

Istituto Statale di Istruzione Superiore "Arturo Malignani"  
Liceo delle Scienze Applicate

# Algoritmi Genetici

Interpretare la natura attraverso l'informatica

Andrea Covre  
Classe 5<sup>a</sup> LSA F

Anno scolastico 2016 - 2017

# Indice

Introduzione .....	3
La passione per il codice .....	3
Il fascino degli algoritmi genetici.....	3
Processing.....	4
Obbiettivi.....	5
Ringraziamenti.....	6
1. Definizione.....	7
1.1 Terminologia .....	7
1.2 Tipologie di GAs.....	8
1.2.1 Algoritmo genetico tradizionale .....	8
1.2.2 Selezione Interattiva .....	9
1.2.3 Simulazione di un Ecosistema (o Ecosystem Simulation).....	9
2. La nascita degli algoritmi genetici.....	11
3. Teorema della scimmia instancabile .....	12
3.1 Dimostrazione .....	12
3.2 Probabilità .....	13
3.3 Amleto come il DNA.....	13
4. Teorema fondamentale degli algoritmi genetici .....	14
4.1 Ipotesi.....	14
4.2 Tesi.....	14
4.3 Dimostrazione .....	15
4.4 Conclusioni .....	15
5. I 3 principi dell'evoluzione di Darwin .....	16
5.1 Ereditarietà .....	16
5.1.1 Crossing-over paritario .....	16
5.1.2 Crossing-over casuale.....	17
5.1.3 Frammentazione.....	17
5.2 Variabilità .....	18
5.3 Selettività .....	18

5.3.1 Funzione di fitness .....	18
5.3.2 La scelta dei “genitori” .....	21
<b>6. Visualization of Evolution.....</b>	<b>22</b>
6.1 Inizializzazione della popolazione .....	22
6.2 Calcolo della fitness .....	23
6.3 Calcolo delle probabilità e selezione .....	24
6.4 Riproduzione .....	26
6.5 Mutazione.....	28
6.6 Iterazione.....	29
<b>7. Altri esempi di GAs.....</b>	<b>30</b>
7.1 The Traveling Salesman Problem.....	30
7.2 EvolveFlowField.....	31
7.3 evolutionMATH2 .....	31
7.4 Evolve Steering.....	32
7.5 Evolving a Human Face .....	32
<b>8. L'applicazione degli algoritmi genetici nel mondo reale.....</b>	<b>33</b>
8.1 Design automobilistico .....	33
8.2 Progettazione di antenne specifiche .....	33
8.3 Robotica .....	34
8.4 ANN Training .....	34
8.5 Altre applicazioni.....	35
<b>9. La probabilità che una scimmia componga Amleto .....</b>	<b>36</b>
<b>Conclusioni.....</b>	<b>39</b>
<b>Bibliografia .....</b>	<b>40</b>

# Introduzione

## La passione per il codice

Durante il primo anno delle superiori ho imparato i primi concetti fondamentali della programmazione, prima in HTML, e successivamente in PHP. In entrambi i linguaggi è piuttosto facile comprendere e metabolizzare le basi, ma ciò che ho trovato di davvero complesso e stimolante è la sintetizzazione di alcune "computazioni" che la nostra mente esegue in maniera automatica ed inconscia, in codice. Infatti, scrivere il codice di un programma che esegua una particolare funzione ci costringe ad analizzare il processo mentale che facciamo per eseguire quella stessa funzione, e ridurlo ad una sequenza di istruzioni basilari ed inequivocabili. Per esempio, se ci venisse detto di individuare il numero più alto all'interno di un insieme di 5 numeri, ci basterebbe dare una rapida occhiata all'insieme per sapere la risposta. I computer, però, sono ancora delle macchine stupide per ora, pertanto non sono capaci di "dare un'occhiata" per sapere la risposta. Il compito del programmatore è quindi quello di analizzare il proprio processo di "dare un'occhiata" per comprendere cosa esso significhi veramente, così da poterlo schematizzare in istruzioni semplici e compressibile per un computer. In questo caso "dare un'occhiata" significherebbe memorizzare il primo numero come il numero massimo; guardare poi il secondo numero e verificare se esso sia maggiore del numero che per noi è attualmente il massimo, se la risposta è sì, allora questo numero diventerà il nostro nuovo massimo; dopo di che si ripete il procedimento per tutti i numeri rimanenti.

A me piace considerare la programmazione come un atto, con cui si tenta di insegnare e trasmettere ad un computer alcuni propri concetti e modi di pensare, con i quali si possono concretizzare svariate idee e progetti.

Il concetto di mettere per iscritto il proprio processo mentale mi ha spinto ad approfondire e sviluppare le mie abilità di programmazione anche al di fuori dell'ambiente scolastico, imparando e scoprendo nuove tecniche e nuovi linguaggi di programmazione come C++, Processing e Node.js.

Ciò che mi appassiona della programmazione è anche la possibilità di applicare, e quindi comprendere meglio, concetti riguardanti la matematica, la fisica (applicando diverse leggi fisiche all'interno di una simulazione creando di fatto un Physics Engine) e la biologia (realizzando ad esempio un algoritmo genetico).

## Il fascino degli algoritmi genetici

Una sera, navigando su YouTube, mi imbattei in un video riguardante le simulazioni effettuate nel 1994 da Karl Sims sull'evoluzione di creature formate da dei blocchi, chiamato "*Karl Sims - Evolving Virtual Creatures With Genetic Algorithms*". In questo video venivano mostrate delle creature virtuali che, grazie all'evoluzione artificiale per mezzo di un algoritmo genetico, si sono sviluppate per riuscire a nuotare, camminare, saltare, inseguire un target e competere per un oggetto.

Ho trovato queste simulazioni assolutamente sbalorditive e straordinarie. Le creature che più mi hanno colpito sono state quelle che si sono evolute in ambiente acquatico perché, nonostante siano formate da semplici blocchi, ricordano molto la forma ed i movimenti delle anguille, dei girini e delle tartarughe.



Vedendo come un computer sia stato capace di evolvere dei gruppi di blocchi in veri e propri pseudo-animali, ho subito voluto approfondire l'argomento per capire come questo processo evolutivo artificiale potesse essere in grado di raggiungere risultati tanto complessi e accurati.

Grazie agli insegnamenti del mio mentore, Daniel Shiffman, professore d'arte associato del programma di telecomunicazioni interattive della New York University, ho compreso il funzionamento del meccanismo che c'è dietro ad ogni algoritmo genetico. Dopo di che ho iniziato a programmare alcuni algoritmi genetici sulle basi degli esercizi proposti dal professor Shiffman.

Una volta presa confidenza con tutte le componenti necessarie per l'evoluzione artificiale, ho deciso di realizzare i miei primi algoritmi genetici. Uno di questi è "*Visualization of Evolution*" (che verrà presentato successivamente in questo documento) che ho realizzato con lo scopo di semplificare la comprensione dell'evoluzione artificiale attraverso lo smascheramento dei suoi meccanismi interni.

## Processing

"*Visualization of Evolution*" e la maggior parte degli altri esempi di algoritmi genetici che verranno presentati in questo documento sono stati realizzati tramite Processing.

Processing è un linguaggio di programmazione Open-Source basato su JavaScript, famoso per le sue svariate applicazioni in ambito artistico ed educativo, e per il suo utilizzo nella realizzazione di sofisticate animazioni. Ciò che caratterizza questo linguaggio è la possibilità di programmare in un contesto visuale, ovvero di avere, ad ogni esecuzione del codice, un complesso e dinamico feedback visivo di immagini vettoriali.



Processing è nato durante la primavera del 2001 da un progetto di Casey Reas e Benjamin Fry, entrambi studenti al Media Lab del Massachusetts Institute of Technology, con lo scopo di creare uno strumento con il quale si potesse fondere l'informatica e la tecnologia con le arti visuali. Il progetto si è poi esteso fino a partecipare alla realizzazione di Arduino sulla base degli stessi ideali.

Oggi Processing non appartiene più all'M.I.T., ma è parte della “*The Processing Foundation*” dove Daniel Shiffman è il terzo membro fondatore, dopo Reas e Fry. “*The Processing Foundation*” ha l’obiettivo di “promuovere l’alfabetizzazione software nell’ambito delle arti visive e nei settori tecnologici correlati, così da renderli accessibili a persone dai diversi interessi e dalle diverse formazioni, per far imparare loro come programmare e realizzare progetti creativi”.

La community di Processing, al giorno d’oggi, conta migliaia di utenti fra artisti, studenti, educatori, scienziati, designers, web-devolpers, fotografi, architetti e animatori.

## Obbiettivi

Un obiettivo che mi sono posto scrivendo questo documento è quello di spiegare e dimostrare il funzionamento degli algoritmi genetici, nella maniera più esaustiva e completa possibile, proponendo occasionalmente degli esempi che possano trasmettere efficacemente i concetti più complessi.

Un altro obiettivo è quello di approfondire le mie conoscenze su questo argomento esplorando e raccogliendo le diverse tecniche che possono essere applicate all’interno di un algoritmo genetico, evidenziandone, se possibile, gli eventuali parallelismi con l’evoluzione biologica.

# **Ringraziamenti**

Vorrei dedicare questa breve sezione al mio mentore Daniel Shiffman che, nonostante il suo lavoro all'università di New York e alla sua impegnativa passione di creare video-lezioni e contenuti educativi gratuiti su YouTube, è comunque riuscito a dedicarmi del tempo per indicarmi risorse molto utili e per darmi preziosi consigli.

Oltre al supporto ricevuto per la realizzazione di questo documento, ringrazio Shiffman anche per essere stato un eccellente professore che è riuscito non solo a trasmettere a me, come a molte altre persone, la passione per la programmazione in maniera divertente e stimolante, ma anche per essere stato capace di scalare in modo straordinario l'enorme ostacolo relazionale che si pone tra studente ed insegnante attraverso la videocamera utilizzata per la registrazione delle video-lezioni.

# 1. Definizione

Gli algoritmi genetici (conosciuti anche come GAs) sono degli algoritmi euristici ispirati al principio di selezione naturale ed evoluzione biologica teorizzata nel 1895 da Charles Darwin. Il termine “genetico” deriva dal fatto che l’algoritmo sfrutta dei meccanismi concettualmente simili a quelli dei processi biologici scoperti dalla biologia e dalla genetica come il crossing-over, le mutazioni e la trasmissione di informazioni genetiche da genitore a prole.

L’obiettivo dei GAs è quello di ricavare la soluzione (o un’accurata approssimazione) a determinati problemi di ottimizzazione di vario ambito, partendo da delle soluzioni generate casualmente e successivamente ricombinandole, in maniera pseudo-casuale, ed eventualmente introducendovi nuovi elementi di disordine. In poche parole i GAs si occupano di far evolvere delle soluzioni non adatte, in soluzioni ottimali.

È opportuno, però, sottolineare che gli algoritmi genetici rappresentano un meccanismo evolutivo puramente informatico, molto meno complesso rispetto all’evoluzione che avviene in natura e ai corrispondenti fenomeni biochimici, pertanto ci possono essere alcune leggere discrepanze tra, i termini e i processi biologici, e quelli utilizzati per questa classe di algoritmi.

## 1.1 Terminologia

Quando si discute degli algoritmi genetici si tende convenzionalmente ad utilizzare i termini della biologia e della genetica, così da evidenziare ulteriormente la forte analogia che c’è tra questa classe di algoritmi e la teoria dell’evoluzione di Darwin.

### **Individuo**

Questo termine identifica l’insieme di informazioni, dati e valori che caratterizzano una possibile soluzione al problema in questione. Nel parallelismo con l’evoluzione di Darwin, un individuo può rappresentare un essere vivente, che a sua volta è anch’esso insieme di informazioni genetiche.

### **Popolazione**

L’insieme di individui che costituiscono una determinata generazione.

### **Generazione**

Indica la popolazione appartenente ad un preciso momento storico dell’esecuzione dell’algoritmo.

### **DNA**

L’insieme di tutte le informazioni di un individuo.

### **Gene**

Un tratto del DNA che determina una caratteristica specifica di un individuo.

### **Fitness**

Valore che indica quanto, un individuo, rappresenti una soluzione adatta al problema in questione (in genetica, invece, la fitness indica il successo riproduttivo di un organismo). Gli individui con una fitness più alta risultano più adatti e quindi avranno maggiori probabilità di generare una prole. Infatti, Darwin, nella sua teoria dell’evoluzione, afferma che l’individuo che ha più probabilità di sopravvivere, e quindi di riprodursi, è quello che risulta più adatto all’ecosistema in cui si trova.

## **Selezione**

Processo tramite il quale si scelgono i “*genitori*” della generazione successiva.

## **Riproduzione**

Processo con il quale si genera un nuovo individuo partendo da uno o più individui appartenenti alla generazione precedente.

## **Mutazione**

Cambiamento casuale di un gene che può avvenire prima della conclusione della generazione di un nuovo individuo.

## **Tasso di mutazione**

Probabilità con cui ogni gene può subire una mutazione, convenzionalmente indicata in percentuale.

## **Search space**

Tutte le possibili soluzioni al problema, pertanto ogni popolazione è un sottoinsieme del search space.

# **1.2 Tipologie di GAs**

Gli algoritmi genetici possono essere sviluppati con molteplici modelli e tecniche, ma generalmente si possono individuare tre sottoclassi principali in base a come vengono selezionati gli individui per la generazione di soluzioni migliori:

## **1.2.1 Algoritmo genetico tradizionale**

Questa tipologia di algoritmi genetici è quella più largamente utilizzata. Questi algoritmi furono sviluppati per risolvere quel genere di problemi in cui un approccio di “*forza bruta*” impiegherebbe semplicemente troppo tempo per trovare la soluzione al problema.

