Università degli Studi di Bologna

Approfondimento di Fondamenti di Intelligenza Artificiale

Ottimizzazione con Algoritmi Genetici

Di
Chiara Simoni

Indice

1 Introduzione

Ottimizzare significa partire da un insieme di valori iniziali relativi a variabili coinvolte nell'esperimento e, tramite delle regole definite, cercare di raggiungere un insieme di valori ottimi, che difficilmente coincide con quello iniziale. In alcuni casi, tali valori sono generati da funzioni molto complesse, che non si possono risolvere manualmente. Tuttavia, è importante effettuare una fase di ottimizzazione: un classificatore potrebbe infatti produrre dei risultati poco accurati non tanto, per esempio, a causa del rumore o dell'algoritmo di apprendimento utilizzato, ma proprio a causa dei valori iniziali.

Di conseguenza, ci sono diverse tecniche di ottimizzazione suggerite dalla Ricerca Operativa (OR), che si possono raggruppare in quattro macro-categorie:

- **Constrained Optimization** - Indica il processo con cui si ottimizza una funzione in riferimento a delle variabili e dei vincoli su di esse; la funzione obiettivo è solitamente la funzione dei costi, che deve essere minimizzata;
- **Multimodal Optimization** - Quando si ottimizzano funzioni multimodali, ovvero funzioni che hanno più ottimi locali. Si cercano diverse soluzioni, preferibilmente ottimi locali;
- **Multiobjective Optimization** - Nel caso in cui ci siano più funzioni obiettivo da ottimizzare in maniera simultanea;
- **Combinatorial Optimization** - Processo tramite cui si cerca un massimo o un minimo di una funzione obiettivo in un largo dominio discreto.

Tramite l'osservazione delle specie naturali, si evince la loro capacità di evolvere e adattarsi all'ambiente circostante. Il beneficio dell'analisi dei sistemi naturali esistenti e la loro evoluzione è poter creare dei sistemi artificiali che riproducono tale comportamento. I principi dell'ottimizzazione derivano dalla natura essi stessi: ad esempio, gli algoritmi genetici si basano sulla teoria dell'evoluzione naturale di Charles Darwin e sul concetto che solo il più adatto sopravvive.

Gli algoritmi genetici sono degli algoritmi di ricerca basati sui meccanismi della selezione e delle genetiche naturali: combinano la sopravvivenza del più adatto a scambi di informazioni strutturati e randomici tra gli individui, con l'obiettivo di ottenere un algoritmo di ricerca accompagnato in parte dall'intuizione umana. In ogni generazione si producono nuove creature artificiali usando parti dei più adatti delle vecchie generazioni, aggiungendo occasionalmente nuove misure nel tentativo di ottenere soluzioni migliori.

Gli algoritmi genetici sono stati sviluppati da John Holland e dal suo team di colleghi e studenti dell'Università del Michingan, con il preciso scopo di spiegare i processi adattativi dei sistemi naturali e creare dei software artificiali che riproducono tali caratteristiche e meccanismi. Uno degli aspetti più interessanti della ricerca è la robustezza,

il bilancio tra efficienza ed efficacia necessari per la sopravvivenza negli ambienti più diversi: i segreti dell'adattabilità risiedono appunto negli esempi naturali. Dal punto di vista computazionale, gli algoritmi genetici sono semplici ma potenti, e non sono fondamentalmente limitati dalle assunzioni restrittive negli spazi di ricerca quali continuità, esistenza di derivate o altre questioni.

Le Strategie Evolutive (ES) si sono sviluppate indipendentemente dagli algoritmi generici, anche se circa nello stesso periodo, nel 1960. In comune con gli algoritmi generici, le strategie evolutive usano la rappresentazione in stringhe delle soluzioni e tentano di evolvere la soluzione tramite una serie di passi evoluzionari basati su una funzione di fitness; a differenza dei GA invece, le ES tipicamente non usano una popolazione di soluzioni ma delle sequenze di mutazioni di una soluzione individuale, usando una fitness come guida. Nonostante le origini siano le une indipendenti dalle altre, i due campi si sono sviluppati insieme e la computazione evoluzionaria viene usato come termine generico per più ambiti.

Lo scopo di questo approfondimento è presentare gli algoritmi genetici e analizzare la loro applicazione nel contesto dell'ottimizzazione, fornendo in conclusione un esempio pratico di codice.

