Algoritmi Genetici

Giovanni Bocchi

28 giugno 2012

Indice

1 Algoritmi Genetici **2**
1.1 Cenni storici 2
1.2 Stack Problem 3

2 Struttura del Codice **6**
2.1 Individuo 7
2.2 Popolazione 7
2.3 Fitness 7
2.4 Crossover 8
2.5 Mutazione 9

3 Mathematica **11**
3.1 Calcolo simbolico 11
3.2 Votazione 12
3.3 Ricombinazione 12
3.4 Generazioni 13

4 Conclusioni **15**
4.1 Prospettive future 16

Capitolo 1

Algoritmi Genetici

L'algoritmo genetico è un algoritmo di ottimizzazione e appartiene ad una particolare classe di algoritmi utilizzati in diversi campi, tra cui l'intelligenza artificiale. È un metodo euristico di ricerca ed ottimizzazione, ispirato al principio della selezione naturale di Charles Darwin che regola l'evoluzione biologica.

Il nome deriva dal fatto che i suoi pionieri si ispirarono alla natura e alla genetica, branca della biologia.

Gli algoritmi genetici sono applicabili alla risoluzione di un'ampia varietà di problemi d'ottimizzazione non indicati per gli algoritmi classici, compresi quelli in cui la funzione obiettivo è discontinua, non derivabile, stocastica, o fortemente non lineare.

L'Algoritmo Genetico (AG) è un algoritmo iterativo che determina l'evoluzione di una popolazione di soluzioni candidate di un dato problema di ricerca. Ad ogni iterazione, ogni soluzione candidata è valutata per stabilire i "rapporti di forza" tra gli "individui" della popolazione in modo che i più abili siano scelti preferenzialmente per generare nuove soluzioni candidate. La valutazione di un individuo necessita di un'intera simulazione del modello che si intende ottimizzare, adottando i parametri specificati dalla particolare soluzione candidata che si sta valutando.

1.1 Cenni storici

Gli AG nacquero in un contesto in cui un numero sempre crescente di ricercatori e studiosi aveva iniziato a interessarsi dei sistemi naturali come fonte d'ispirazione per la realizzazione di algoritmi di ottimizzazione per problemi ingegneristici. Nei primi anni '60 John Holland, con l'obbiettivo di realizzare sistemi capaci di auto-adattamento, si interessò dei principi che

regolano l'evoluzione dei sistemi adattivi naturali ipotizzando che la competizione e l'innovazione fossero i meccanismi fondamentali tramite cui gli individui acquisiscono le capacità di adattarsi ad ambienti che cambiano nel tempo e rispondere ad eventi inattesi.

Dalla metà degli anni '60 vennero realizzati semplici sistemi computazionali che, tuttavia, già presentavano caratteristiche comuni agli AG così come li conosciamo oggi: una popolazione di individui era fatta evolvere nel tempo e i meccanismi di derivazione erano semplici astrazioni di operatori genetici.

Diversamente da quanto accadde per altre principali tecniche computazionali ispirate all'evoluzione che naquero all'incirca nello stesso periodo, vennero poste delle significative basi teoriche per il suo modello su cui si sono basati moltissimi dei successivi studi sugli AG.

Il periodo dal 1990 a oggi è stato caratterizzato da un'enorme crescita della comunità degli Algoritmi Genetici e nuove applicazioni degli stessi hanno interessato un gran numero di nuove aree di ricerca.

1.2 Stack Problem

Il problema trattato in questo elaborato è generalmente conosciuto con il nome di *stack problem*. Ideato da Koza nel 1992, ha come obiettivo è quello di sviluppare un algoritmo che impili dei cubi etichettati da delle lettere, disposti casualmente su un tavolo, finché non sia composta la parola "UNIVERSAL", come mostrato in Figura 1.1.