Consideriamo il seguente esempio: dobbiamo indovinare un numero compreso tra 1 e 1.000.000.000 che un computer ha generato casualmente (fingiamo che il computer abbia generato il numero 425.983.911).

Un approccio di “*forza bruta*” ci indurrebbe a verificare tutti i numeri presenti nel dato intervallo partendo da 1:

è 1? → IL NUMERO NON È CORRETTO

è 2? → IL NUMERO NON È CORRETTO

è 3? → IL NUMERO NON È CORRETTO

e così via...

Nel nostro caso la soluzione è 425.983.911, quindi dovremmo verificare tutti i numeri presenti da 1 fino a 425.983.911 (ovvero fino a quando il computer che ha generato il numero ci conferma che il numero che abbiamo selezionato rappresenta effettivamente la soluzione) e questo processo risulterebbe estremamente dispendioso sia in termini di risorse che in termini di tempo.

Cosa cambierebbe se il computer che ha generato il numero, invece che darci un output binario (“il numero è corretto” o “il numero non è corretto”), ci potesse dire non solo se la nostra ipotesi è corretta oppure no, ma anche se la nostra ipotesi è più o meno “buona”? Se questo fosse possibile, allora basterebbe cominciare verificando alcuni numeri a caso e poi utilizzare gli output del computer per fare ipotesi via via sempre più intelligenti (più evolute), per esempio:

è 180.000.000? → NO

è 530.000.000? → NO, MA SEI PIÙ VICINO ALLA SOLUZIONE

è 950.000.000? → NO, SEI MOLTO LONTANO DALLA SOLUZIONE  
è 450.000.000? → NO, MA SEI PIUTTOSTO VICINO ALLA SOLUZIONE

Dopo qualche tentativo si riesce facilmente a convergere verso la soluzione finale in tempi considerevolmente più brevi rispetto al metodo “*forza bruta*”.

Gli algoritmi genetici tradizionali utilizzano questa tecnica per individuare quali siano le soluzioni più promettenti così da poterle selezionare come i genitori della generazione successiva.

La maggior parte dei programmi che verranno presi in considerazione, in particolare il programma esplicativo *Visualization of Evolution*, sono basati su questa tipologia di algoritmi.

### 1.2.2 Selezione Interattiva

Questa tipologia di algoritmi genetici è utilizzata per generare, di generazione in generazione, una popolazione che rispecchi sempre di più le scelte dell’utente o dell’agente che effettua la selezione.

Proseguiamo con un esempio. Supponiamo di entrare un museo virtuale d’arte dove sono stati generati casualmente cinque quadri. Ogni quadro rappresenta un paesaggio diverso. Ci viene richiesto di selezionare il quadro che ci piace di più e pertanto selezioniamo il quadro che rappresenta una foresta (sebbene questo quadro non ci piaccia molto è comunque quello che ci piace di più). Ora ci vengono presentati cinque quadri nuovi, tutti rappresentanti una foresta, ma ognuno con delle differenze (alberi più alti o più bassi, foresta più o meno fitta, cielo sereno o nuvoloso). Ci viene nuovamente richiesto di scegliere un quadro e scegliamo quello con il cielo nuvoloso e gli alberi bassi. Cinque nuovi quadri vengono generati, la maggior parte di essi rappresenta una foresta con alberi alti e il cielo nuvoloso.

Questo esempio, seppur estremamente semplificato, dimostra come un algoritmo genetico basato sulla Selezione Interattiva tenda a creare degli individui (in questo caso quadri) che rispecchiano le preferenze dell’utente. Infatti siamo partiti da un insieme di quadri che non ci piacevano molto, ma dopo alcune iterazioni i quadri cominciano a piacerci sempre di più, in quanto ad ogni selezione/iterazione assegnamo un certo valore di fitness a determinate caratteristiche del quadro stesso.

Questa categoria di algoritmi genetici viene principalmente impiegata nei social media, nei motori di ricerca e nel web advertising. Infatti quando navighiamo online e clicchiamo la pubblicità di un prodotto piuttosto che su un altro, non facciamo altro che svolgere il nostro compito di utente nella Selezione Interattiva di un possibile algoritmo genetico di questa tipologia. Con le successive selezioni l’algoritmo comincia ad imparare quali colori, quali generi di prodotto e quali marche preferiamo. Questo è il motivo per il quale quando navighiamo su internet tendiamo a vedere sempre più pubblicità che riflettono i nostri eventuali acquisti precedenti, unitamente a contenuti, articoli o notizie che rispecchiano i nostri gusti ed interessi.

### 1.2.3 Simulazione di un Ecosistema (o Ecosystem Simulation)

Questa categoria di GAs è forse quella più affascinante e complessa. Il principio di questi algoritmi è quello di testare gli individui all’interno di un vero e proprio ecosistema virtuale.

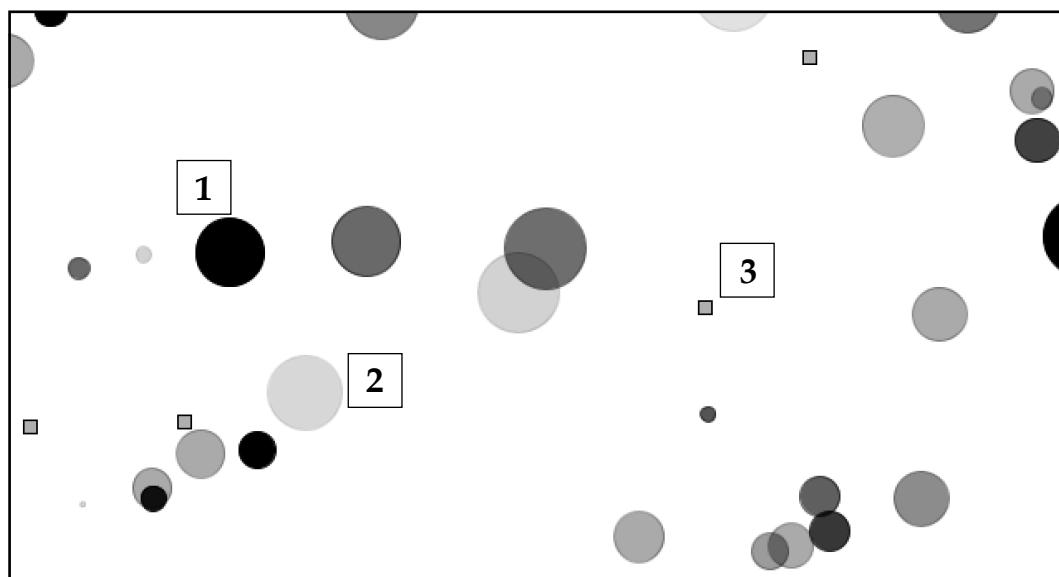
Le principali differenze tra questo genere di algoritmi e gli altri sono che attraverso la simulazione di un ecosistema è possibile testare tutti gli individui contemporaneamente all’interno di un sistema virtuale, ciò significa che l’evoluzione delle generazioni successive è fortemente influenzata da come i vari individui si comportano ed interagiscono all’interno del sistema.

Questi algoritmi si avvicinano molto al concetto di evoluzione naturale, tanto è vero che, in alcuni programmi, gli individui sembrano dei veri e propri esseri pseudo-viventi. In natura una popolazione non nasce e non si riproduce nel medesimo istante (come accade nella maggioranza dei GAs) e di fatto non si riproduce in base ad una fitness calcolata rigorosamente secondo una determinata formula (causando la “dominazione” del più adatto), ma piuttosto si riproducono gli individui che, grazie a determinate condizioni e casualità, riescono a sopravvivere più a lungo nell’ecosistema nel quale si trovano.

Per esemplificare il concetto consideriamo il seguente algoritmo.

Immaginiamo che in un ecosistema virtuale ci sia una popolazione composta da un dato numero di individui e da alcune particelle che identificano del “cibo”.

- Ogni individuo è un cerchio con un raggio di grandezza casuale compresa tra 0 e 100.
- Affinché un individuo riesca a sopravvivere deve “mangiare” delle particelle di cibo.
- Una particella di cibo, per essere mangiata, necessita di entrare in contatto con un qualsiasi individuo.
- Ogni individuo si muove in maniera pseudo-casuale secondo l’andamento di una funzione di Perlin Noise (o rumore di Perlin).
- Più l’individuo è grande, minore è la sua velocità di movimento e maggiore è la sua superficie di contatto con eventuali particelle di cibo.
- Più l’individuo è piccolo, maggiore è la sua velocità di movimento e minore è la sua superficie di contatto con eventuali particelle di cibo.
- Ogni individuo ha, in ogni momento, una probabilità del 0,05% di riprodursi in maniera asessuata, generando un clone che può essere sottoposto ad una mutazione casuale.
- Un individuo muore solamente nel caso in cui il suo valore di “salute” raggiunge lo zero (ovvero muore di fame).
- Il cibo viene generato in maniera casuale nell’ecosistema e ogni qualvolta che un individuo muore.



1. Individuo con il valore di salute al massimo
2. Individuo con il valore di salute basso e quindi prossimo a “morire”
3. Particella di cibo

In questo esempio non è necessario calcolare la fitness in quanto essa è semplicemente rappresentata da quanto tempo un individuo riesce a sopravvivere: più a lungo un individuo sopravvive, maggiori sono le possibilità che riesca a generare uno o più "figli". È importante notare che il numero di individui all'interno di questa popolazione non è costante, ma può variare, pertanto possiamo imbatterci anche nei due scenari più estremi: si può verificare un sovrappopolamento come si può verificare l'estinzione dell'intera popolazione.

Il vantaggio di testare gli individui all'interno di un ecosistema è anche quelli di potervi inserire il ruolo di prede e predatori, veleno e cibo e molte altre variabili che aumentano la complessità dell'algoritmo.

## 2. La nascita degli algoritmi genetici

Nel 1950, Alan Turing propose una "learning machine" che potesse sfruttare i principi dell'evoluzione. Simulazioni computerizzate dell'evoluzione cominciarono nel 1954 con il lavoro di Nils Aall Barricelli, che utilizzò il computer dell'Institute for Advanced Study di Princeton, New Jersey.

Partendo dal 1957, il genetista quantitativo Alex Fraser pubblicò una serie di articoli riguardanti simulazioni della selezione artificiale di organismi con molteplici loci, controllando un tratto misurabile. Da queste basi, la simulazione dell'evoluzione divenne sempre più nota e sperimentata dai biologi all'inizio degli anni '60, e i metodi e le tecniche furono raccolte nei libri di Fraser e Burnell (1970) e successivamente da Crosby (1973). Le simulazioni di Fraser includevano tutti gli elementi essenziali dei moderni algoritmi genetici. Hans-Joachim Bremermann fu inoltre il primo ad adottare una popolazione di soluzioni a problemi di ottimizzazione, che veniva sottoposta a ricombinazione, mutazione e selezione.

Sebbene Barricelli utilizzasse i suoi primi algoritmi genetici per sviluppare un automa in grado di giocare a semplici videogames, l'evoluzione artificiale divenne largamente riconosciuta come valido metodo per la risoluzione di problemi di ottimizzazione, come fu dimostrato dai lavori di Ingo Rechenberg e Hans-Paul Schwefel negli anni '60 e '70. Il gruppo di Rechenberg fu, infatti, capace di risolvere complessi problemi ingegneristici attraverso strategie evoluzionistiche.

Nel 1965 la tecnica di programmazione evolutiva di Lawrence J. Fogel fu proposta come metodo di generazione di intelligenza artificiale (AI).

Gli algoritmi genetici divennero particolarmente conosciuti negli anni '70 grazie ai lavori di John Holland ed in particolare al suo libro "*Adaptation in Natural and Artificial Systems*" (1975). I suoi lavori cominciarono dallo studio dei Cellular Automata presso l'University of Michigan. Holland formalizzò il suo lavoro dimostrando un teorema, capace di prevedere la qualità della generazione successiva in un algoritmo genetico, chiamato "**teorema degli schemi di Holland**" (Holland's Schema Theorem) che discuteremo in maniera più approfondita nella sezione 4.

La ricerca sugli algoritmi genetici è rimasta largamente teorica fino alla metà degli anni '80, quando la Prima Conferenza Internazionale sugli Algoritmi Genetici ebbe luogo a Pittsburgh, Pennsylvania. Al giorno d'oggi, invece, l'evoluzione artificiale viene messa in pratica in molti ambiti diversi, alcuni dei quali verranno discussi nella sezione 8 di questo documento.

# 3. Teorema della scimmia instancabile

Il “teorema della scimmia instancabile” (o infinite monkey theorem) afferma che una scimmia che prema a caso i tasti di una macchina da scrivere, per un tempo infinitamente lungo, quasi certamente riuscirà a comporre un qualsiasi testo prefissato come potrebbe essere Amleto, uno dei più famosi lavori di Shakespeare.

Si può facilmente osservare come sia estremamente difficile che la scimmia riesca casualmente a comporre l’Amleto.



## 3.1 Dimostrazione

Ricordiamo che se due eventi sono statisticamente indipendenti, allora la probabilità che entrambi accadano è uguale al prodotto delle probabilità che accada ciascun evento in maniera indipendente.

Supponiamo (per semplicità) che la macchina da scrivere abbia solo 27 tasti, ovvero lo spazio e le 26 lettere dell’alfabeto, e che la parola che deve essere scritta sia “banana”.

La probabilità di ogni tasto di essere premuto dalla scimmia è quindi pari a 1 su 27, ed è indipendente rispetto agli altri tasti. Inoltre la parola “banana” è composta da 6 caratteri quindi questi specifici eventi determinati dalla parola “banana” devono verificarsi 6 volte consecutive affinché la parola in questione venga scritta dalla scimmia.

