e la precisione del risultato (ovvero quanto effettivamente ci siamo avvicinati alla soluzione migliore). È proprio quando gli approcci computazionali meno complessi (come per esempio la random search di una soluzione), sia quello analitico, falliscono che entrano in gioco gli **algoritmi genetici**.

Questi sono un approccio alla computazione che trae ispirazione dal processo naturale dell'**evoluzione**. Vediamone ora il funzionamento per comprenderli e per introdurre la terminologia. Al fine di ottenere la soluzione ottima il procedimento è il seguente:

1. generare una **popolazione** casuale di N_{pop} individui (possibili soluzioni);
2. assegnare un valore di **fitness**, che quantifica quanto la soluzione proposta si avvicina alla risoluzione del problema, ad ogni individuo;
3. selezionare, in base alla fitness (fitness più alta, probabilità di essere selezionato più alta), una coppia di individui (genitori) e implementarne una **ricombinazione genetica** che, tramite **crossover** e **mutazione**, generi una nuova coppia di individui (figli) (si ha quindi la **riproduzione** degli individui);
4. ripetere il punto precedente $N_{\text{pop}}/2$ volte in modo da creare una nuova popolazione (**generazione** successiva);

In questo modo abbiamo creato un algoritmo che, partendo da una popolazione iniziale casuale, tramite un meccanismo di selezione ed ereditarietà dei caratteri genetici (durante la riproduzione) porta, **al procedere delle generazioni**, a una popolazione che contiene come individuo la soluzione ottima. Esiste infatti un teorema (dimostrato da Holland) che assicura la convergenza di un algoritmo genetico alla soluzione ottima dopo un numero sufficiente di generazioni.

2 Quale problema vogliamo risolvere?

Supponiamo di avere una serie di blocchi, ognuno contrassegnato da una lettera; prendiamo 10 di questi, in modo da avere tutte e sole le lettere che compongono la parola "universale". Il problema che ci proponiamo di risolvere consiste nella ricerca di un individuo che sia in grado di costruire, partendo da qualunque configurazione iniziale di questi 10 blocchi (che possono essere già parzialmente impilati oppure sulla tavola di lavoro), una pila ordinata sulla quale si possa leggere la parola desiderata (ovvero "universale").

N.B.: ovviamente la parola che abbiamo scelto è solo un esempio e le considerazioni presenti in questa relazione possono essere estese a qualsiasi parola che ci passi per la mente.

2.1 Come costruire gli individui

Ogni individuo che compone la popolazione sarà un "programma", nel senso che compierà delle azioni sulle lettere, spostandole dal tavolo alla pila e viceversa. Ogni programma viene composto dalle seguenti operazioni fondamentali (rispettivamente, 3 sensori e 5 funzioni):

- CS (per *current stack*): restituisce la lettera in cima alla pila;
- TB (per top correct block): restituisce la lettera corretta più in alto nella pila. **N.B.**: tutti i blocchi sotto a TB devono essere corretti, ovvero devono essere le lettere giuste nelle posizioni giuste;
- NN (per *next needed*): restituisce il blocco successivo che serve per scrivere la parola corretta;
- MS(x) (per *move to stack*): sposta il blocco x (se è sul tavolo) in cima alla pila;
- MT(x) (per *move to table*): sposta il blocco dalla cima della pila al tavolo, se x è contenuto nella pila;
- DU(expr1, expr2): valuta expr1 finchè expr2 è falsa;
- NOT(expr): restituisce vero se l'espressione è falsa e viceversa;
- EQ(expr1, expr2): restituisce vero se expr1 ed expr2 hanno lo stesso valore, altrimenti restituisce falso.

Gli individui verranno quindi costruiti tramite estrazioni casuali che però rispettino delle regole sintattiche: MT e MS prendono come input solo i sensori, il secondo argomento di DU prende come input solo espressioni che restituiscono vero o falso. Gli altri argomenti non hanno limitazioni.

3 Implementazione del codice

Per programmare l'algoritmo genetico abbiamo utilizzato il linguaggio di programmazione Mathematica (in particolare le versioni 4.0 e 5.2). Questo progetto è stato quindi un esercizio per imparare ad utilizzare questo linguaggio molto potente e ad alto livello.