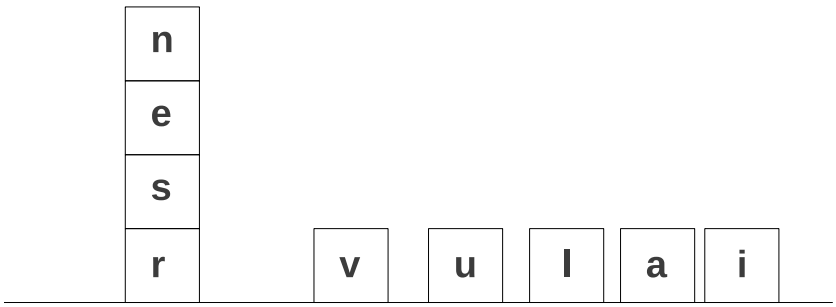

Figura 1.1: Rappresentazione dei dati iniziali. I blocchi impilati vanno a costituire la lista Stack mentre i restanti formeranno la lista Table.

É facile intuire che ci troviamo di fronte ad un tipico caso in cui gli Algoritmi Genetici si prestano benissimo alla risoluzione del problema. Nel problema in esame infatti, lo spazio delle soluzioni è troppo grande per essere

analizzato in modo esauriente e, pur avendo un'idea qualitativa di "soluzione ottimale", non è nota a priori una procedura sistematica per determinarla.

Per tale problema si è deciso di utilizzare il linguaggio di programmazione *Mathematica*. Tale linguaggio si è rivelato particolarmente adeguato poichè mette a disposizione un'ampia scelta di primitive che consentono di effettuare operazioni immediate su liste, pile e code con estrema semplicità rispetto ad altri linguaggi. Inoltre Mathematica consente di rappresentare le operazioni sotto forma di alberi decisionali, ossia come rappresentazione parentetica di un albero. Tale peculiarità rende tale linguaggio particolarmente adatto a rappresentare le forme di pensiero deduttivo, sotto forma appunto di alberi decisionali e aiuta altresì a rappresentare la popolazione di un Algoritmo Genetico per l'applicazione di operatori come ad esempio l'incrocio (crossover).

Koza, per risolvere questo problema, utilizzò un insieme di tre terminali e cinque funzioni. Tali funzioni devono essere combinate in maniera casuale a dare istruzioni sensate (sintatticamente corrette).
Di seguito vengono presentate brevemente le funzioni sopracitate:

Funzioni con output algebrico (Lettere)

- TB (top current block): Valuto stack partendo dal fondo; se è in parte corretto il programma restituisce l'ultima lettera corretta in stack.
- MS (move to stack): Se l'elemento x si trova in table lo sposta in cima a stack.
- MT (move to table): Se l'elemento x si trova in stack allora sposta il primo elemento di stack in table.
- NN (next needed): Se stack è un blocco di lettere corrette, resitutisce la prossima lettera necessaria.
- CS (current stack): Se la lunghezza di stack è diversa da 0 restituisce il primo elemento di stack.

Funzioni con output logico (Booleani)

- EQ (equal): Considera due funzioni e le eguaglia restituendo "TRUE" o "NIL" a seconda che l'uguaglianza sia verificata o meno.
- DU (do until): Il programma prende in esame due funzioni (exp1 e exp2); continua a valutare exp1 finchè exp2 diventa vera.

- NOT: Considera un'espressione e restituisce "TRUE" se l'output dell'espressione è "NIL".

Sulla base delle funzioni appena definite viene implementato l'algoritmo genetico presentato nel Capitolo 2.