2 Terminologia

Sistemi Naturali	Algoritmi Genetici
Cromosoma	Stringa
Gene	Caratteristica
Locus	Posizione della stringa
Genotipo	Struttura
Fenotipo	Set di parametri o soluzione alternativa
Epistasi	Non linearità

Gli algoritmi genetici fanno parte della famiglia degli algoritmi evolutivi, che sono tecniche informative ispirate alla biologia e che si basano sulla metafora di fondo illustrata nella tabella. Come un individuo di una popolazione di organismi deve adattarsi all'ambiente circostante per conseguire gli obiettivi di sopravvivenza e riproduzione, così anche una soluzione deve essere in grado di evolversi per risolvere il problema in questione. L'ambiente in cui la soluzione vive si trova all'interno di una popolazione di possibili altre soluzioni: esse differiscono l'una dall'altra per qualità (costo o merito), valutata tramite la funzione obiettivo, così come nel mondo reale possono presentarsi individui che si adattano meglio o sono più robusti rispetto ad altri. Se la selezione naturale permette agli organismi di adattarsi all'ambiente, sarà possibile fare evolvere in maniera analoga anche le soluzioni di un problema per ottenerne sempre di migliori, fino a raggiungere gli obiettivi di ottimalità.

Le *stringhe* sono dunque paragonabili ai cromosomi dei sistemi biologici e combinandole tra loro si ottengono forme di organismi viventi, quindi particolari soluzioni di un problema. Nei sistemi naturali i cromosomi sono composti dai *geni*, che sono a loro volta formati dagli *alleli*, ovvero le possibili configurazioni che un gene può assumere. Nei sistemi artificiali, le *stringhe* sono composte invece da *caratteristiche* che assumono diversi valori e che possono trovarsi in diverse posizioni nella stringa stessa; la posizione di un valore è determinata dal suo significato.

Una *struttura* invece corrisponde al genotipo di un sistema naturale, che è l'insieme completo delle stringhe. A ciascuna struttura corrisponde un particolare set di parametri, una soluzione alternativa o un punto nello spazio delle soluzioni. In generale nei sistemi artificiali il gene sarà codificato nella forma di un parametro da ottimizzare e la funzione di fitness sarà l'obiettivo da ottimizzare.

3 Ottimizzazione

3.1 Metodi Tradizionali

Ci sono tre macro-categorie in cui sono raggruppati i metodi tradizionali per l'ottimizzazione: metodi basati sull'analisi matematica, enumerativi e casuali.

- **Basati sull'analisi matematica** - Divisi in metodi diretti e indiretti; questi ultimi cercano estremi locali risolvendo insiemi di equazioni non lineari impostando il gradiente della funzione obiettivo uguale a zero. Data una funzione non vincolata, trovare un possibile massimo consiste nel restringere innanzitutto la ricerca a quei punti che hanno pendenza a zero in tutte le direzioni. I metodi diretti invece cercano ottimi locali in direzione del gradiente usando la tecnica dell'hill climbing. Entrambe le soluzioni mancano di robustezza: la ricerca è locale e gli ottimi sono limitati al vicinato di ricerca, che potrebbe non includere vette più alte. Inoltre, una volta che viene raggiungo un minimo locale, diventa necessario ricominciare la ricerca da capo. I metodi basati sull'analisi matematica dipendono dall'esistenza delle derivate. Il mondo reale è caratterizzato da discontinuità e spazi di ricerca rumorosi, per cui metodi che dipendono da tali limitazioni non sono adatti, se non ad un numero limitato di problemi.
- **Enumerativi** - Metodi che ricercano in spazi finiti. L'algoritmo analizza i valori della funzione obiettivo corrispondenti ad ogni punto nello spazio, uno alla volta. Anche se la semplicità di questo algoritmo è allettante, questo schema manca fondamentalmente di efficienza, poichè molti spazi sono troppo ampi per consentire una ricerca di punto alla volta.
- **Casuali** - Gli algoritmi casuali sono diventati sempre più popolari una volta osservate le mancanze degli altri due metodi di ricerca appena elencati. Tuttavia anche in questo caso non si rientra negli obiettivi di efficienza necessari. Una tecnica di ricerca molto popolare è il *Simulated Annealing*, che utilizza processi casuali per guidare la forma di ricerca di stati di energia minimi.

In conclusione, i metodi tradizionali non sono robusti: analizzando problemi sempre più complessi, saranno necessari altri metodi di ricerca. Molti schemi tradizionali lavorano bene in domini ristretti, mentre in domini più ampi sia gli schemi enumerativi sia le ricerche casuali lavorano in maniera inefficiente. Un metodo di lavoro è robusto se lavora in maniera corretta su un ampio spettro di problemi.