Data quindi una tastiera di  $m$  tasti e un testo da riprodurre di  $k$  battute, la probabilità di non effettuarlo in  $n$  tentativi (indipendenti) è:

$$\left(1 - \frac{1}{m^k}\right)^n$$

Se concediamo alla scimmia un tempo infinito, allora la probabilità di non effettuare il testo dato in un numero infinito di tentativi (indipendenti) è:

$$\lim_{n \rightarrow \infty} \left(1 - \frac{1}{m^k}\right)^n = 0$$

Ciò significa che, se si prova per un numero infinitamente grande di volte, la probabilità di scrivere un testo di lunghezza prefissata  $k$ , è 1, ovvero l’evento è certo.

## 3.2 Probabilità

La probabilità che la scimmia scriva il primo carattere corretto è 1 su 27, che scriva i primi due caratteri corretti è 1 su 729. Dato che le probabilità diminuiscono esponenzialmente, la probabilità di scrivere solo i primi 10 caratteri in maniera corretta si riduce a 1 su  $2,06 \cdot 10^{14}$ .

Possiamo quindi notare come le probabilità che la scimmia riproduca l'Amleto sia straordinariamente piccola, da essere quasi concettualmente inimmaginabile e che di fatto richiederebbe una considerevole potenza di calcolo per essere anche semplicemente calcolata.

Sapendo infatti che l'Amleto è composto da circa 130.000 lettere, allora la probabilità che la scimmia componga correttamente il testo al primo tentativo è pari a 1 su  $27^{130.000}$ . Se provassimo a calcolare  $27^{130.000}$  genereremmo un numero di ben 186.078 cifre, e questo ignorando completante la differenza tra lettere maiuscole e minuscole, e tutti i segni di punteggiatura.

Per chi fosse curioso ed interessato a comprendere l'enormità del numero di cui stiamo discutendo, ho riportato il risultato di  $27^{130.000}$  nella sezione 9 di questo documento. Per rendere meglio l'idea possiamo anche considerare il caso in cui ogni protone dell'universo sia una scimmia che ha cominciato a premere i tasti dal Big Bang fino alla fine dell'universo: in tali circostanze è altamente probabile che nessuna scimmia abbia mai composto neppure i primi 5000 caratteri.

## 3.3 Amleto come il DNA

Il teorema della scimmia instancabile non è altro che una metafora: la scimmia rappresenta un qualsiasi fenomeno che genera una serie di eventi casuali e l'Amleto una sequenza ben ordinata di informazioni.

Doug Powell, un apologo cristiano, ha affermato che anche se la scimmia battesse accidentalmente le lettere di Amleto, fallirebbe comunque nel tentativo di produrlo, in quanto non ne aveva intenzione. L'intenzione di Powell era quella di dimostrare che le leggi naturali non sarebbero mai in grado di produrre le informazioni contenute nel nostro DNA. Un'altra comune tesi è quella presentata dal reverendo John F. MacArthur: le mutazioni genetiche necessarie a produrre un verme a partire da un'ameba sono tanto improbabili quanto è improbabile che una scimmia riproduca il soliloquio di Amleto, e pertanto l'inattuabilità dell'evoluzione non può essere superata.

Il biologo evolutivo Richard Dawkins utilizza il concetto della scimmia instancabile nel suo libro “*The Blind Watchmaker*” (L'Orologio Cieco), per dimostrare l'abilità della selezione naturale di produrre complessità biologica partendo da mutazioni casuali.

In un esperimento di simulazione, Dawkins produsse la frase di Amleto “*methinks it is like a weasel*” partendo da una frase digitata casualmente e facendo “riprodurre” le successive generazioni scegliendo sempre la frase che combaciava maggiormente con il target prescelto. Come abbiamo visto precedentemente, le probabilità che la frase compaia al primo tentativo sono estremamente piccole, tuttavia Dawkins dimostrò che il target poteva essere riprodotto piuttosto rapidamente (40 generazioni) utilizzando la selezione cumulativa delle frasi.

Gli individui di partenza, che vengono generati casualmente, forniscono il materiale grezzo, mentre la selezione cumulativa trasmette le informazioni.

Dawkins afferma che il suo esperimento è comunque una simulazione imperfetta dell'evoluzione, dato che ogni individuo viene selezionato in base ad un “*distant*” ed “*ideale*” target. Infatti, Dawkins sostiene che l'evoluzione non ha piani a lungo termine e pertanto non progredisce verso un ideale e distante obiettivo (come gli umani).

La sua simulazione aveva piuttosto lo scopo di mostrare le potenzialità della selezione cumulativa non casuale. Ovvero, riportando il discorso nei termini del teorema della scimmia instancabile, Amleto potrebbe essere riprodotto in tempi relativamente brevi se si utilizza un metodo di selezione non casuale basato sulla teoria di Darwin, perché il valore di fitness tenderà a preservare le lettere che, casualmente, combaciano col target migliorando di conseguenza la qualità delle generazioni successive.

(Questo potrà essere verificato attraverso il programma "Visualization of Evolution" che ho scritto, e che simula l'evoluzione di stringhe di caratteri generate a caso, verso un target preimpostato, mostrando passo dopo passo il funzionamento dell'algoritmo)

Con il concetto di Dawkins in mente possiamo facilmente notare come i fenomeni chimico-fisici (che conferiscono un certo livello di casualità molecolare) ricoprono il compito di scimmia instancabile, mentre il DNA ricopre il ruolo di Amleto.

## 4. Teorema fondamentale degli algoritmi genetici

Il teorema fondamentale degli algoritmi genetici (**teorema degli schemi di Holland**) è oggi considerato uno dei teoremi più importanti per dimostrare la potenzialità dei GAs, inoltre evidenzia le intuizioni di Dawkins relative alla conservazione dei "frammenti" della soluzione finale al problema in questione, grazie al valore di fitness.

### 4.1 Ipotesi

Dato un algoritmo genetico, le informazioni contenute all'interno dei suoi cromosomi possono essere categorizzate ed identificate con particolari stringhe dette schemi. Gli schemi consistono di un insieme di metacaratteri (wildcards) riconosciuti come validi e da un insieme di simboli prelevati dall'alfabeto di codifica utilizzato. Per una maggiore comprensione di quanto affermato è opportuno procedere con un esempio. A tale scopo consideriamo:

- Una codifica binaria ed i suoi simboli di codifica 0 e 1.
- Il metacarattere (wildcard) \* che identifica un valore qualsiasi tra quelli ammessi dalla codifica.
- Tre generiche soluzioni codificate in forma di cromosoma binario A = 1001101, B = 1000100 e C = 0000010.
- Uno schema S = 100\*.

In questo esempio, lo schema identifica tutte le sottostringhe di A, B e C che cominciano con 100 e che sono lunghe quattro caratteri ossia: la sottostringa 1001 appartenente ad A e la sottostringa 1000 appartenente a B. C, a differenza di A e di B, non possiede sottostringhe identificate dallo schema.

### 4.2 Tesi

All'interno dei cromosomi sono codificati dei passi elementari che permettono la risoluzione parziale del problema (se non dovessero essere presenti, prima o poi verrebbero comunque introdotti e codificati grazie alle mutazioni). Questi passi, se sopravvivono e vengono

sufficientemente ricombinati, si ampliano e permettono di giungere ad una soluzione completa e definitiva. Delle simili informazioni, rappresentabili con degli *schemi*, potrebbero essere definite come "blocchi costruttori".

## 4.3 Dimostrazione

Basandosi sul fatto che durante l'esecuzione di un algoritmo genetico, valutando le stringhe relative alla codifica vettoriale delle soluzioni del problema considerato, si valuta implicitamente il valore di fitness di diversi schemi, con qualche calcolo matematico si può dedurre la formula che permette di approssimare il numero atteso del valore di fitness dello schema successivo.

La formula che si ricava da questa considerazione è la seguente:

$$M(H_i, t + 1) \geq M(H_i, t) \cdot \left[ \frac{f(H_i)}{\bar{f}} \right] \cdot \left[ 1 - p_c \cdot \frac{\delta(H_i)}{l - 1} \right] \cdot \left[ (1 - p_m)^{o(H_i)} \right]$$

dove:

- $f(H_i)$  è la media dei parametri associati dalla funzione di fitness alle stringhe identificate dall'i-esimo schema.
- $\bar{f}$  è la media dei parametri associati dalla funzione di fitness alle soluzioni della popolazione considerata.
- $\delta(H_i)$  è la lunghezza dell'i-esimo schema considerato.
- $l$  è la lunghezza delle stringhe nello spazio di ricerca.
- $o(H_i)$  è il numero di bit conosciuti nello schema considerato.
- $p_m$  e  $p_c$  sono le probabilità di mutazione e di crossover.

La formula sopra riportata indica che, quando esistono stringhe caratterizzate da schemi corti, alto parametro di fitness e basso ordine ci sono maggiori probabilità di sopravvivenza. Queste caratteristiche permetterebbero la ricombinazione di stringhe simili che a loro volta andrebbero a generare altre stringhe linearmente più lunghe, ma con un valore di fitness esponenzialmente più alto sulle quali la formula potrebbe essere nuovamente applicata.

## 4.4 Conclusioni

Quanto evidenziato dalle precedenti argomentazioni, sancisce l'esistenza e la forma dei *blocchi costruttori* inizialmente teorizzati, che rappresentano poi l'essenza e la natura degli algoritmi genetici, ossia la possibilità di ricombinare soluzioni parziali (esistenti o generate per mutazione) attraverso la selezione non casuale di Darwin, per poi giungere ad una soluzione completa e finale.

# 5. I 3 principi dell'evoluzione di Darwin

Per comprendere pienamente il funzionamento di un algoritmo genetico, è opportuno discutere tre principi fondamentali della teoria dell'evoluzione di Darwin che devono essere presenti in ogni simulazione affinché possa verificarsi un fenomeno evolutivo simile a quello che avviene in natura.

## 5.1 Ereditarietà

È necessario che ci sia un processo con il quale i "genitori" possano trasmettere le proprie informazioni genetiche ai "figli". In natura questo avviene grazie al DNA, in *Visualization of Evolution* questo avviene grazie ad un array dove ad ogni indice (gene) è presente un codice ASCII (genotipo) rappresentante una lettera maiuscola o lo spazio (fenotipo).

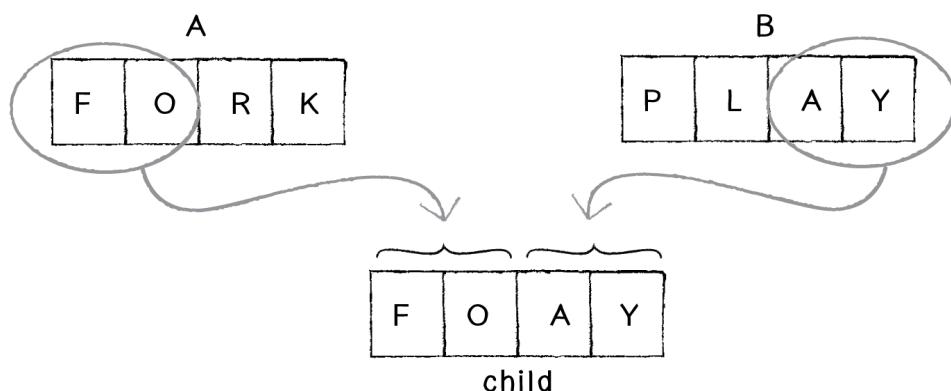
Come in natura, anche negli algoritmi genetici esistono due modi con cui un individuo si può riprodurre e quindi trasmettere il proprio patrimonio genetico alla generazione successiva.

- **Riproduzione asessuata:** un individuo non necessita di un "partner" per riprodursi, e pertanto il codice genetico della prole sarà identico a quello del genitore (a meno che non avvenga una mutazione).
- **Riproduzione sessuata:** un individuo necessita di almeno un altro individuo per potersi riprodurre e la prole avrà una ricombinazione delle informazioni genetiche dei genitori.

La ricombinazione del DNA dei genitori, nella riproduzione sessuata, può essere eseguita attraverso tre tecniche principali: crossing-over paritario, crossing-over casuale e frammentazione.

### 5.1.1 Crossing-over paritario

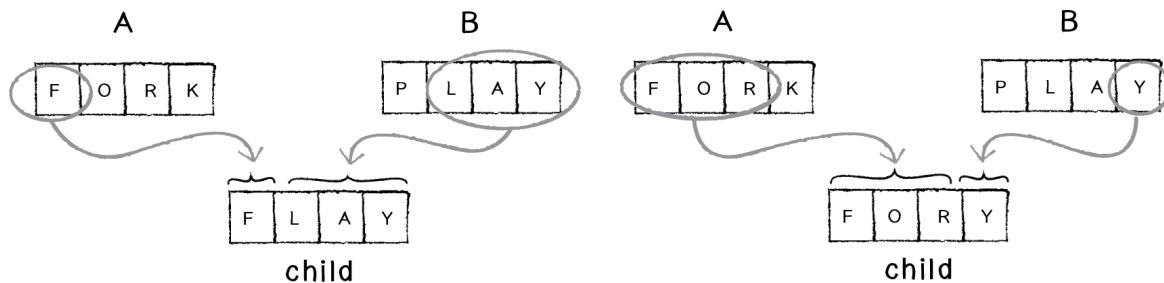
Questa tecnica si basa sul semplice concetto di prendere metà patrimonio genetico da un genitore e metà dall'altro.



Questa tecnica produrrà sempre lo stesso risultato, ovvero "FOAY".