Capitolo 2

Struttura del Codice

La struttura del codice si basa su semplici analogie con la biologia. L'idea strutturale che sta alla base dell'algoritmo viene schematizzata nella Figura 2.1. Seguendo tale diagramma di flusso vengono implentati i vari elementi

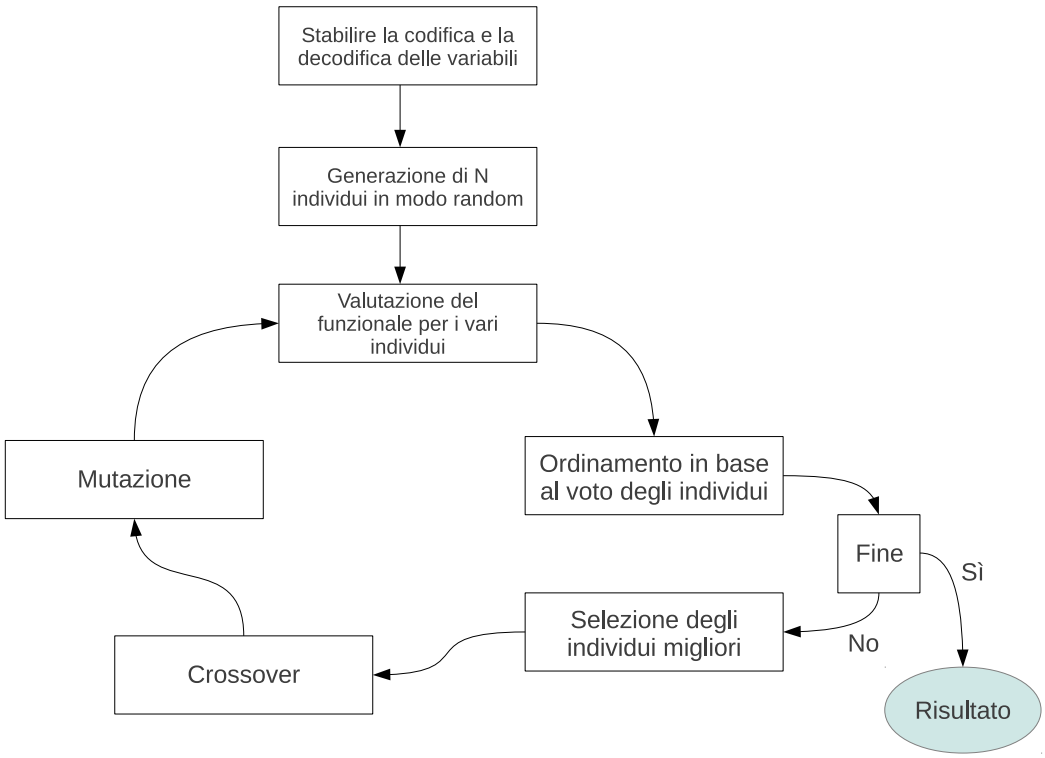

Figura 2.1: Diagramma di Flusso dell'algoritmo genetico.

costituenti l'algoritmo stesso.

2.1 Individuo

Una volta formalizzato il problema da risolvere, il primo passo consiste nel creare casualmente una possibile soluzione. Tale soluzione prenderà il nome di individuo. Nel nostro caso ogni individuo viene descritto da una serie di "mosse", la cui espressione è facilmente identificabile con un diagramma ad albero come mostrato in Figura 2.2. I nodi di tale diagramma, rappresentano

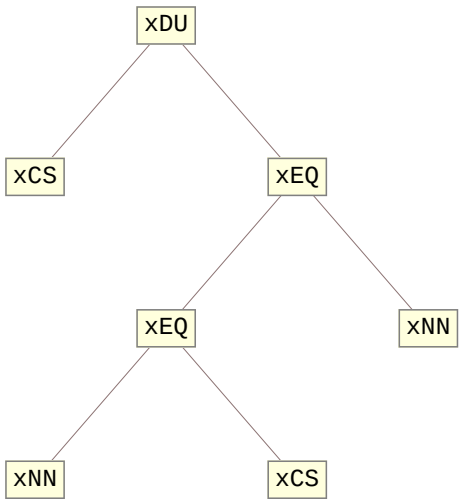

Figura 2.2: Rappresentazione ad albero di un individuo.

le mosse fondamentali descritte nel Capitolo 1.