3.2 Obiettivi

Tramite il processo di ottimizzazione si cerca di migliorare le performance verso un punto o punti ottimi. Bisogna tuttavia differenziare tra il processo di miglioramento

e la destinazione, o ottimo. Gli algoritmi genetici, al fine di superare gli algoritmi tradizionali in termine di robustezza, devono caratterizzarsi nei seguenti aspetti:

- Gli algoritmi genetici lavorano con la codifica dei parametri, non i parametri stessi: l'insieme di parametri del problema viene dunque codificato in forma di stringhe di lunghezza finita con caratteri di un alfabeto finito.
- La ricerca avviene in una popolazione di punti (cromosomi), non su singolo punto: questo significa che la possibilità di localizzare dei falsi ottimi locali è minima, poichè l'algoritmo considera diverse vette contemporaneamente.
- La funzione obiettivo è usata per ricavare informazioni: gli algoritmi genetici sono ciechi. Per effettuare una ricerca efficace necessitano solo di valori associati alle singole stringhe, in seguito ai quali si valuta la fitness di ogni individuo da riprodurre.
- Al posto di regole non deterministiche si usano regole probabilistiche: la scelta casuale in questo caso è utilizzata come guida per la ricerca verso regiorni di spazio con probabile miglioramento della soluzione.

Queste caratteristiche rendono gli algoritmi genetici più robusti e dunque una scelta migliore rispetto alle tradizionali tecniche di ricerca. Un algoritmo genetico è costruito a partire da un numero di elementi distinti, che possono essere ri-utilizzati e adattati in base alla situazione. I principali temi legati all'uso di tali algoritmi sono la codifica dei cromosomi, la funzione di fitness, la selezione, la ricombinazione e lo schema evolutivo.

3.3 Codifica Dei Cromosomi

Un algoritmo genetico manipola popolazioni di cromosomi, che sono rappresentazioni sotto forma di stringa, di soluzioni di un particolare problema. Un cromosoma è una astrazione dell'omonimo componente di DNA, che può essere concepito come una sequenza di lettere dall'alfabeto A,C,G,T. Una particolare posizione o locus in un cromosoma è definito *gene*, e la lettera che si presenta in un preciso punto del cromosoma (locus), ovvero una delle forme che il gene assume, è invece l'*allele*.
Ogni specifica rappresentazione scelta per un determinato problema è la codifica dell'algoritmo genetico.

Il classico GA utilizza la rappresentazione in stringhe di bit per codificare le soluzioni e in particolare si usano i simboli 0,1 appartenenti all'alfabeto binario, per questo si parla di *codifica binaria*. Per problemi in cui i GA sono applicabili, le soluzioni sono finite ma estremamente lunghe, tanto che la valutazione tramite forza bruta non è computazionalmente realizzabile. L'interpretazione delle stringhe inoltre, dipende interamente dal problema: ad esempio una stringa di bit di lunghezza 20 potrebbe essere usata per rappresentare un singolo valore intero nella notazione binaria in un problema, o rappresentare la presenza o l'assenza di 20 diversi fattori in un altro problema. La

forza degli algoritmi genetici è proprio la possibilità di rappresentare una molteplicità di problemi.
Al contrario, la conseguenza è che la codifica dei cromosomi in sé contenga solo informazioni specifiche del problema e sia necessario consultare la funzione di fitness per comprendere al meglio il significato di un cromosoma all'interno dello specifico problema.

Esistono tuttavia altre tipologie di codifica: *Permutation Encoding*, in cui i cromosomi sono delle stringhe di numeri e rappresentano delle sequenze; *Value Encoding* in cui i cromosomi sono composti da stringhe di valori (interi, reali o char) connessi al problema in esame; *Tree Encoding* in cui i cromosomi sono rappresentati tramite alberi di oggetti quali funzioni o comandi di linguaggi di programmazione.

3.4 Funzione di Fitness

La funzione di fitness rappresenta il calcolo computazionale che valuta la qualità dei cromosomi come soluzioni di un particolare problema. In analogia con la biologia, il cromosoma è definito come genotipo, mentre la soluzione che rappresenta è il fenotipo. La funzione di fitness determina quindi il raggiungimento di criteri e obiettivi come il tempo di completamento, utilizzo delle risorse o minimizzazione dei costi. Tale complessità ricorda l'evoluzione biologica, dove il cromosoma in una molecola di DNA rappresenta un insieme di istruzioni per costruire un organismo. Una serie di complessi processi chimici trasforma dunque una ristretta collezione di embrioni contenenti il DNA in organismi completamente formati, che sono poi valutati in termini di successo, usando come riferimento un range di fattori.

3.5 Selezione

Un algoritmo genetico utilizza la funzione di fitness come discriminatore per la qualità della soluzione rappresentata dai cromosomi appartenenti alla popolazione. La selezione è il processo che usa la fitness per guidare l'evoluzione: i cromosomi più adatti alla sopravvivenza hanno la possibilità di essere selezionati più volte o anche essere ricombinati tra loro.