### 5.1.2 Crossing-over casuale

Con questa tecnica si procede scegliendo in maniera casuale un punto della struttura del DNA dove si vuole applicare il crossing-over. Dopo di che si assegna alla prima sezione del DNA del figlio, la sezione corrispondente del DNA di un genitore, e alla seconda sezione, si assegna invece, la sezione corrispondente del DNA del secondo genitore.

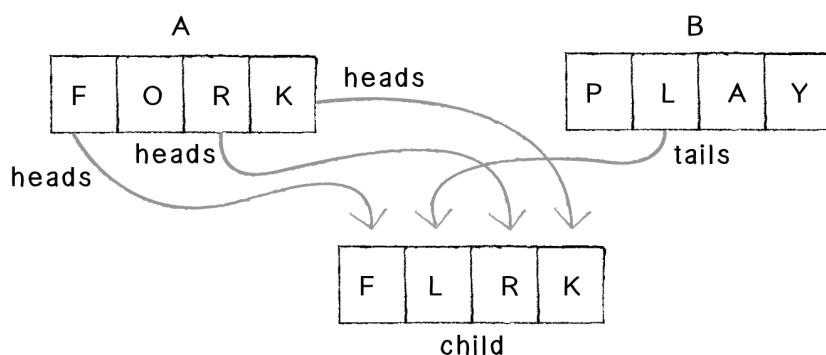


Questa tecnica può produrre diversi risultati; nel caso mostrato in esempio, i risultati possono essere tre (lunghezza della sequenza - 1): "FLAY", "FOAY" e "FORY". Notiamo che gli estremi, sinistro di un genitore (F) e destro dell'altro (Y), vengono sempre trasmessi.

### 5.1.3 Frammentazione

Questa tecnica può anche essere interpretata come una molteplice applicazione del crossing-over, tuttavia il processo di applicazione è differente.

Ogni gene del figlio (in questo caso la posizione delle lettere all'interno della sequenza) può ereditare le informazioni (le lettere) o dal genitore A oppure dal genitore B. Quindi ogni informazione del figlio avrà il 50% di probabilità di essere stata ereditata da un genitore piuttosto che dall'altro. Per semplificare l'idea, immaginiamo di dover creare un nuovo individuo partendo da due genitori composti da una sequenza di 4 lettere ognuno. Per creare il figlio possiamo, ad esempio, tirare una moneta: se esce testa allora la prima lettera del figlio sarà ereditata dal genitore A, mentre se esce croce la prima lettera sarà ereditata dal genitore B. Nel nostro caso esce prima testa, e quindi la prima lettera verrà ereditata dal genitore A, poi croce, la seconda lettera viene ereditata dal genitore B, poi esce per due volte di fila testa, e quindi le ultime due lettere vengono ereditate dal genitore A.



Possiamo notare che con questa tecnica si può generare un numero superiore di risultati diversi rispetto a quelle precedenti, infatti in questo caso si possono generare fino a 16 sequenze diverse: (numero di genitori)<sup>(lunghezza della sequenza)</sup> =  $2^4 = 16$ .

È opportuno precisare che in natura, il crossing-over, non avviene durante il processo riproduttivo, ma durante il processo di formazione dei gameti, più precisamente durante la meiosi.

## 5.2 Variabilità

Affinché una popolazione possa evolvere è necessario che ci sia un certo livello di variabilità all'interno del proprio patrimonio genetico. Infatti, se avessimo una popolazione di farfalle dove tutti gli individui hanno gli stessi colori, la stessa grandezza, la stessa forma di ali ed ogni altra caratteristica identica, allora i loro figli non potranno che essere identici ai propri genitori e di conseguenza anche al resto della popolazione. In uno scenario del genere non si potrebbe mai verificare un fenomeno evolutivo.

In natura e negli algoritmi genetici la variabilità genetica è garantita dalle mutazioni. In natura le mutazioni possono essere causate da fattori esterni, come radiazioni cosmiche o agenti mutageni, oppure da fattori interni, come errori di duplicazione del DNA. Negli algoritmi genetici, invece, le mutazioni avvengono secondo un tasso di mutazione che è stato predefinito, e quando un gene subisce una mutazione, il valore corrispondente viene generalmente sostituito da uno generato casualmente.

È facile intuire che se, sia in natura che nei GAs, la probabilità di mutazione è nulla, allora la popolazione, dopo alcune generazioni, converge verso uno stesso patrimonio genetico. Mentre se la probabilità di mutazione è piuttosto elevata, allora si verificherà una forte instabilità nel patrimonio genetico, dove le informazioni che rappresentano parte della soluzione vengono continuamente perse, rendendo così impossibile il processo di ottimizzazione. È pertanto fondamentale trovare il corretto tasso di mutazione quando si progetta un GA, in modo da bilanciare nella maniera più efficace possibile la generazione di nuovi possibili frammenti della soluzione, con il numero di perdite dei frammenti utili già esistenti.

## 5.3 Selettività

Affinché una popolazione si evolva verso un target e si ottimizzi, è necessario che solo gli individui più adatti riescano a riprodursi. Nella natura questa selezione è principalmente effettuata dall'ecosistema; per esempio, gli individui più lenti di una popolazione di gazzelle vengono eliminati dai leoni, e solo le giraffe col collo abbastanza lungo riusciranno a mangiare sufficientemente a lungo fino a riprodursi. Questo tipo di selezione si verifica solo negli algoritmi genetici a simulazione di ecosistema, mentre negli altri la selezione è dettata dalla funzione di fitness e dal metodo di scelta dei "genitori".

### 5.3.1 Funzione di fitness

La fitness di ogni individuo viene calcolato attraverso una funzione ben strutturata, dove si premiano gli individui che risultano più adatti e che quindi portano nel proprio DNA un maggior numero di frammenti della soluzione finale. È fondamentale utilizzare una funzione di fitness appropriata per ogni algoritmo genetico. Infatti, affinché si verifichi un fenomeno evolutivo di qualità, è necessario che i frammenti utili, all'interno di pochi individui, si propaghino nel resto della popolazione, e che invece si eviti la propagazione di informazioni genetiche non ottimali a discapito di quelle ottimali.

Consideriamo l'esempio di un algoritmo genetico che ha lo scopo di far evolvere delle sequenze casuali di lettere verso una sequenze predeterminata (*Visualization of Evolution* ha lo stesso scopo evolutivo).

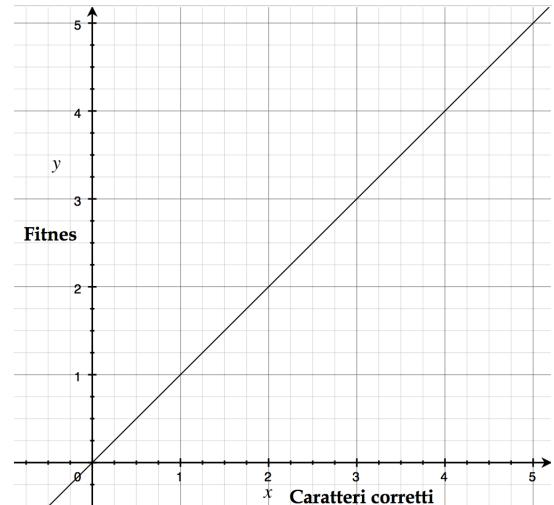
In questo caso la fitness verrà semplicemente calcolata in base a quante lettere corrispondono al target. È tuttavia possibile e fondamentale decidere quanto vantaggio concedere in più a determinati individui che rappresentano una soluzione parziale ma più completa, rispetto agli individui che rappresentano una soluzione parziale ma leggermente meno completa.

Discutiamo ora tre possibili funzioni di fitness che possono essere utilizzate in questo scenario, ricordando che in un GA non è tanto importante il valore di fitness in sé affinché un individuo si riproduca, ma piuttosto come quel valore di fitness si relaziona rispetto a quello degli altri individui.

- **Fitness lineare:** la fitness viene calcolata con una funzione lineare, ovvero ad un individuo viene assegnato un punto di fitness per ogni carattere corretto.

$$\text{fitness} = (\text{numero caratteri corretti})$$

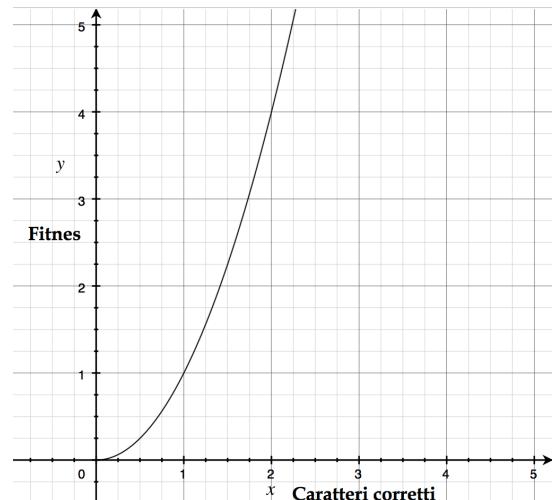
Caratteri corretti	Fitness
1	1
2	2
3	3
4	4
5	5



- **Fitness quadratica:** la fitness viene calcolata con una funzione quadratica, ovvero ad un individuo viene assegnato un valore di fitness pari al quadrato del numero di caratteri corretti.

$$\text{fitness} = (\text{numero caratteri corretti})^2$$

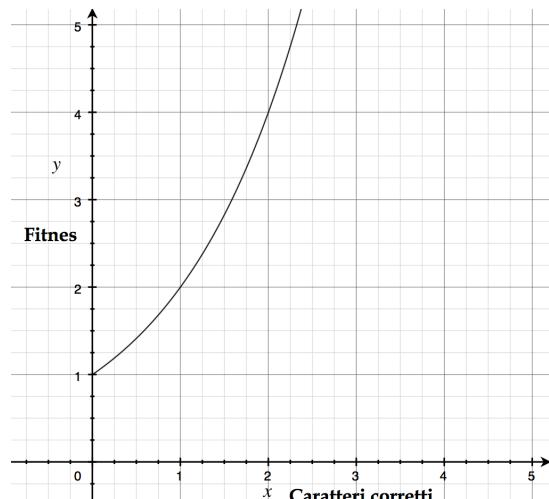
Caratteri corretti	Fitness
1	1
2	4
3	9
4	16
5	25



- **Fitness esponenziale:** la fitness viene calcolata con una funzione esponenziale, ovvero, ad un individuo viene assegnato un valore che raddoppia (nel caso in cui la base sia 2) per ogni carattere corretto.

$$\text{fitness} = 2^{(\text{numero caratteri corretti})}$$

Caratteri corretti	Fitness
1	2
2	4
3	8
4	16
5	32



La fitness quadratica ed esponenziale assegnano un punteggio maggiore per ogni carattere corretto in più, rispetto a quello lineare che assegna solo un punto per carattere. Inoltre la fitness esponenziale cresce più velocemente rispetto a quella quadratica.

È molto importante considerare questi concetti quando si progetta un GA perché ne determinano il successo del processo evolutivo. Infatti, se in un GA ogni individuo è una sequenza di 100 caratteri, con una fitness lineare, il vantaggio di un individuo con 85 caratteri corretti è relativamente paragonabile a quello di un individuo con 80 caratteri corretti, pertanto il fenomeno evolutivo rallenta notevolmente. In questo caso, è quindi opportuno utilizzare una funzione di fitness esponenziale. Invece nel caso in cui ogni individuo sia una sequenza di 10 caratteri, una fitness esponenziale spingerebbe la riproduzione dei soli individui migliori all'interno di ogni generazione, perdendo così i possibili frammenti di soluzione presenti negli altri individui e riducendo la variabilità generale della popolazione.

Gli esempi precedenti riportano un calcolo della fitness piuttosto semplice, esistono tuttavia funzioni di fitness molto più complesse ed interessanti. Ad esempio, in un algoritmo genetico dove lo scopo evolutivo degli individui è quello di avvicinarsi ad un determinato punto dello spazio, la fitness verrà calcolata sottraendo a 0 la distanza dell'individuo dal target, così che un individuo distante 10 abbia una fitness pari a -10 e uno distante 50 abbia una fitness pari a -50 (favorendo sempre l'individuo con la fitness più alta).

In un secondo esempio, molto più complesso, possiamo immaginare un algoritmo genetico che evolve l'intelligenza artificiale (una rete neurale artificiale o ANN) di una macchina a guida autonoma che deve percorrere un percorso da un punto A ad un punto B. In questo caso la funzione di fitness dovrà tenere conto di molteplici aspetti per valutare in maniera appropriata l'efficacia di ogni rete neurale testata. Infatti un minore tempo impiegato a raggiungere il punto B causerebbe un aumento del valore di fitness, invece gli urti contro ostacoli, un alto consumo di energia e un tragitto poco omogeneo (scatti e frenate improvvise) ne causerebbero una diminuzione.

### 5.3.2 La scelta dei “genitori”

La funzione di fitness non fa altro che valutare quanto un individuo risulti una soluzione ottimale al problema in questione, ma di fatto la scelta dei genitori da parte dell'algoritmo può non essere necessariamente dettata solo dal valore di fitness, ma anche da un certo livello di casualità. Pertanto individuiamo due tecniche principali per la scelta dei genitori che si basano sul valore di fitness generale della popolazione e di ogni singolo individuo.