In questa sezione ci soffermeremo sulle parti più interessanti del codice che abbiamo implementato e non commenteremo invece quelle più banali.

3.1 Programmazione delle operazioni

Le operazioni sono state tutte di semplice programmazione. L'unica osservazione degna di nota è a proposito del DU: è importante notare che Mathematica valuta le espressioni nel momento stesso in cui vengono passate come argomenti di un'operazione; noi vogliamo evitare che questo succeda, siccome gli argomenti del DU devono essere valutati ripetutamente all'interno di un ciclo While. A tal scopo abbiamo assegnato a DU l'attributo "HoldAll" e applicato ReleaseHold[expr] ogni volta che un'espressione viene chiamata all'interno dell'operazione.

Inoltre, è necessario imporre delle limitazioni al numero di iterazioni che possono essere compiute da un DU. In particolare, se la parola che vogliamo scrivere è lunga N lettere, allora stopperemo il nostro DU dopo 3N iterazioni; se non imponessimo questa condizione il DU rischierebbe di entrare in un loop infinito, di fatto bloccando l'esecuzione del programma.

3.2 Costruzione dell'individuo

La prima componente importante di un algoritmo genetico è la creazione degli individui di partenza. Anche in questo caso bisogna ricordare che Mathematica valuta subito le espressioni che incontra. Noi invece vogliamo generare degli individui sintatticamente corretti, ma che ancora non agiscano sulla pila e sulla tavola e quindi vogliamo rimandare la valutazione delle componenti (cioè le varie operazioni) di questi individui. Per fare ciò costruiremo gli individui con dei comandi privi di definizione, che andremo a sostituire una volta che vorremo valutare effettivamente l'azione dell'individuo (ogni comando sarà preceduto da una x, e.g. NOT→xNOT).

Come primo comando considereremo sempre xEQ, che ci permette di avere una struttura non banale degli individui. A partire da questo estrarremo i comandi sintatticamente compatibili e otterremo un individuo formato da molte ramificazioni che si interromperanno una volta che estrarremo un sensore (che non ha argomenti).

In seguito per eseguire l'individuo implementiamo una funzione che sostituisca tutti i comandi preceduti da una x in comandi senza la x (e.g. xNOT→NOT)).

3.3 Ricombinazione genetica

I due momenti fondamentali della ricombinazione genetica, nel momento in cui selezioniamo due individui genitori e li facciamo riprodurre, sono il **crossover** e la **mutazione**. Quest'ultima non è molto differente dalla costruzione di un individuo; infatti, quello che facciamo è selezionare (con una probabilità p_m) un punto casuale dell'individuo che vogliamo mutare e sostituirlo con una serie di comandi generati proprio con la funzione che usiamo per generare un nuovo individuo.

Per quanto riguarda il crossover (che avviene con probabilità p_c), l'implementazione è più complicata. A questa funzione passiamo una coppia di genitori e selezioniamo casualmente un comando presente in uno di questi due individui (più precisamente prendiamo la ramificazione dal comando selezionato in poi). Successivamente escludiamo tutti i comandi dell'altro individuo per i quali lo scambio sarebbe sintatticamente sbagliato e in seguito scambiamo due comandi tra i programmi genitori. Questa funzione restituisce quindi una coppia di figli a cui in seguito applicheremo la mutazione.