2.2 Popolazione

Una volta generato il primo individuo, tale processo dovrà essere ripetuto N volte creando così una lista di possibili soluzioni create casualemente; tale lista andrà a costituire la *popolazione*.

2.3 Fitness

Al fine di valorizzare gli individui più forti è necessario assegnare un punteggio f_i che quantifichi la qualità dell'individuo in termini di soluzione del problema. La votazione assegnata ha lo scopo di influenzare la scelta dei genitori che produrranno la generazione successiva (figli).

Ad ogni individuo viene poi associata una probabilità p_i secondo la relazione

$$p_i = \frac{f_i}{\sum_i^N f_i} \tag{2.1}$$

La probabilità così trovata peserà la possibilità che l'individuo i-esimo venga inserito nella lista dei genitori. L'insieme di tutte le probabilità permetterà di costruire una *Roulette di probabilità* come mostrato in Figura 2.3. Al fine

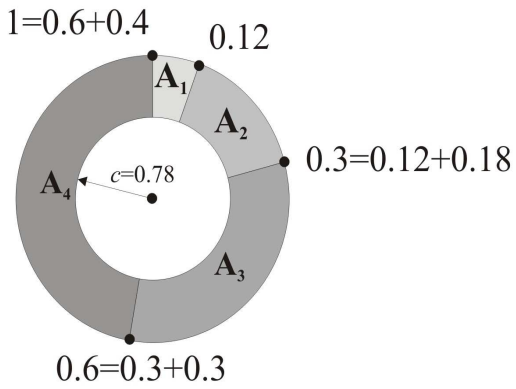

Figura 2.3: Roulette di probabilità.

di chiarire come vengono scelti i genitori viene proposto di seguito un piccolo esempio eplicativo.
Supponiamo di avere una popolazione costituita da 4 individui A_i con rispettive probabilità p_i. I quattro individui A_1, A_2, A_3 e A_4, con probabilità di selezione 0.12, 0.18, 0.3 e 0.4, occupano uno spicchio di roulette di ampiezza pari alla propria probabilità di selezione. Nell'esempio l'operatore di selezione genera il numero casuale $c = 0.78$ e l'individuo A_4 viene scelto e inserito nella lista genitori. L'operatore di selezione determina, dunque, quali individui della vecchia popolazione hanno la possibilità di generare dei discendenti. Poichè gli individui con fitness alta sono quelli favoriti, potendo contare un numero di copie maggiore rispetto a quelli con fitness più bassa, l'operatore di selezione gioca, nel contesto dell'algoritmo genetico, il ruolo della selezione naturale.

2.4 Crossover

In analogia con quanto accade in natura nei cromosomi tra genitori e figli, la funzione *crossover* prende due individui dalla lista genitori e li combina casualmente generando altri due individui che andranno inseriti nella lista

figli. Se da un punto di vista formale potrebbe non essere immediato, da un punto di vista grafico il concetto di crossover è facilmente identificabile come lo scambio di due rami. In figura 2.4 e 2.5 viene mostrato come da due individui vengono generati due figli.

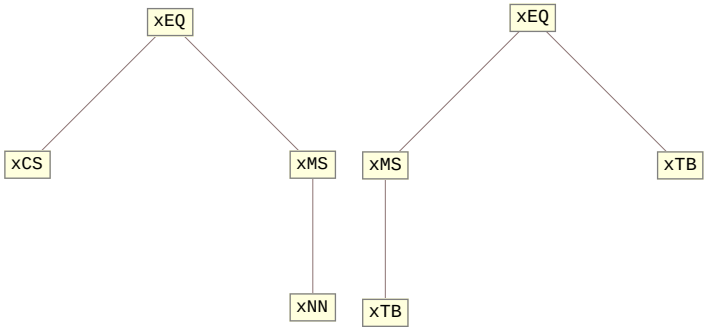

Figura 2.4: Rappresentazione ad albero dei due genitori che verranno utilizzati dalla funzione crossover.