- **Metodo elitario:** in una popolazione vengono fatti riprodurre solo gli individui migliori. Nel metodo elitario assoluto viene fatto riprodurre solo l'individuo (nel caso di riproduzione asessuata) o i due individui (nel caso di riproduzione sessuata) col valore di fitness più alto in assoluto. Questa tecnica però contrasta con il principio di variabilità in quanto dopo una sola generazione tutti gli individui avranno un patrimonio genetico molto simile. Tale problema può essere evitato utilizzando il metodo elitario ponderato, tramite il quale viene fatta riprodurre solo una percentuale ponderata della popolazione con il valore di fitness maggiore. In altre parole se avessimo una popolazione di 100 individui che vogliamo far riprodurre attraverso il metodo elitario ponderato con una percentuale del 30%, allora solo i 30 individui con il valore di fitness più alto si potranno riprodurre.
- **Metodo probabilistico:** tanto maggiore è la fitness di un individuo, tanto maggiori sono le probabilità che questo possa riprodursi. Questo metodo è quello che mima di più l'evoluzione così come avviene in natura; Darwin, infatti, non afferma che solo gli individui più adatti possono riprodursi, ma solo che essi ne hanno una maggior probabilità.

Consideriamo il seguente esempio: immaginiamo di avere 5 individui che dobbiamo far riprodurre con il metodo di selezione probabilistica.

Elemento	A	B	D	E	F
Fitness	3	4	0,5	1,5	1

La prima cosa che occorre fare è normalizzare il punteggio di fitness di ogni elemento, ovvero trasformare il valore di fitness in uno compreso tra 0 e 1, in modo che la somma dei valori normalizzati di tutti gli individui della popolazione sia pari ad 1. Di conseguenza sarà poi possibile indicare il valore di fitness di ogni individuo come percentuale di fitness totale della popolazione.

Elemento	Fitness	Fitness normalizzata	Espresso come percentuale
A	3	0,3	30%
B	4	0,4	40%
C	0,5	0,05	5%
D	1,5	0,15	15%
E	1	0,1	10%

Applicando il concetto che gli individui con una maggiore fitness hanno una maggior probabilità di originare una prole, possiamo individuare la percentuale di fitness normalizzata come la probabilità di ogni individuo di riprodursi al momento della creazione della generazione successiva.

# 6. Visualization of Evolution

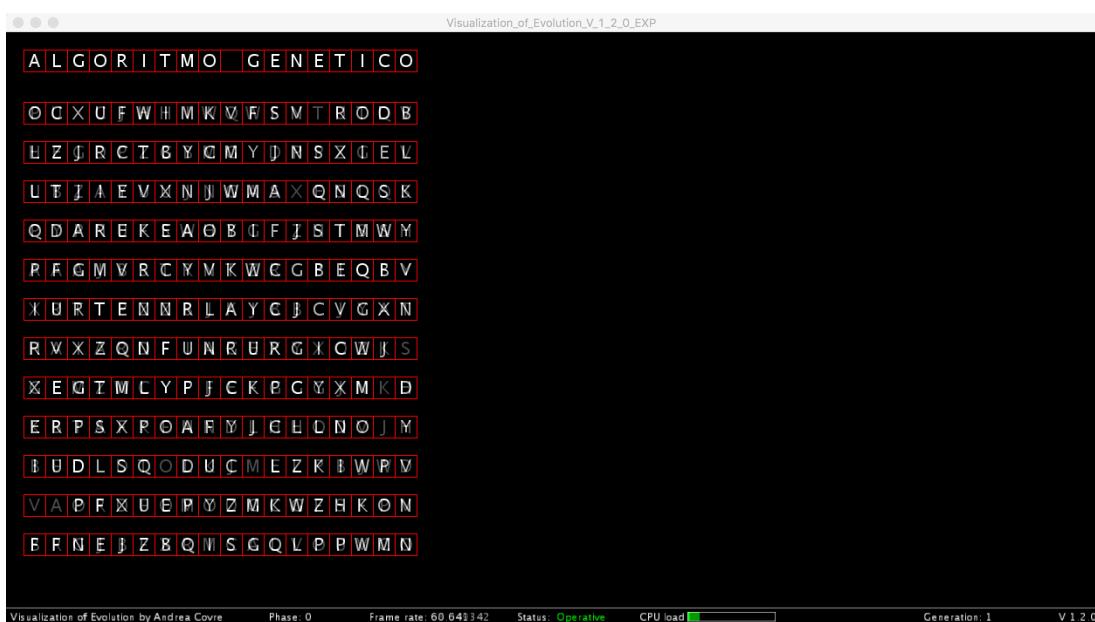
In questa sezione discuteremo le basi del funzionamento di un qualsiasi algoritmo genetico attraverso il programma, da me scritto, *Visualization of Evolution*.

*Visualization of Evolution* ha lo scopo di far evolvere una popolazione di 12 individui verso un target preimpostato, che nel nostro caso sarà la sequenza “ALGORITMO GENETICO”.

## 6.1 Inizializzazione della popolazione

Il primo passo che un algoritmo genetico deve compiere quando viene avviato è l'inizializzazione della popolazione, ovvero la generazione casuale degli individui.

Il numero degli individui generati corrisponderà al numero impostato in precedenza. Ogni individuo avrà un proprio DNA costituito da valori generati casualmente all'interno di intervallo o insieme predefinito.



Nel nostro caso il numero di individui da generare corrisponde a 12 e ognuno di questi avrà 18 geni (numero di caratteri che compongono il target). Solitamente si utilizza una popolazione sull'ordine delle centinaia, ma in questo caso ho scelto di utilizzare una popolazione piuttosto piccola in quanto lo scopo del programma è principalmente esplicativo. Il valore iniziale dei geni sarà selezionato casualmente dall'insieme composto dai numeri compresi tra 65 e 90 (estremi inclusi) più il numero 32. I numeri compresi tra 65 e 90 corrispondono, in codifica ASCII, alle lettere maiuscole dalla A alla Z, e il numero 32 corrisponde allo spazio. Questi valori numerici, infatti, rappresentano il genotipo, mentre le lettere vere e proprie corrispondono al fenotipo.

<b>Genotip</b>	32	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90
<b>Fenotipo</b>	“	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Nel programma, la popolazione viene generata dal seguente blocco di codice:

```
void createPopulation() {
    for (int i = 0; i < popSize; i++) {
        for (int j = 0; j < targetLength; j++) {
            population[i][j] = charPool[ floor( random(0, charPool.length) ) ];
        }
    }
}
```

Dove `popSize` corrisponde alla grandezza della popolazione desiderata, `targetLength` alla lunghezza del target (e quindi degli individui) e `charPool` ad un array contenente tutti i possibili genotipi.

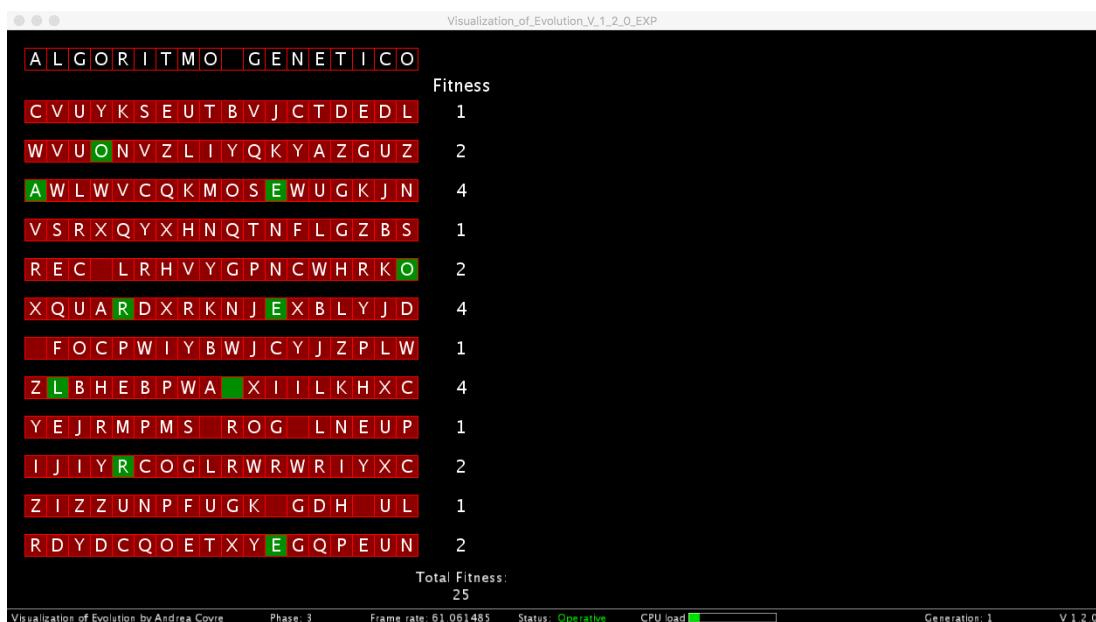
## 6.2 Calcolo della fitness

Il secondo passo corrisponde al calcolo della fitness di ogni individuo secondo la funzione che si è scelta di utilizzare.

Nel caso di questo programma, ho scelto di utilizzare una funzione esponenziale in quanto la popolazione è composta da soli 12 individui.

$$\text{fitness} = 2^{(\text{numero caratteri corretti})}$$

*Visualization of Evolution* è anche dotato di una funzione che permette di evidenziare in ogni individuo quali geni siano corretti e quali no.



Il calcolo della fitness avviene grazie al seguente blocco di codice:

```
void calcFitness() {
    for (int i = 0; i < popSize; i++) {
        int correctGeneN = 0;
        fitnessValue[i] = 0; //resetting fitness value before calculating it again
        for (int j = 0; j < targetLength; j++) {
            if (population[i][j] == target[j]) {
                rightGene [i][j] = 1;
                correctGeneN += 1;
            } else {
                rightGene [i][j] = 0;
            }
        }
        fitnessValue[i] = pow(2, correctGeneN); //exponential fitness function
    }
}
```

## 6.3 Calcolo delle probabilità e selezione

Successivamente si calcolano le probabilità che ogni individuo avrà di riprodursi.

Per questo programma ho preferito scegliere i genitori tramite il metodo probabilistico in quanto esso rappresenta in maniera relativamente accurata ciò che accade in natura.

Il calcolo delle probabilità avviene all'interno del seguente blocco di codice:

```
void calcProbability() {

    matingPool.clear(); //clear out the mating pool "bucket"

    for (int i = 0; i<popSize; i++) {
        probability[i] = fitnessValue[i]/totalFitness * 100;
        text(round(probability[i]) + "%", cellSize*targetLength +150, 80+45*i );

        //add individuals indexes n times based on their fitness
        for (int j = 0; j < fitnessValue[i]; j++) {
            matingPool.add(i);
        }
    }
}
```

Possiamo notare come all'interno di questo blocco di codice vi sia presente l'arrayList `matingPool`, che apparentemente non mostra nessuna utilità nel calcolo delle probabilità. Di fatto, accade il contrario. Tutta la parte di codice dove sembra che vi siano calcolate le probabilità, in realtà ha lo solo scopo di effettuare tale calcolo affinché esse vengano elencate all'interno dell'interfaccia grafica come è mostrato nell'immagine successiva.

Infatti i risultati prodotti da questo calcolo non vengono utilizzati per la scelta dei genitori della generazione successiva.

`matingPool` non è altro che un `arrayList` (array a lunghezza variabile) dove viene inserito l'indice di ogni individuo, che lo identifica all'interno dell'array (`population[][]`) rappresentante la popolazione attuale, tante volte quanto è il suo corrispondente valore di fitness. In altre parole è come se `matingPool` fosse un secchio dove inseriamo un numero di copie di un specifico individuo pari al suo valore della fitness.

ALGORITMO GENETICO		
	Fitness	Chances
CVUYKSEUTBVJCTDEDL	1	4%
WVUONVZLIYQKYAZGUZ	2	8%
AWLWVCQKMOSEWUGKJN	4	16%
VSRXQYXHNQTNFLGZBS	1	4%
REC L RH V Y G P N C W H R K O	2	8%
XQUARDXRKNJEXBLYJD	4	16%
FOCPWIYBWJCYJZPLW	1	4%
ZLBHEBPWA X I I L K H X C	4	16%
YEJRMPMS ROG LNEUP	1	4%
IJIYRCOGLRWRWRIYXC	2	8%
ZIZZUNPFUCK GDH UL	1	4%
RDYDCQOETXYEGQPEUN	2	8%
Total Fitness: 25		

Visualization of Evolution by Andrea Covre    Phase: 4    Frame rate: 61.07606    Status: Operative    CPU load:     Generation: 1    V 1.2.0

I genitori di un individuo della generazione successiva, vengono selezionati tramite questo blocco di codice:

```
void getParents() {
    parentA = matingPool.get(floor(random(totalFitness)));
    parentB = matingPool.get(floor(random(totalFitness)));

    //to be sure that a parent doesn't breed with itself...
    while (parentA == parentB) {
        parentB = matingPool.get(floor(random(totalFitness)));
    }
}
```

Questo blocco di codice non fa altro che prelevare due individui da questo “secchio” immaginario (individuato come `matingPool`), e, nel caso in cui il secondo individuo sia una copia del primo, ripesca un altro individuo finché i due genitori pescati non siano copie dello stesso individuo.

ALGORITMO GENETICO		
	Fitness	Chances
VVOTLPLXPUPUTUFLGWH	1	4%
OQDKPXQXYJYY XCCZ	2	7%
YQSWMNAGR PQLJNETND	1	4%
XZ CDS E FW Z Y C K V U V Y U	1	4%
QLXLOIYM E FN X ER Z H NA	8	30%
WLJDCGP GHE V Q U I G R F	2	7%
KBGCUNZ J NY COOMSSXG	2	7%
ITYKZTRM E M UPN X MRUH	4	15%
NACUJFYYR F NH DGGHBR	1	4%
T F P Z SCY C FM NS I Q K	2	7%
UXCTFPLVYAHKT Y OIWI	2	7%
BNSYUFVITHHDLV BHVT	1	4%
Total Fitness: 27		

Visualization of Evolution by Andrea Covre    Phase: 6    Frame rate: 66.94692    Status: Operative    CPU load:     Generation: 1    V 1.2.0

Per semplificare il concetto riprendiamo l'esempio fatto nella sezione 5.3.2.