3.4 Fitness

Il calcolo della fitness è probabilmente il momento più importante per il funzionamento di un algoritmo genetico ed è stata infatti la parte più complicata da implementare. Questa ci farà decidere la probabilità per ogni individuo di venire selezionato per essere un genitore. È quindi la formulazione stessa di questa funzione che influenzerà maggiormente le performance dell'algoritmo e la sua capacità di trovare una soluzione nel modo più efficiente possibile. L'approccio che abbiamo deciso di intraprendere è stato quello di considerare una funzione di fitness il più minimale possibile. Il compito di questa funzione è quello di valutare quanto un individuo si avvicini a trovare una soluzione e di premiarlo in caso positivo. Altre caratteristiche dell'individuo verranno premiate, come la varietà del suo patrimonio genetico. In particolare un individuo viene eseguito partendo da sette diverse pile (per ottenere una soluzione generale, cioè che risolva tutte le condizioni casuali che può incontrare) e abbiamo studiato come questo le costruisse correttamente. Gli individui che costruiscono più pile contemporaneamente e di pari passo ricevono premi più alti rispetto a individui che costruiscono correttamente una sola pila e lasciano le altre sbagliate. Ci sono diversi criteri che si possono utilizzare per selezionare gli individui di una popolazione in base alla loro fitness. Per questo progetto abbiamo deciso di utilizzare quello probabilmente più semplice e intuitivo: **fitness proportionate**. Questo criterio consiste nell'estrarre un individuo con probabilità proporzionale alla sua fitness. In pratica questo corrisponde a una roulette truccata in cui gli individui con una fitness maggiore hanno una probabilità maggiore di essere estratti.

3.5 Tempi di esecuzione

Nel codice abbiamo anche implementato (in modo piuttosto rapido e non troppo approfondito) il *profiling* del programma. Abbiamo quindi potuto osservare che oltre il 95% del tempo viene utilizzato per calcolare la fitness degli individui; in particolare, poco meno del 95% del tempo totale è impiegato per eseguire effettivamente gli individui (questa operazione avviene infatti all'interno della funzione che calcola la fitness). È sorprendente osservare come la ricombinazione degli individui, che consiste nella selezione dei genitori, nel crossover e nella mutazione, impiega circa l'1% del tempo totale. Alla fine di ogni run il codice testa la generalità della soluzione trovata eseguendo l'individuo su 100 pile generate casualmente; questa operazione richiede solo il 2% del tempo di esecuzione.

Con la fitness che abbiamo scelto (v. sezione 4.1) il tempo medio di esecuzione per una run è ~ 40 secondi.

4 Risultati

Iniziamo ora a discutere i risultati ottenuti attraverso diverse simulazioni compiute con il codice appena descritto. L'obiettivo principale di questa sezione è quello di selezionare i parametri della fitness nel modo più minimale possibile tale per cui la soluzione viene trovata nel minor numero di generazioni. Sarà quindi importante riuscire a trovare l'equilibrio corretto tra generalità delle soluzioni, velocità di esecuzione del codice e numero di generazioni (in media) dopo le quali si trova la soluzione.

4.1 Il ruolo dei parametri della fitness

Ci concentriamo ora sui parametri della fitness. Quali di questi sono indispensabili? Quali non variano i risultati, ma ci permettono di avere dei tempi di esecuzione molto più brevi? Quali ci permettono di avere una soluzione più generale (ovvero che riesce a ottenere la parola desiderata partendo da qualsiasi condizione iniziale)?

4.1.1 Limitazioni sul numero di EQ, NOT e DU

Ci siamo chiesti se limitare il numero delle operazioni EQ, NOT e DU che appaiono negli individui potesse aiutarci a rendere più rapida la convergenza dell'algoritmo; se gli individui ottenuti contengono un numero di operazioni superiore al limite, allora non vengono eseguiti e la loro fitness è caratterizzata solo dalla varietà del patrimonio genetico che trasportano. Abbiamo quindi lanciato 200 run imponendo dei limiti ($\#EQ = 6, \#NOT = 6, \#DU = 6$) e altrettante senza.

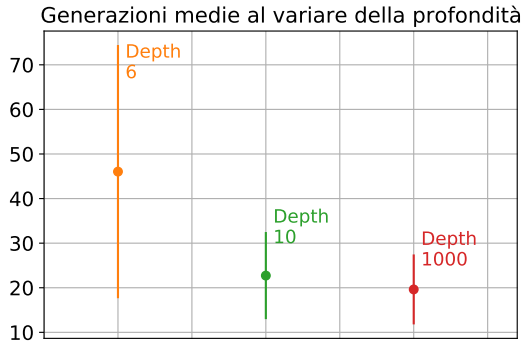

Figura 1: **Valori delle generazioni soluzione medie e del rispettivo errore per diverse profondità massime.** Passare da 6 livelli a 10 ha una grande influenza sulla generazione media. Rimuovere il limite alla profondità consente di abbassare ulteriormente l'incertezza che si ha.