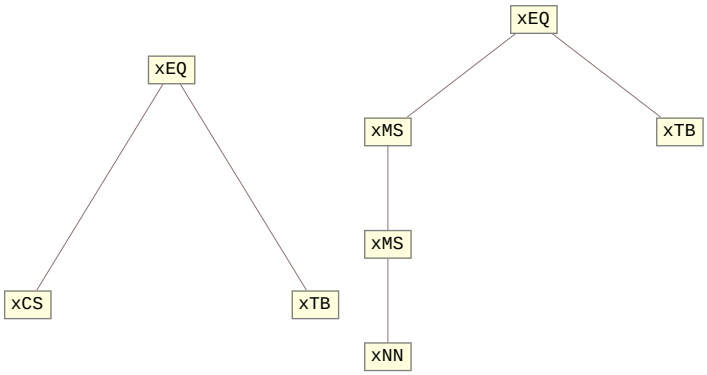

Figura 2.5: Rappresentazione ad albero dei due figli ottenuti attraverso la funzione crossover.

2.5 Mutazione

L'individuo figlio, come detto precedentemente, conterrà quindi una parte dell'individuo A e una parte dell'individuo B. L'algoritmo quindi restringerà sempre di più il dominio delle soluzioni trovando così la soluzione corretta. Per

evitare di cadere in minimi locali o addirittura non far convergere l'algoritmo, viene introdotto nell'individuo figlio una certa probabilità di mutazione, dove con il termine mutazione si intende una piccola variazione della propria struttura interna.

Nel nostro caso la mutazione non è stata eseguita in senso stretto come variazione di programmi preesistenti bensì con l'inserimento ad ogni generazione di n nuovi individui. Tale meccanismo verrà spiegato meglio nel Capitolo 3.

Capitolo 3

Mathematica

In questo capitolo verranno presentati e discussi alcuni passaggi fondamentali del codice.

L'algoritmo è stato implementato per mezzo del software "Mathematica". Tale software permette di eseguire sia calcoli numerici sia calcoli simbolici, permettendo così di effettuare operazioni su oggetti apparentemente privi di significato, significato che verrà definito in un secondo momento.

3.1 Calcolo simbolico

Come detto nel Capitolo 2 il primo passo nella costruzione dell'Algoritmo Genetico è l'implementazione dell'individuo. La creazione dell'individuo sarà del tutto casuale pur rispettando la correttezza sintattica; se una funzione richiede un argomento algebrico non potrà essere collegata ad una funzione con output logico. A titolo d'esempio viene riportata la definizione della funzione CS (current stack).

```
CS := Module[{temp},
        If[Length[stack] == 0,
         NIL,
         stack[[1]]
        ]
    ]
```

Dopo essere state implementate, le varie funzioni sono state successivamente divise in liste in funzione del tipo di output esplicitando gli eventuali argomenti

```
   lettere = {xCS, xTB, xNN, xMS[lett], xMT[lett]}
booleani = {xEQ[gen, gen], xNOT[lett], xDU[lett, bul]}
```

Come si può notare è stata aggiunta una "x" davanti ad ogni funzione. Tale scelta è motivata dal fatto che in Mathematica quando una funzione viene richiamata, questa agisce immediatamente secondo la sua definizione. Modificandone il nome (CS → xCS) viene perso il significato algebrico e la funzione viene vista dal programma come simbolo. Quando sarà necessario la sostituzione

```
individuo /. {xCS :> CS}
```

riporterà le funzioni allo stato originale così che, soltanto ora, potranno agire.

3.2 Votazione

L'assegnazione di un punteggio ad ogni programma costituisce, sotto certi aspetti, la parte più complicata del programma. Tenendo presente che i genitori delle generazioni future verranno scelti sulla base del voto ad esso associato, il criterio di votazione deve essere scelto molto attentamente in modo da volorizzare gli individui migliori ma non sopprimere completamente i peggiori. Un errore di questo tipo porterebbe a limitare fortemente lo spazio delle soluzioni analizzato non facendo più convergere il programma ad una soluzione.