Elemento	Fitness	Fitness normalizzata	Espresso come percentuale	Numero di volte inserito nel matingPool
A	3	0,3	30%	30
B	4	0,4	40%	40
C	0,5	0,05	5%	5
D	1,5	0,15	15%	15
E	1	0,1	10%	10



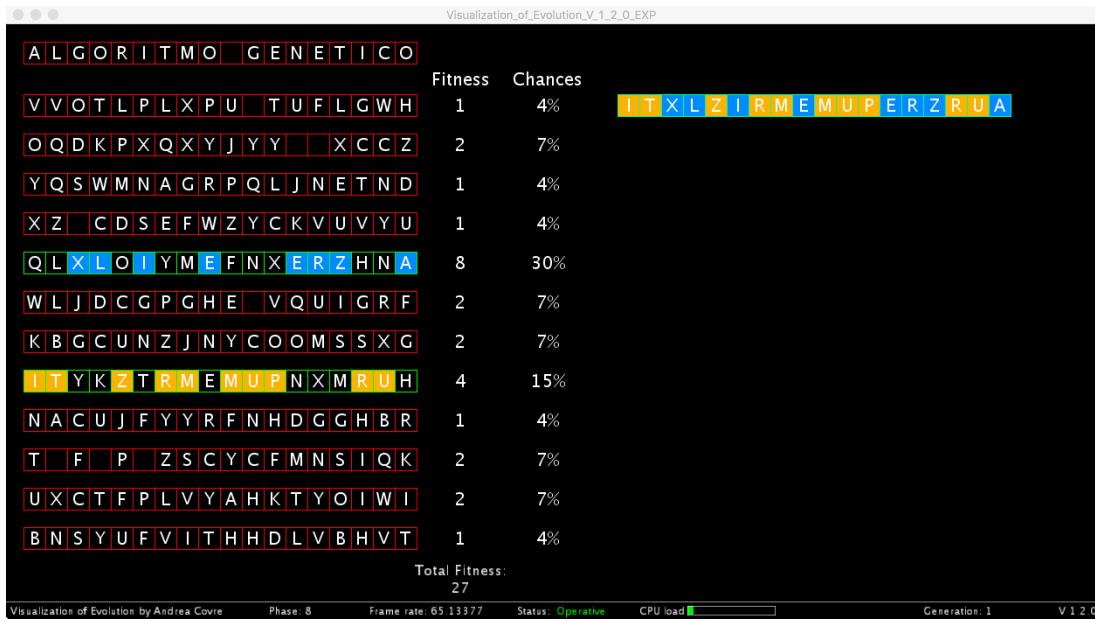
In questo caso, nel secchio, sono presenti 30 copie di A, 40 di B, 5 di C, 15 di D e 10 di E. Pertanto, nel momento in cui si estrae un elemento a caso, si hanno maggiori probabilità di pescare l'elemento B, che di fatto ha il valore di fitness maggiore.

Vale la pena precisare che la funzione `matingPool.get(floor(random(totalFitness)))` simula l'atto di prelevare un elemento dal secchio. Infatti `random(totalFitness)` genera un numero casuale e decimale compreso tra 0 e il valore di fitness totale della popolazione. A questo valore viene poi applicata la funzione `floor()` che arrotonda il numero per difetto, così da garantire uniformità alle probabilità di selezione di ogni elemento (specialmente nelle prime generazioni) e un numero valido (intero) che indichi la posizione dell'individuo selezionato all'interno dell'array della popolazione. Difatti, se venisse utilizzato il metodo di arrotondamento classico (`round()`), gli estremi avrebbero la metà delle probabilità di essere selezionati rispetto a tutti gli altri numeri. Per esempio gli unici valori che porterebbero alla selezione dello 0 sono quelli compresi tra 0 e 0,49, mentre quelli che porterebbero alla selezione di 1, ammesso che la fitness totale sia maggiore o uguale a 2, sono quelli compresi tra 0,5 e 1,49, ovvero un intervallo di larghezza doppia rispetto a quello per la selezione dello 0. Utilizzando l'arrotondamento per difetto, ci si assicura che un dato numero viene selezionato ogni volta che viene generato un numero casuale compreso tra quel dato numero e il suo successivo numero intero (escluso).

## 6.4 Riproduzione

Una volta selezionati gli individui sarà necessario farli riprodurre in maniera asessuata o sessuata. Come già discusso in precedenza nella sezione 5.1, la riproduzione asessuata consiste solamente nel generare un clone dell'individuo selezionato, mentre quella sessuata richiede la ricombinazione del patrimonio genetico.

In *Visualization of Evolution* ho optato per la riproduzione sessuata, dove la ricombinazione avviene tramite la frammentazione, in quanto più complessa ed interessante. Inoltre questa tecnica offre una maggiore variabilità degli individui e garantisce che ogni informazione genetica abbia le stesse probabilità di essere trasmessa alla generazione successiva.



La frammentazione e la creazione dei nuovi individui viene effettuata dal blocco di codice che segue:

```
void makeNewChild() {
    for (int i = 0; i < targetLength; i++) {
        if (floor(random(0, 2)) == 0) {
            newChildMap[i] = 'A';
            newGeneration[childIndex-1][i] = population[parentA][i];
        } else {
            newChildMap[i] = 'B';
            newGeneration[childIndex-1][i] = population[parentB][i];
        }
    }
}
```

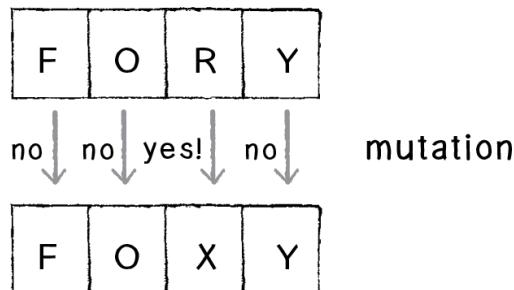
Inizialmente avevo optato per il metodo di crossing-over paritario, ma, dal momento che con tale metodo i geni che si trovano più vicini agli estremi hanno una maggiore probabilità di persistere nelle generazioni successive, accadeva che il processo evolutivo si bloccava a causa della persistenza di informazioni genetiche non utili al raggiungimento della soluzione finale.

Ad ogni modo, il crossing-over paritario, può essere eseguito tramite questo codice:

```
void makeNewChild() {
    //randomly selecting a crossover point, then take half from A and half from B
    crossOverPoint = floor(random(1, targetLength-1));
    for (int i = 0; i < targetLength; i++) {
        if (i < crossOverPoint) {
            newGeneration[childIndex-1][i] = population[parentA][i];
        } else {
            newGeneration[childIndex-1][i] = population[parentB][i];
        }
    }
}
```

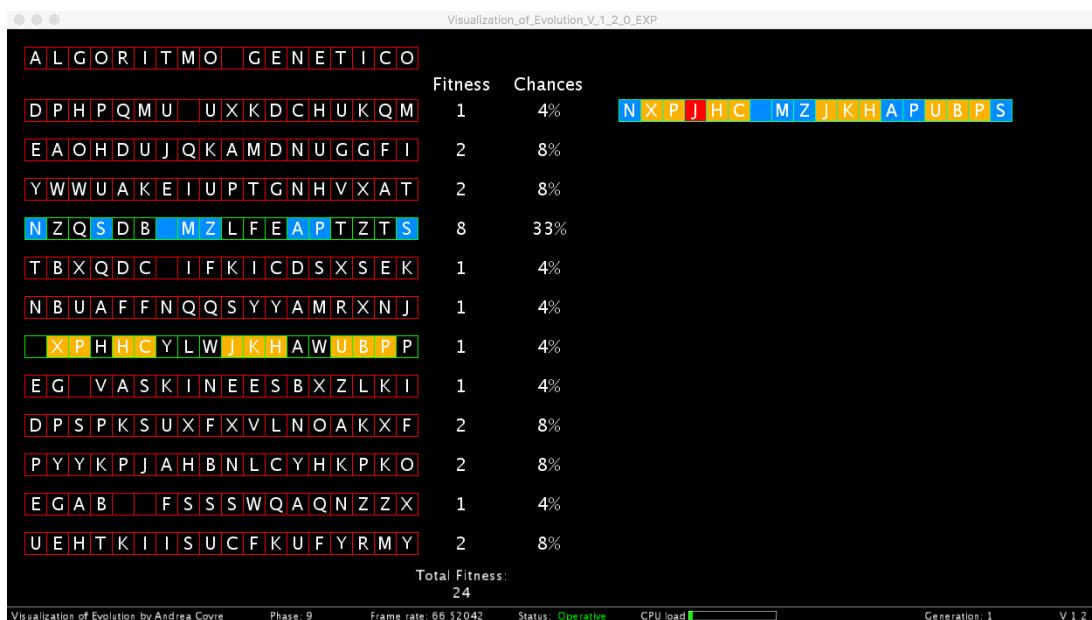
## 6.5 Mutazione

Dopo aver ricombinato le informazioni genetiche bisogna assicurare un certo livello di variabilità, e questo viene fatto attraverso delle mutazioni spontanee (casuali).



In *Visualization of Evolution* il tasso di mutazione è pari al 5%, ovvero ogni gene dopo la ricombinazione ha il 5% di probabilità di mutare e quindi trasformarsi in una lettera diversa oppure in uno spazio. Solitamente si utilizzano tassi di mutazione più bassi (intorno al 2%), ma dato che la popolazione è piuttosto piccola, il numero di momenti in cui una mutazione può verificarsi, eventualmente introducendo nuovi frammenti di soluzione, sono solo 216 (prodotto tra il numero di geni di ogni individuo e il numero di individui nella popolazione). Mentre in una popolazione di 100 individui si possono verificare mutazioni fino a 1200 volte.

Nel programma le mutazioni vengono evidenziate in rosso.



In questo caso il quarto elemento del primo individuo generato, dopo il processo di ricombinazione, era una S (come si può osservare dal genitore che gli ha donato l'informazione). Tuttavia tale informazione ha subito una mutazione, perciò l'informazione si è tramutata in una J.

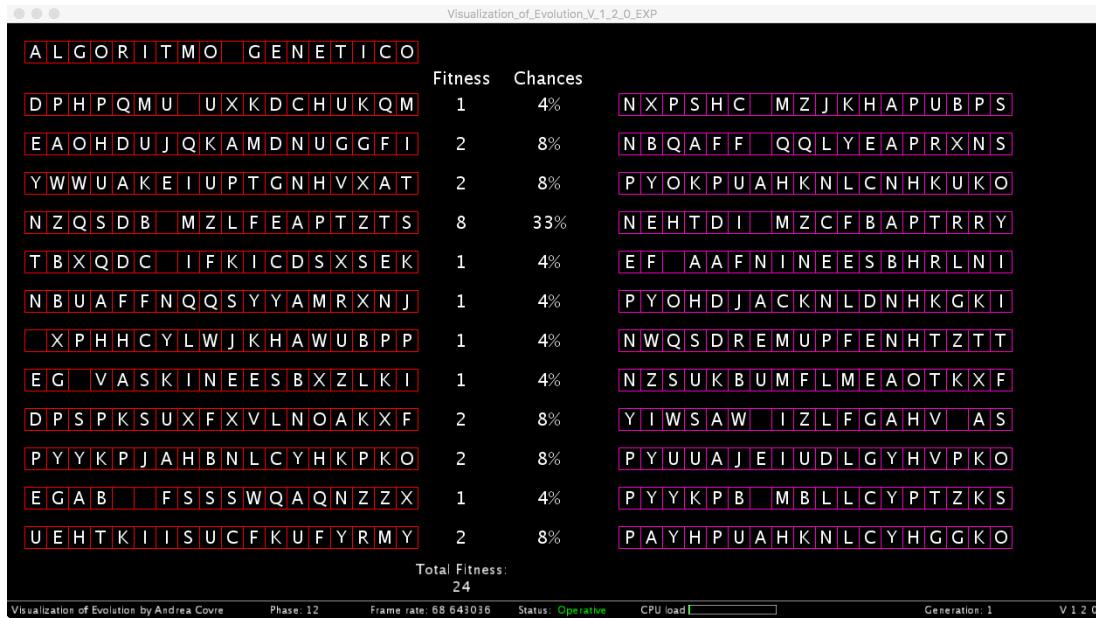
Le mutazioni hanno luogo tramite il seguente blocco di codice:

```

void applyMutation () {
    for (int i = 0; i<targetLength; i++) {
        if (random(0, 100) <= mutationRate) {
            newGeneration[childIndex-1][i] = charPool[floor(random(0, charPool.length))];
        }
    }
}
  
```

## 6.6 Iterazione

Dopo aver creato il primo individuo della nuova generazione è necessario creare tutti gli altri individui rimanenti. Per eseguire tale processo è necessario ripetere per ogni altro individuo da creare la selezione dei genitori, la riproduzione, e infine applicare le eventuali mutazioni.