Quello che abbiamo potuto notare è che le limitazioni su EQ e NOT non influiscono in nessun modo sulla convergenza. Per i DU la situazione è leggermente più complicata: sebbene la limitazione sul numero di questa operazione non influisca direttamente sulla convergenza, questo condiziona drasticamente il tempo di esecuzione; infatti in assenza di questo vincolo il codice impiega più di tre volte tanto (tempo medio per una run 126 secondi invece che 38) rispetto al caso in cui abbiamo imposto i limiti.

I dati mostrano che in entrambi i casi la convergenza (con profondità massima dell'individuo fissata a 6) avviene attorno a un valore medio $\mu_{\text{gen}} \simeq 46 \pm 28$.

4.1.2 Limitazione sulla profondità degli individui

In quest'ottica di decostruzione del problema abbiamo lanciato delle simulazioni con una limitazione sulla profondità massima degli individui e senza (la fitness si comporta nel medesimo modo della sezione precedente, ovvero non esegue gli individui troppo profondi). Abbiamo quindi eseguito 500 run con tre diversi limiti per la profondità: 6, 10 e 1000 (che a livello pratico equivale a non avere limiti).

Come si osserva dalla figura 1, aumentare la profondità ci permette di ottenere la soluzione, in media, con un numero di generazioni molto inferiore. È significativo notare che aumentando di poche unità la profondità massima (da 6 a 10), la generazione media crolla di quasi metà del suo valore; invece rimuovere il limite (o meglio, avere al massimo 1000 livelli) diminuisce di poco il valore medio delle generazioni, però abbassa la deviazione standard σ_{gen}.

4.1.3 Valori numerici dei parametri rimanenti

Non ci resta ora che studiare l'effetto dei valori numerici dei parametri che consideriamo; in altri termini: quale peso diamo ad ogni caratteristica degli individui e questo quanto influisce sulla fitness? I fattori che premiamo di un individuo sono:

1. la varietà del patrimonio genetico, ovvero quanti comandi diversi appaiono nell'individuo;
2. il numero di lettere corrette posizionate per ogni pila;
3. il fatto che le diverse pile vengano "costruite" contemporaneamente (un individuo che posiziona 2 blocchi corretti in 5 pile viene premiato di più rispetto a uno che costruisce perfettamente una sola pila).

Lanciamo quindi diverse simulazioni per verificare se tutti questi incrementi sono necessari o se la soluzione viene trovata a prescindere da alcuni di essi. È complicato esprimersi a priori sulla risposta, ma è ragionevole credere che il criterio più utile tra i tre presentati sia probabilmente il terzo, visto che guida l'algoritmo genetico verso una generalità che gli altri due criteri ignorano.

Sorprendentemente, le simulazioni rivelano una fenomenologia completamente diversa da quella attesa. In particolare, abbiamo fatto agire un criterio per volta, ponendo a zero il peso degli altri due, e quello che si osserva è che il secondo di questi criteri è di gran lunga il più importante. Infatti, con questo criterio abbiamo ottenuto una convergenza media di 20 ± 9 generazioni, mentre con gli altri due la convergenza richiede più di dieci generazioni in più e l'incertezza cresce altrettanto. È però interessante (e utile) notare (v. figura 2) che le simulazioni svolte con tutti e tre i criteri attivi richiedono il tempo minimo per run e restituiscono la convergenza migliore. Questo ci suggerisce di mantenere tutti i criteri: infatti con questo accorgimento non perderemo di generalità, ma avremo un codice molto più rapido, perché gli individui selezionati avranno una struttura che richiede un minor tempo d'esecuzione.

4.2 Fitness per una sola run

Dopo aver selezionato i parametri della fitness siamo pronti a commentare l'andamento di questa funzione al crescere delle generazioni e a discutere della statistica che segue il nostro sistema.

Iniziamo illustrando il comportamento della fitness ottenuta in una singola run. Il grafico ottenuto ovviamente ha una *valenza qualitativa*, in quanto è una singola realizzazione del nostro insieme statistico; è però interessante e istruttivo, al fine di comprendere il comportamento del nostro algoritmo genetico, studiarne l'andamento.