Sulla base di quanto appena detto sono stati scelti i seguenti parametri di votazione:

- É stato assegnato un voto in funzione della variabilità di funzioni utilizzate dall'individuo. Questo impedisce che, con il susseguirsi delle generazioni, alcune funzioni vengano perdute.
- É stato assegnato un voto alla lunghezza del programma (numero di funzioni utilizzate), andando a penalizzare i programmi troppo lunghi.
- É stato dato un punteggio sulla base del risultato ottenuto dopo aver fatto agire l'individuo sulla lista stack. Sono stati premiati individui che svuotassero piuttosto che riempissero lo stack e penalizzati tutti quei programmi che lasciavano inalterato lo stack.

Una volta specificati tutti i criteri di valutazione ogni sottovoto è stato sommato piuttosto che moltiplicato, trovando così il voto definitivo f_i.

3.3 Ricombinazione

Vale la pena soffermarsi sulla ricombinazione degli individui e quindi sulla funzione *crossover*.

A differenza di quanto accade nei casi in cui un individuo viene identificato da una sequenza di 1 e 0 dove, ogni possibile *taglia* e *incolla* è concesso, nel nostro caso il dominio dei figli è fortemente ridotto dal vincolo che il programma sia sintatticamente corretto. Nell'ottica di un diagramma ad albero questo si traduce nel fatto che due rami possono essere sempre scambiati a patto che il loro output sia lo stesso. Per questo motivo viene riportata la parte di codice che si occupa dello scambio di rami.

- Attraverso la funzione Position cerco tutti i possibili tagli eliminando quelli inutili e concettualmente sbagliati. Successivamente viene estratto caso uno di essi.

```
pos1 = Position[individuo1, x_, Infinity];
pos1 = DeleteCases[pos1, {x___, 0}];
pos1 = DeleteCases[pos1, {}];
rami1 = RandomChoice[pos1];
```

- Una volta scelto il punto in cui tagliare il primo individuo bisogna ripetere lo stesso procedimento con il secondo individuo, specificando il tipo di ramo da tagliare. Questo viene fatto aggiungendo l'opzione /; all'interno di Position.

```
appartenenza =
   Which[MemberQ[funzlettere,testa],funzlettere,
         MemberQ[funzbooleani, testa], funzbooleani];

rami2 = Position[individuo2, x_ /; MemberQ[appartenenza, x]];
rami2 = DeleteCases[rami2, {0}];
rami2 = rami2 /. {x__, 0} -> {x};
```

Una volta trovati gli indirizzi sintatticamente corretti da scambiare i due figli possono essere creati.

3.4 Generazioni

Formalizzati tutti i passaggi fondamentali dell'algoritmo, si passa alla fase iterativa secondo lo schema descritto in Figura 2.1.

Nella creazione delle generazioni future è stata inizialmente trascurata la *mutazione*, limitandosi quindi al solo incrocio degli individui. Tale omissione ha permesso di constatare che, pur con probabilità molto bassa, la mutazione risulta fondamentale per la convergenza dell'algoritmo. Una delle possibili spiegazioni risiede nella struttura della soluzione stessa; in un problema di questo tipo l'espressione della fitness presenta un andamento molto piccato

in prossimità della soluzione corretta (in altre parole in generale si trovano programmi che funzionano o molto bene o molto male). Un andamento di questo tipo farà si che nella selezione dei genitori vengano scelti con alta probabilità solo ed esclusivamente i pochi individui migliori.