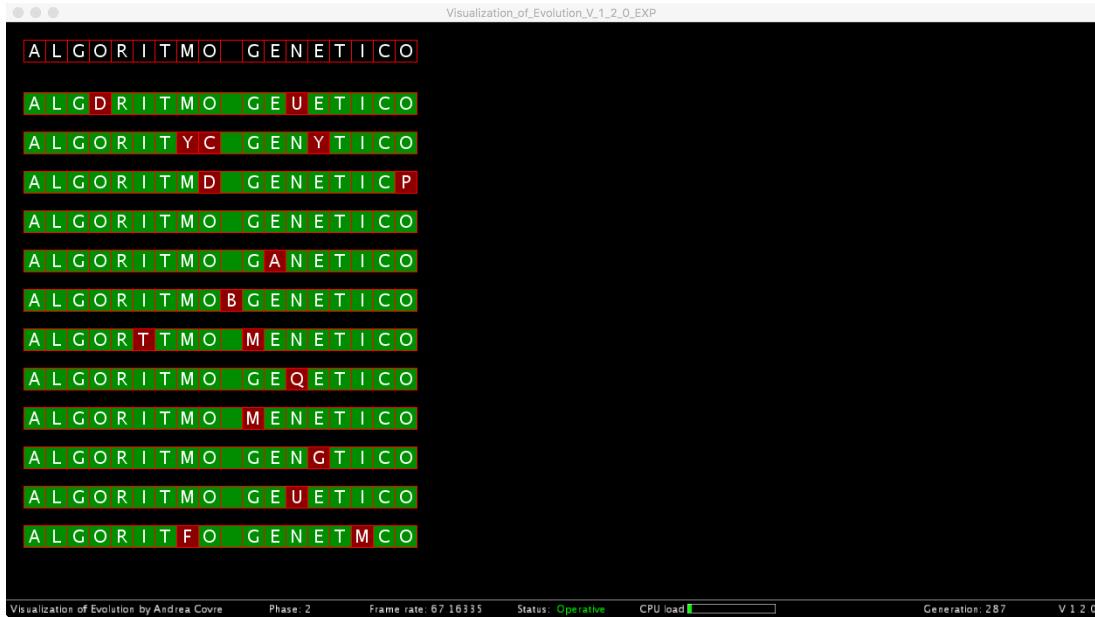
Dopo aver creato la nuova generazione occorre eliminare (uccidere o far morire in termini biologici) quella vecchia, così che quella nuova possa generare a sua volta un'altra popolazione.

Generazione dopo generazione si potrà osservare un generale aumento della fitness, ciò vuol dire che la popolazione sta evolvendo verso la soluzione del problema.



Dopo solo 206 generazioni ogni individuo ha in media 15 caratteri corretti su 18. In questo particolare caso possiamo osservare che l'informazione L al secondo posto della sequenza non è ancora stata introdotta attraverso una mutazione.

Alla 287<sup>a</sup> generazione abbiamo finalmente ottenuto la nostra soluzione; infatti il quarto individuo elencato possiede tutti i frammenti di soluzione necessari e pertanto ha raggiunto il valore di fitness massimo in assoluto, ovvero 262.144 ( $2^{18}$ ).



Gli altri individui sono prossimi all'ottimizzazione dato che la maggior parte dei geni non corretti sono semplicemente causati dall'alto tasso di mutazione.

## 7. Altri esempi di GAs

In questa sezione verranno presentati in breve altri programmi che funzionano con un algoritmo genetico.

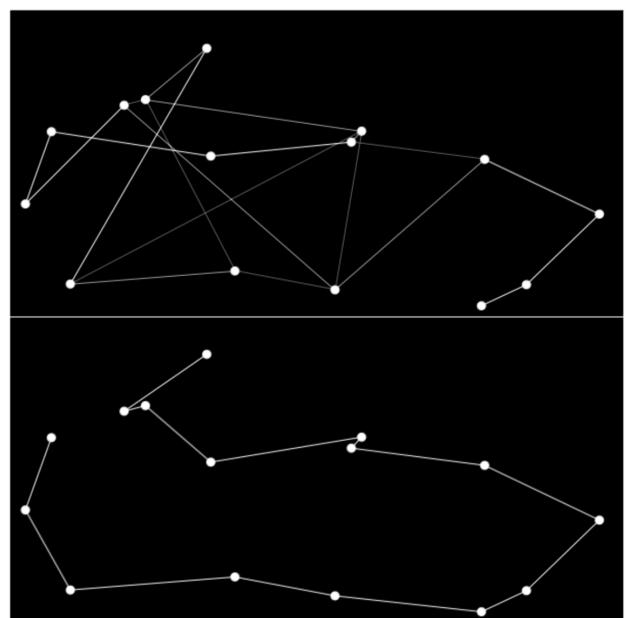
### 7.1 The Traveling Salesman Problem

Questo programma ha lo scopo di risolvere il famoso *problema del commesso viaggiatore*. Tale problema consiste in un commesso che deve passare per esattamente  $n$  città distinte percorrendo la strada più breve possibile.

In questo esempio l'algoritmo cerca di trovare il percorso più breve per passare per tutti i 15 punti.

Nell'immagine superiore si può osservare l'algoritmo che testa gli individui, mentre in quella inferiore si può osservare il miglior percorso trovato dopo 500 generazioni.

Ricordiamo che con 15 città esistono  $15!$  possibili permutazioni semplici, ovvero circa  $1,31 \cdot 10^{12}$  percorsi che un algoritmo a forza bruta dovrebbe calcolare e verificare.

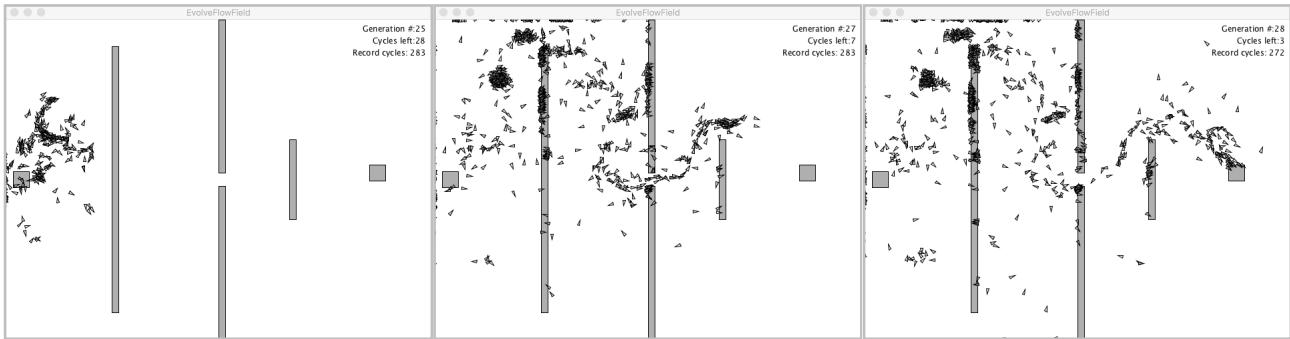


## 7.2 EvolveFlowField

In questo programma ogni individuo corrisponde ad un campo vettoriale che dirige il flusso del proprio oggetto, un triangolino, nello spazio.

Lo scopo di questo GA è quello di sviluppare un campo vettoriale in grado di far superare al proprio oggetto tutti gli ostacoli, fino a raggiungere il target rappresentato dal quadrato a destra.

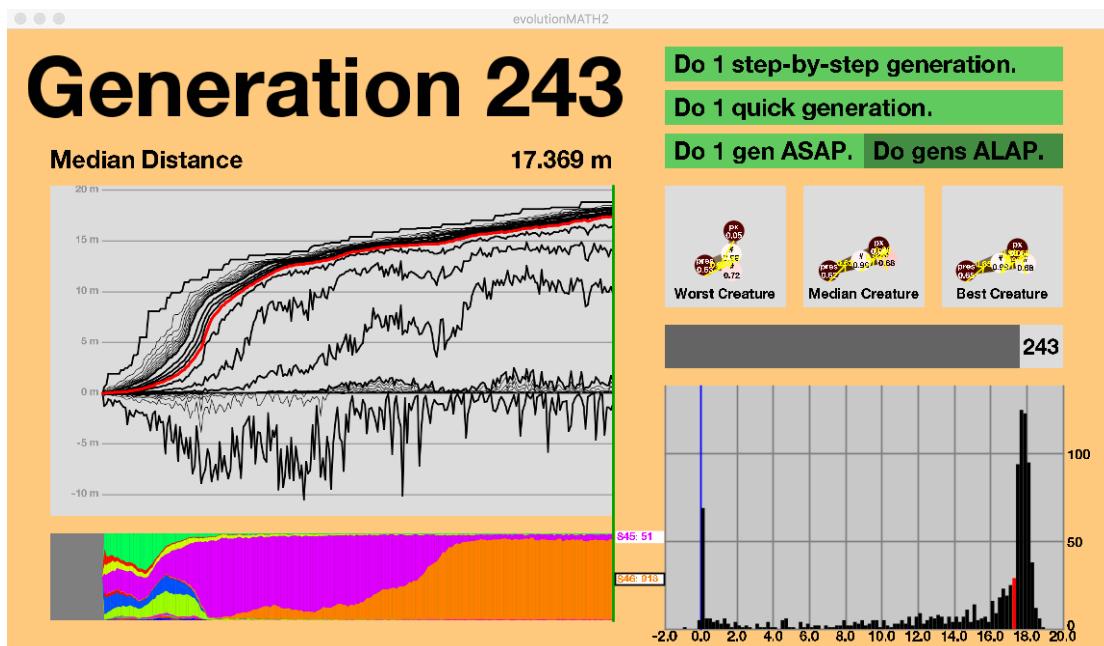
È bene notare come gli individui siano riusciti ad evolvere fino a far passare il proprio oggetto attraverso un piccolissima apertura.



Nelle immagini sopra riportate possiamo osservare una sequenza dei movimenti degli oggetti nei rispettivi campi vettoriali.

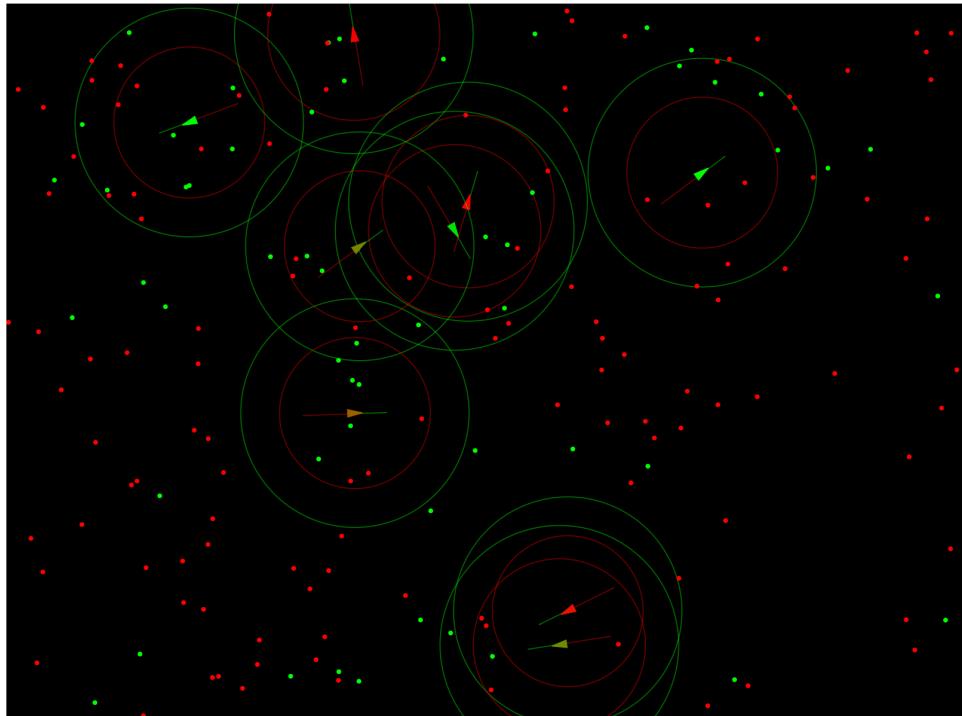
## 7.3 evolutionMATH2

In questo complesso ed affascinante programma (realizzato dall'utente di YouTube "carykh"), gli individui sono delle vere e proprie creature virtuali, dotate di muscoli e giunzioni. Ogni muscolo ha una propria forza e tempo di contrazione, mentre le giunzioni (o nodi) sono dotati di massa e coefficiente d'attrito. Lo scopo del programma è quello di sviluppare la creatura che riesce a percorre la maggior distanza in 15 secondi. Un particolare molto interessante di questo programma è il fatto che viene mostrato l'andamento delle varie generazioni e delle varie specie. Infatti la popolazione è composta da varie specie dotate di un numero diverso di muscoli e nodi, e una diversa configurazione fra essi.



## 7.4 Evolve Steering

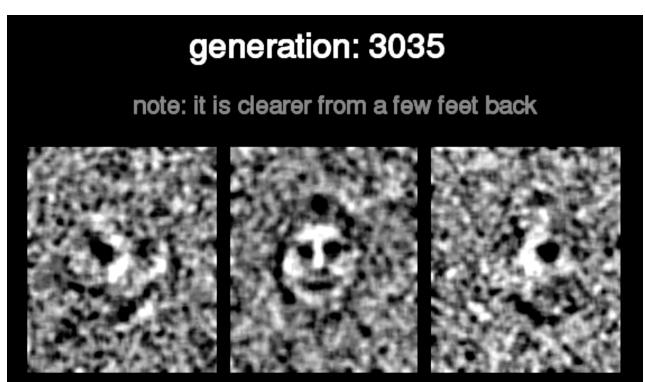
Questo algoritmo genetico funziona tramite la simulazione di un ecosistema. Molto simile a quello presentato nella sezione 1.2.3, questo programma consiste in un ecosistema nel quale gli individui si possono muovere tramite una "Steering Behavior" (una tecnica con la quale si riescono a gestire i movimenti di oggetti in relazione a fattori esterni) con la quale possono ricercare il cibo e stare lontani dal veleno. Se gli individui mangiano il cibo allora sopravviveranno più a lungo e quindi potranno riprodursi in maniera assicurata, se invece non mangiano abbastanza cibo o mangiano troppo veleno allora moriranno senza avere la possibilità di trasmettere il proprio patrimonio genetico.