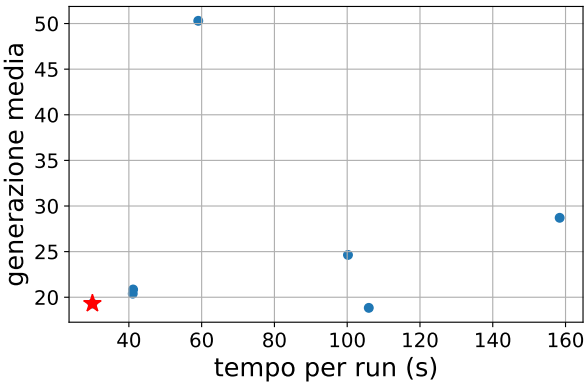

Figura 2: **Generazione media in funzione del tempo impiegato per run al variare dei parametri**. In figura si osservano la convergenza (in termini di generazioni) e i tempi di esecuzioni medi delle run al variare dei parametri del nostro algoritmo genetico. Ogni punto rappresenta una scelta diversa di parametri. La stellina rappresenta la scelta migliore, in cui abbiamo tutti e tre i criteri attivati.

Se ci interessiamo alla fitness massima e alla fitness media che si ottengono per ogni generazione, otterremo il grafico presentato nella figura 3. La caratteristica più importante è il salto che la fitness massima compie in corrispondenza della generazione 25, in cui viene trovata la soluzione dall'algoritmo genetico. Da questo punto in poi il valore massimo della fitness (per ogni generazione) rimarrà stabile attorno a quel valore, il che significa che, dal momento in cui troviamo la soluzione, l'algoritmo produce sempre popolazioni in cui almeno un individuo risolve il problema. Inoltre, possiamo osservare che con il crescere delle generazioni avremo più individui corretti (ovvero che risolvono il problema) e questa cifra si stabilizza molto presto attorno al 50% della popolazione (questo grafico si accompagna di pari passo con l'andamento della fitness media). Questo dato è interessante perché significa che la fitness scelta fa sì che metà degli individui siano corretti e l'altra metà della popolazione invece garantisca la presenza di una certa varianza genetica, impedendo che tutti gli individui si specializzino nel risolvere il medesimo problema.

La situazione è invece completamente diversa nelle generazioni iniziali, prima di quella in cui viene trovato l'individuo soluzione. In questo regime, la fitness (sia quella media che quella massima) cresce lentamente e i picchi che mostra il grafico sono degli evidenti tentativi dell'algoritmo di trovare un individuo che la massimizzi. Un tratto fondamentale del comportamento degli algoritmi genetici è esemplificato dal picco che si osserva attorno alla quindicesima generazione: il nostro programma ha trovato un individuo che performa in modo migliore rispetto

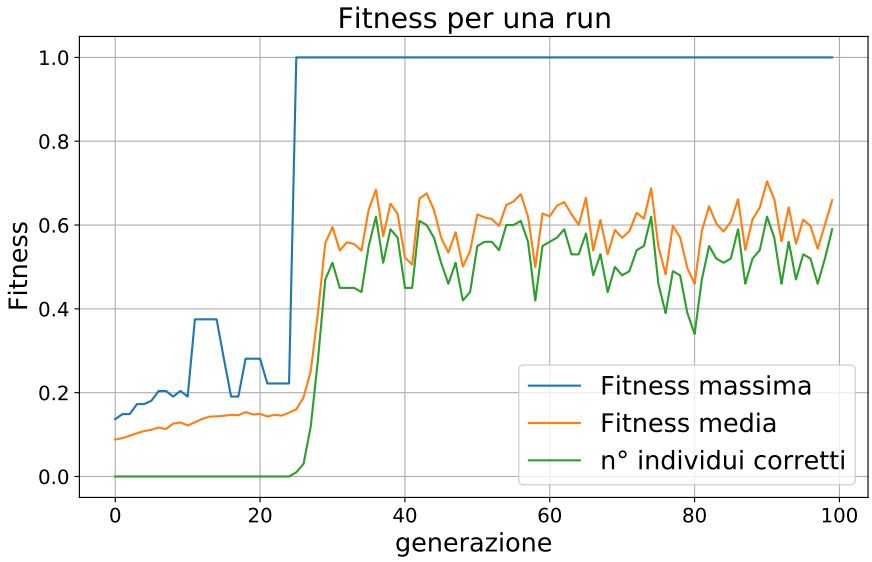

Figura 3: **Fitness massima e media al variare delle generazioni per una run.** La figura mostra molte caratteristiche fondamentali per comprendere gli algoritmi genetici: la forma "a gradino" della fitness massima, la presenza di picchi nelle prime generazioni e la crescita della fitness media insieme al numero di individui corretti.