Per questo motivo in ogni generazione metà popolazione con fitness più bassa è stata sostituita con nuovi individui creati casualmente. Viene riportato qui sotto il codice atto all'introduzione di nuovi individui.

```
For[m = 1, m <= pop/2, m++,
  a = First[Position[fit1, Min[fit1]]];
  pop1 = Delete[pop1, posizione[a]];
  pop1 = Join[pop1, {individuo}];
  fit1 = Map[fitness, pop1];
  ];
```

Capitolo 4

Conclusioni

In questo capitolo vengono discussi i risultati ottenuti.
L'algoritmo è stato testato su 1000 situazioni iniziali ottenendo i risultati mostrati in Figura 4.1. Come si evince dal grafico è possibile individuare una

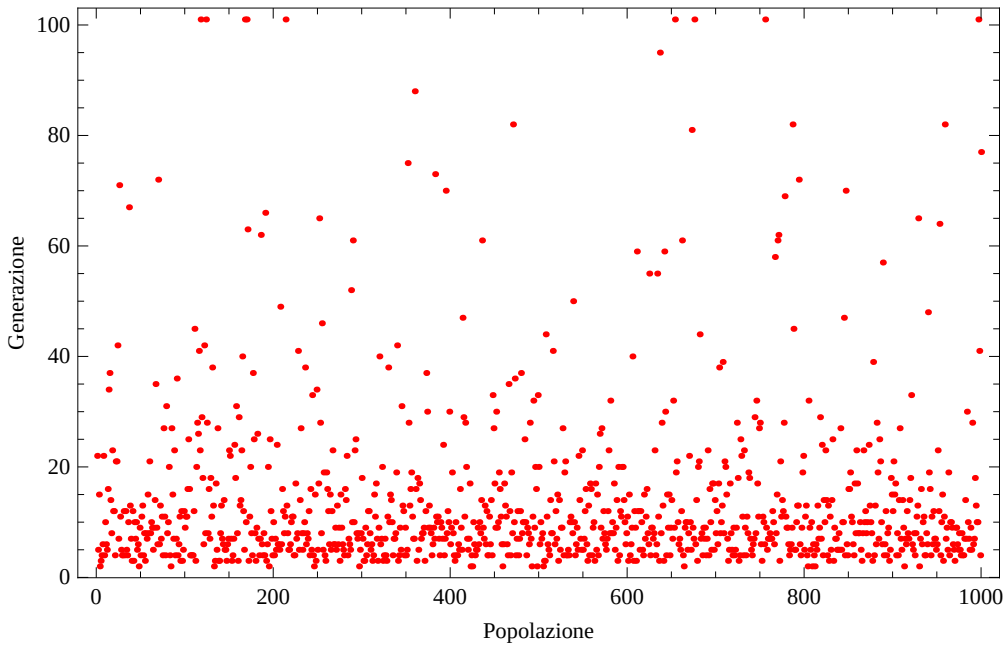

Figura 4.1: Nel grafico sono riportati i risultati ottenuti testando l'algoritmo su 1000 situazioni iniziali diverse.

zona di convergenza media sicuramente inferiore a 20.

Rimane comunque ancora una questione da discutere: per come è costruito l'algoritmo la soluzione trovata risolve sicuramente il caso analizzato ma

non si ha la certezza che la stessa soluzione possa essere utilizzata con un altra condizione iniziale. Per questo motivo nelle righe finali del codice è stata inserita una piccola parte in cui la soluzione trovata viene testata su altre configurazioni iniziali ottenendo quindi informazioni sulla validità o meno della soluzione trovata in termini generali.

4.1 Prospettive future

Il progetto sviluppato risulta essere un lavoro preliminare che può essere migliorato condiderevolmente.
Come si osserva le soluzioni trovate non sempre hanno un carattere generale ma sono strettamente legate alla condizione iniziale. Inoltre, spesso la combinazione di funzioni trovata non è la più veloce sia in termini di tempo che in termini di mosse effettuate.

Una delle possibili migliorie del programma potrebbe quindi essere quella di implementare una sezione atta a selezionare varie soluzioni verificandone l'universalità e individuandone quella che minimizzi il numero di mosse.