I parametri che sono sottoposti ad evoluzione sono i parametri che determinano la "Steering Behavior", ovvero, in parole semplici, quanto e come un individuo è attratto dal cibo e "respinto" dal veleno, ed il raggio in cui il cibo o il veleno possono essere "visti" dall'individuo.

## 7.5 Evolving a Human Face

Questo algoritmo è stato creato per puro divertimento da un informatico della Pittsburgh Pattern Recognition, un'azienda che sviluppa anche sistemi per il riconoscimento facciale (che è stata successivamente acquistata da Google Inc.). In questo GA gli individui sono tre immagini in bianco e nero, e la fitness viene calcolata sfruttando un avanzato sistema per il riconoscimento dei volti. Il risultato di questo algoritmo sono tre immagini di una faccia umana (un'immagine frontale e due laterali) sviluppate interamente dal nulla. Questo evidenzia come l'informazione venga effettivamente trasmessa attraverso la selezione, come, per l'appunto, ha affermato Dawkins.



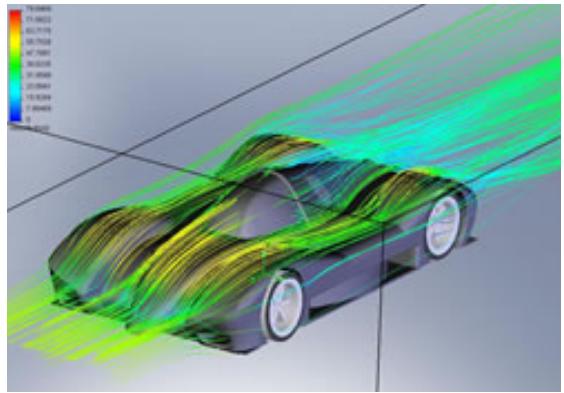
# 8. L'applicazione degli algoritmi genetici nel mondo reale

Gli algoritmi genetici possono essere davvero applicati in moltissimi campi, dall'intelligenza artificiale fino alla progettazione di sistemi antiterroristici. In questa sezione, però, verranno discusse brevemente solo le applicazioni principali e più diffuse, e verranno poi solamente citate altre applicazioni molto interessanti.

## 8.1 Design automobilistico

I GAs vengono utilizzati sia per la progettazione dei materiali compositi che per quella del design aerodinamico delle automobili.

Prima della diffusione dei GAs, gli ingegneri di questo campo dovevano lavorare molti anni alla progettazione di particolari forme aerodinamiche in scala, per poi testarle numerose volte all'interno delle gallerie del vento. Inoltre, una volta selezionata la tipologia di materiale da utilizzare nella costruzione dell'involucro esterno, era necessario produrne diverse varianti per poi testarle e verificare quale variante fosse la migliore.



Con il supporto dei GAs è ora possibile far generare ai computer forme con aerodinamicità eccellente. Inoltre i GAs vengono anche utilizzati per progettare i materiali compositi così da individuare l'ottimale compromesso tra resistenza, peso e costo.



## 8.2 Progettazione di antenne specifiche

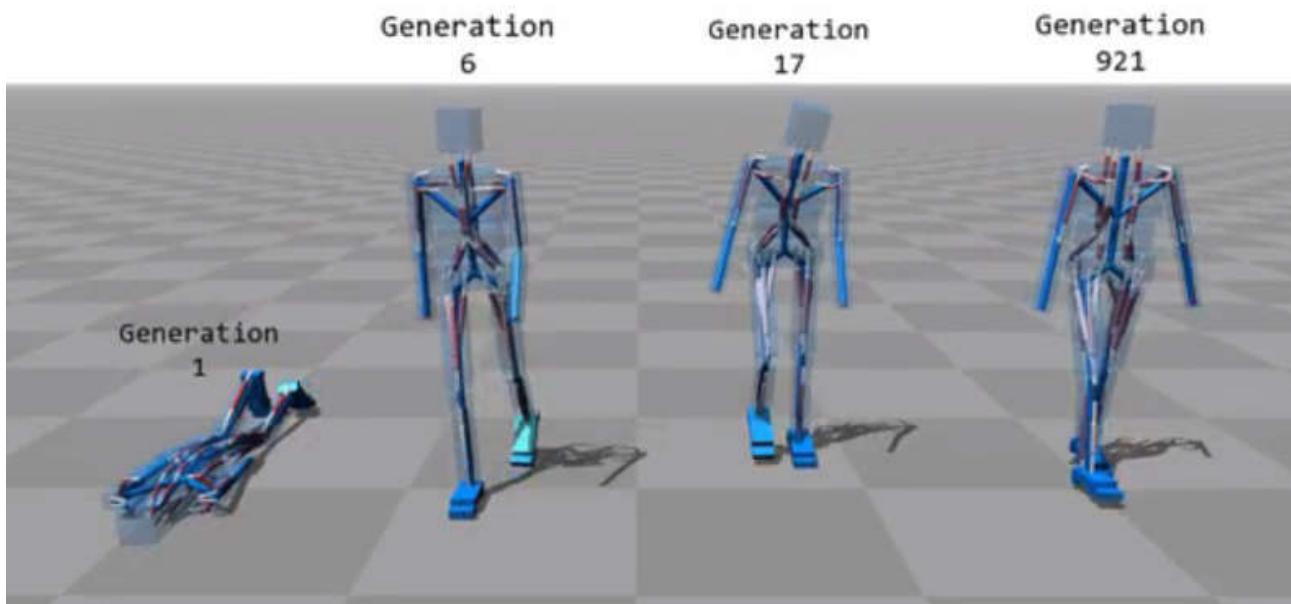
Gli algoritmi genetici permettono di individuare le migliori strutture per antenne di specifico impiego. Infatti, forniti dei parametri base, come ad esempio la frequenza di operatività, lo scopo, la grandezza massima e quella minima, un algoritmo genetico può far evolvere diverse forme fino a trovare quella che più si adatta alla situazione.

La forma complicata dell'antenna riportata qua a fianco è stata generata da un algoritmo genetico che aveva lo scopo di trovare la miglior forma possibile che producesse uno specifico schema di radiazione. Essa è stata sviluppata dalla NASA e oggi è montata sul satellite 2006 NASA STE (che aveva esigenze molto restrittive), rappresentando così il primo oggetto, nello spazio, evoluto artificialmente.

## 8.3 Robotica

Creare un robot dotato di attuatori, motori e sensori è una sfida relativamente semplice, ma quando si tratta di progettare l'algoritmo che riesca a combinare tutti i componenti in modo che il robot si muova in maniera fluida e corretta, allora la sfida si fa molto più complicata.

Quando viene scritto il codice che permette ai roboti di muoversi su un terreno piano e regolare, può accadere che questo non sia molto efficace sui terreni non regolari. Quando si adatta il codice ai terreni non regolari, si può scoprire che il codice non funziona sui piani in pendenza. Un GA può invece permettere lo sviluppo di un modello ottimale che integri in maniera corretta la gestione dei motori con i dati ricevuti dai sensori. Inoltre il GA potrebbe calcolare la fitness di ogni modello anche in base alla sua capacità di guidare i movimenti del robot su diversi terreni.



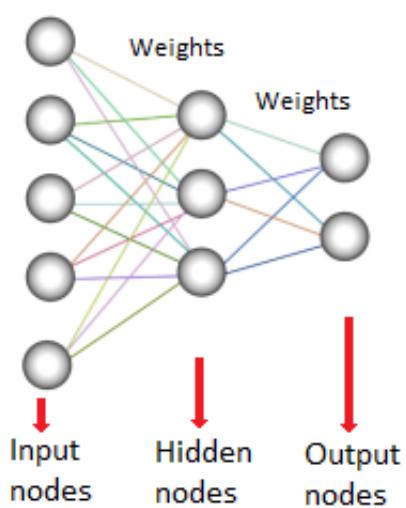
## 8.4 ANN Training

Quando si progetta una rete neurale artificiale si comincia con la creazione della struttura di base e delle interconnessioni fra i vari neuroni.

Ogni "sinapsi" ha un "peso" che modifica l'informazione trasmessa da un neurone all'altro. Affinché la rete neurale artificiale funzioni è necessario assegnare ad ogni connessione il peso corretto, e una tecnica per fare questo, è quella di "allenare" la rete utilizzando un GA che evolva il valore del peso di ogni sinapsi così che l'ANN impari a svolgere un preciso compito.

Questo processo risulta estremamente utile quando le reti sono piuttosto complesse e grandi, come ad esempio negli apparati di machine learning.

Artificial Neural Network



## 8.5 Altre applicazioni

- **Bioinformatica:** individuazione di schemi molecolari ricorrenti e predizione della struttura dell'RNA.
- **Climatologia:** ricerca di modelli predittivi dei cambiamenti climatici.
- **Crittoanalisi:** ricerca di soluzioni in spazi di ricerca di cifrari vastissimi.
- **Logistica:** ottimizzazione dell'organizzazione della merce all'interno dei container e del loro percorso per raggiungere la destinazione.
- **Economia:** ricerca di modelli finanziari e strategie d'investimento efficaci.
- **Telecomunicazioni:** individuazione dei percorsi migliori all'interno delle reti informatiche per garantire il massimo della velocità e un rischio di corruzione dei dati inferiore.
- **Medicina:** supporto al personale medico per la predizione dell'efficacia di un trattamento sui pazienti e dei suoi relativi effetti (oftalmologia e oncologia).
- **Urbanistica:** ottimizzazione delle reti di distribuzione di gas, elettricità ed acqua.







# Conclusioni

Tramite alcuni dei programmi che ho utilizzato per la realizzazione di questo documento, ho potuto effettuare diversi esperimenti, ad esempio utilizzare dei tassi di mutazione prima molto alti e poi molto bassi, oppure aumentare la probabilità di riproduzione degli individui negli algoritmi a simulazione di ecosistema. Grazie a ciò ho potuto comprendere meglio l'interconnessione che esiste tra i parametri che determinano il fenomeno evolutivo artificiale, così da arricchire sia le mie conoscenze su questo argomento, sia le mie capacità di sviluppare in futuro algoritmi genetici sempre più complessi ed efficaci. La creazione degli algoritmi genetici, oltre ad essere stata particolarmente divertente e stimolante, mi ha permesso di sviluppare una conoscenza più "concreta" della teoria dell'evoluzione di Darwin e dei suoi corrispondenti principi.

Durante la ricerca di risorse adeguate per la realizzazione di questo documento ho avuto la possibilità di scoprire e approfondire nuove tecniche e concetti che possono essere applicati nella realizzazione degli algoritmi genetici, ma anche nella realizzazione di altri programmi.

Infine, scrivere questo documento mi ha spinto ad immaginare e maturare altre possibili applicazioni che questa particolare classe di algoritmi potrebbe comportare sia a scopo puramente creativo, sia in applicazioni nel mondo reale.

# Bibliografia

- <https://processing.org>
- <https://processingfoundation.org>
- <https://www.openprocessing.org>
- <http://thecodingtrain.com>
- <https://www.youtube.com/user/shiffman/>
- <https://www.youtube.com/user/carykh/featured>
- <http://natureofcode.com>
- <http://www.karlsims.com>
- <https://www.burakkanber.com/blog/machine-learning-genetic-algorithms-part-1-javascript/>
- <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-034-artificial-intelligence-fall-2010/lecture-videos/lecture-13-learning-genetic-algorithms/>
- <https://it.mathworks.com/help/gads/what-is-the-genetic-algorithm.html>
- <https://www.eecs.harvard.edu/margo/papers/graphics94/paper.html>
- [https://en.wikipedia.org/wiki/Infinite\\_monkey\\_theorem](https://en.wikipedia.org/wiki/Infinite_monkey_theorem)
- [https://it.wikipedia.org/wiki/Teorema\\_della\\_scimmia\\_instancabile](https://it.wikipedia.org/wiki/Teorema_della_scimmia_instancabile)
- [https://en.wikipedia.org/wiki/Richard\\_Dawkins](https://en.wikipedia.org/wiki/Richard_Dawkins)
- <https://www.cse.unr.edu/~sushil/class/gas/notes/GASchemaTheorem2.pdf>
- <http://www.fernandolobo.info/ec1213/lectures/basic-GA-theory.pdf>
- <http://dynamics.org/Altenberg/FILES/LeeSTPT.pdf>
- [https://en.wikipedia.org/wiki/Holland%27s\\_schema\\_theorem](https://en.wikipedia.org/wiki/Holland%27s_schema_theorem)
- [http://ljs.academicdirect.org/A04/44\\_58.htm](http://ljs.academicdirect.org/A04/44_58.htm)
- <https://pdfs.semanticscholar.org/0883/fe0c5ca3c5277e9ff1add18426000915af48.pdf>
- <https://pdfs.semanticscholar.org/52c6/497f6ebc2d8e295ce9849e3e67bdca2084c5.pdf>
- <http://www.cs.le.ac.uk/people/syang/Papers/MISTA09.pdf>
- [https://www.doc.ic.ac.uk/~nd/surprise\\_96/journal/vol1/hmw/article1.html](https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html)
- <http://brainz.org/15-real-world-applications-genetic-algorithms/>
- [https://en.wikipedia.org/wiki/Evolved\\_antenna](https://en.wikipedia.org/wiki/Evolved_antenna)
- <https://defuse.ca/big-number-calculator.htm>