a quelli della generazione precedente, ma la natura casuale (e distruttiva) della programmazione genetica implementata nel crossover e nelle mutazioni distrugge questo individuo, che viene perso nella generazione successiva.

4.3 Statistica su più run

Per conferire una valenza generale alla nostra trattazione ci proponiamo di studiare la fenomenologia del programma in discussione su più run.

4.3.1 Distribuzione delle run

Siamo innanzitutto interessati alla distribuzione delle generazioni in cui il problema viene risolto (che chiameremo "generazioni soluzione"). Per ottenere dei dati quantitativamente rilevanti abbiamo lanciato 1000 run (un compito che richiede all'incirca 10 ore, date le prestanzioni del codice ($\sim$ 40 secondi per run)) e costruito l'istogramma e la funzione cumulativa di questi dati, presenti in figura 4. In questi due grafici abbiamo anche rappresentato la generazione media (la riga ver-

ticale in arancione) e la sua deviazione standard σ (l'area arancione rappresenta un'incertezza di un sigma attorno al valore medio).

L'istogramma (plot **a**) mostra una statistica per cui le generazioni soluzione si concentrano soprattutto entro una deviazione standard dalla media (in quest'area abbiamo più del 70% dei risultati); in particolare la distribuzione di probabilità delle generazioni ha un andamento somigliante a quello di una gaussiana, anche se il grafico da noi ottenuto presenta una coda (quella di destra) più lunga dell'altra.

Nel plot **b** abbiamo invece riportato il grafico della funzione cumulativa delle generazioni. Questo grafico rende più evidente e visivamente più chiara la somiglianza tra la distribuzione da noi ottenuta e quella gaussiana. Senza addentrarci in una trattazione quantitativa, che richiederebbe un numero di dati ben maggiore, possiamo osservare che la media della CDF gaussiana (la linea rossa tratteggiata verticale) rientra abbondantemente nella barra d'errore della media dei dati raccolti. È quindi interessante osservare come la statistica che caratterizza l'emergere delle soluzioni dell'algoritmo genetico da noi implementato rispetti il teorema del limite centrale.

4.3.2 Media della fitness massima

Ci chiediamo ora cosa succede se consideriamo i grafici presentati nella sezione 4.2 per 200 run e ne prendiamo la media (generazione per generazione). Notiamo che queste 200 run richiedono molto tempo perché abbiamo modificato il codice affinché ogni run continui a calcolare la fitness fino alla centesima generazione e non si interrompa nel momento in cui trova l'individuo.

Se la fitness massima per tutte le run avesse una forma a gradino, con il salto alla stessa generazione, otterremo una funzione come quella tratteggiata in figura 5. Invece, le generazioni a cui viene trovato l'individuo ideale sono distribuite attorno a un valore medio (come abbiamo discusso nella sezione 4.3.1); il grafico presente in figura tiene conto di questa varianza nei valori ottenuti e assume una forma più smussata. In particolare l'area gialla presente in figura mostra il range di tutte le generazioni che sono state estratte. Se osserviamo l'istogramma di queste estrazioni e lo confrontiamo con la differenza tra la funzione a gradino e la fitness massima ottenuta (riportato nell'inset, dove abbiamo opportunamente riscalato le due curve per osservare meglio il confronto), notiamo un ottimo accordo. Questo significa che più volte troviamo la soluzione in una generazione diversa da quella media più ci allonteremo dal grafico ideale in corrispondenza di quella generazione.

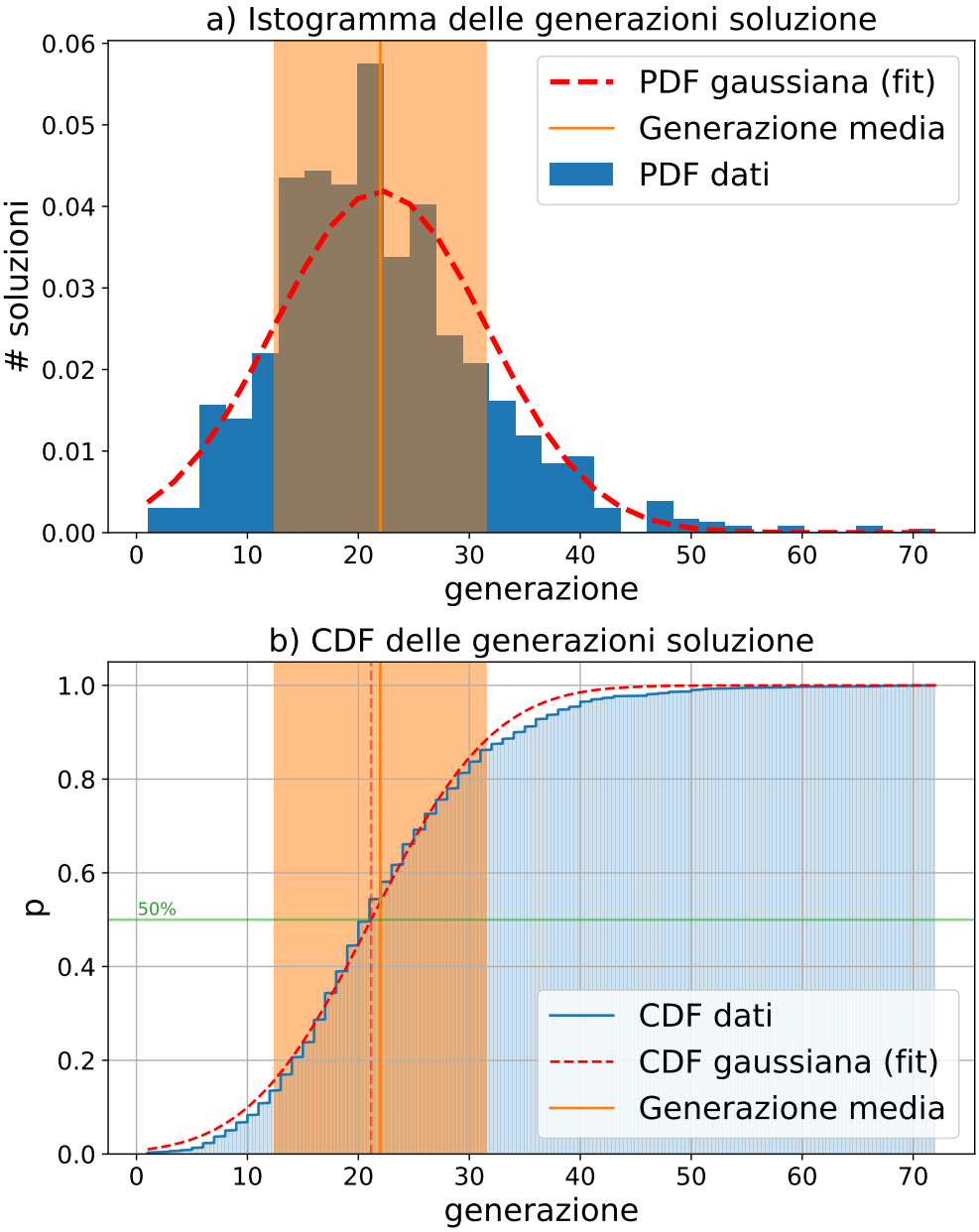

Figura 4: **Istogramma e CDF**. **a)** Il plot mostra un andamento simil-gaussiano dell'istogramma (normalizzato) che presenta un'asimmetria. **b)** Il grafico presenta la funzione cumulativa dei dati e il fit di una CDF gaussiana. Possiamo osservare un ottimo accordo.

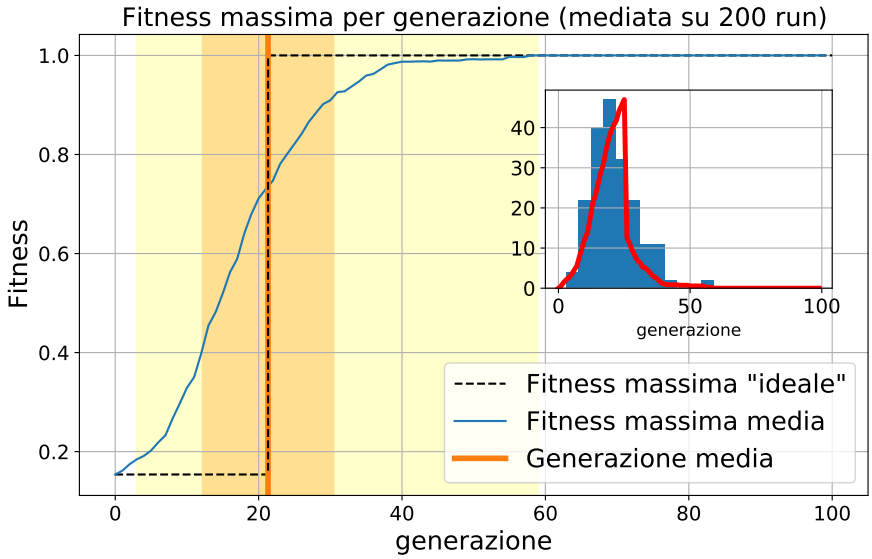

Figura 5: **Fitness massima mediata su 200 run**. Il grafico mostra il confronto tra la fitness ottenuta mediando 200 simulazioni e la fitness "ideale" che si otterrebbe se la distribuzione delle generazioni ideali fosse una delta centrata nel valore medio ottenuto. L'inset mostra il confronto tra l'istogramma delle generazioni soluzione e la differenza tra la fitness massima ideale e quella ottenuta. L'andamento di questi due grafici è evidentemente fortemente correlato.

5 Conclusioni

Ci chiediamo quindi se questo metodo di ricerca della soluzione ottima sia conveniente rispetto a, per esempio, una ricerca casuale della soluzione. Questa è implementata quando nel nostro codice genetico poniamo $p_{\mathrm{c}} = 0\%$, $p_{\mathrm{m}} = 0\%$ e quando assegniamo la medesima fitness a tutti gli individui della popolazione. Considereremo una popolazione di $N_{\mathrm{pop}} = 10^4$ individui e una sola generazione.

Lanciando diverse run (200), ci si rende condo che la ricerca casuale trova la soluzione ottima poche volte, circa per il 5% delle run, mentre per le altre l'esecuzione termina senza che sia stato trovato un individuo che risolve il problema.

Questo significa che il nostro codice, andando ad eseguire "solamente" 2000 individui (in media) è computazionalmente conveniente rispetto ad una random search, che utilizzandone 10000 spesso non trova la soluzione.

5.1 Domande e possibili approfondimenti

Durante questo progetto non è stato svolto uno studio sistematico al variare delle **probabilità di crossover e mutazione**. L'unico caso che è stato investigato è stato quello in cui $p_{\mathrm{c}} = p_{\mathrm{m}} = 100\%$; in questa situazione abbiamo osservato una convergenza più rapida (15 ± 5 generazioni) e anche dei tempi d'esecuzione minori (in media 20 secondi per run). Questi dati ci spingono a pensare che uno studio più approfondito di questi parametri potrebbe condurci a una ricerca più efficiente della soluzione ottima.

Un'altra domanda che meriterebbe un approfondimento è l'influenza del criterio di selezione sulle performance del nostro algoritmo. Abbiamo fatto alcuni tentativi, utilizzando degli altri criteri. Se il criterio è l'**elitismo** si ottengono delle performance simili, ma una correttezza leggermente inferiore; utilizzando invece la **fittest half** (gli individui estratti casualmente sono solo nella metà con la fitness più alta) otteniamo invece una convergenza leggermente peggiore. Questa analisi molto grossolana ci permette però di apprezzare che fitness proportionate sia un criterio molto flessibile ed efficace.