- Il metodo tradizionale di selezione è la **Roulette Wheel**, o fitness proporzionale: ad ogni cromosoma si associa la probabilità di essere selezionato, proporzionale alla sua relativa fitness, relativa alla somma di tutte le fitness dei cromosomi nella popolazione. La ruota viene girata finchè non sono selezionati due genitori, quindi vengono prodotti i successori e si continua fino a che non viene creata l'intera nuova popolazione.
  Il motivo per cui questo metodo non è sempre efficiente, è che se il valore della fitness per tutti gli individui è molto simile, i genitori saranno scelti con uguale

probabilità. In aggiunta, la selezione tramite Roulette Wheel è molto sensibile alla forma della funzione di fitness e generalmente richiede alcune calibrazioni per funzionare al meglio.

- **Tournament Selection** - Schema alternativo che prevede che prima si selezionino i due cromosomi con probabilità uniforme e poi si scelgano quelli con la fitness più alta. La selezione continua fino al punto in cui la nuova popolazione è creata; gli individui possono essere genitori di molteplici figli o di nessun figlio.
- **Rank Selection** - Gli elementi vengono ordinati in base alla fitness e si assegna a ciascuno di essi un peso inversamente proporzionale all'ordine (rank).
- **Truncation Selection** - Seleziona casualmente gli elementi dalla popolazione, dopo avere eliminato un numero prefissato dei cromosomi meno adatti alla sopravvivenza.

3.6 Ricombinazione

Una volta che due cromosomi sono stati selezionati come genitori per la ricombinazione, si applica l'operatore: tra le soluzioni più comuni si trova il one-point crossover, anche se esistono delle varianti.

- **One-point crossover** - Significa selezionare un valore intero k, ovvero la posizione nella striga che determina due sottoinsiemi della stringa stessa, che sono poi scambiate. Di seguito un esempio usando delle stringhe da 10 bit. Esempio in cui k= 4.

  Parent one: 1 1 1 0 **1 0 0 1 1 0**
  Parent two: **0 0 1 0** 0 1 1 1 0 0

  Child one: 1 1 1 0 0 1 1 1 0 0
  Child two: **0 0 1 0 1 0 0 1 1 0**

- **Multi-point crossover** (o two-point crossover) - Sono degli operatori in cui sequenze di punti di crossover vengono scelti insieme alla lunghezza del cromosoma e i figli generati sono costruiti a partire dai valori degli alleli dei due genitori, scambiandoli ad ogni punto di ricombinazione.

  Parent one: **1 1 1 0 1 0 0 1 1 0**
  Parent two: 0 0 1 0 0 1 1 1 0 0

  Child one: **1** 0 **1 0** 0 **0 0 1** 0 **0**

L'azione del crossover e della riproduzione negli algoritmi genetici determina scambi di informazioni tra i geni, nel tentativo ripetuto, di ottenere performance sempre migliori.

3.7 Mutazione

Il processo indispensabile per garantire che del materiale genetico potenzialmente utile non sia eliminato è la mutazione: la riproduzione e il crossover non sono sufficienti a garantire che questo non succeda. L'indice è determinato da un valore estremamente piccolo negli algoritmi, e in natura probabilmente ancora minore. Dalla letteratura emergono in generale otto tipologie di mutazione:

- *Insert Mutation* - Usata nella codifica a permutazione, Prende due geni a caso, poi sposta il secondo in seguito al primo, e il resto viene accordato in ordine. Questo procedimento preserva la maggior parte dell'ordine e l'adiacenza delle informazioni;

  Before: 0 1 **2** 3 4 **5** 6 7 8 9
  After: 0 1 **2 5** 3 4 6 7 8 9

- *Inversion Mutation* - Usata nella Permutation Encoding; si selezionano due alleli a caso e si invertono dei sottoinsiemi di stringhe al loro interno. Si preserva per la maggior parte l'adiacenza ma non l'ordine delle informazioni;

  Before: 0 1 **2 3 4 5** 6 7 8 9
  After: 0 1 **5 4 3 2** 6 7 8 9

- *Scramble Mutation* - Usata anch'essa nella Permutation Encoding. Si sceglie un sottoinsieme di geni a caso e i loro valori sono mescolati in maniera casuale (i sottoinsiemi non devono essere contigui);

  Before: 0 1 **2 3 4 5** 6 7 8 9
  After: 0 1 **3 4 2 5** 4 6 7 8 9

- *Swap Mutation* - Usata nella Permutation Encoding; si selezionano due alleli a caso e si scambiano. Viene mantenuta la maggior parte dell'adiacenza delle informazioni tranne i collegamenti degli alleli selezionati con i vicini;

  Before: 0 1 **2** 3 4 **5** 6 7 8 9
  After: 0 1 **5** 3 4 **2** 6 7 8 9

- *Flip Mutation* - Usata nella codifica binaria; se un gene è a 0 diventa 1, e viceversa;

Before: **1** 0 1 0 0 0 0 **1** 0 **0**
After: **0** 0 1 0 0 0 0 **0** 0 **1**

- *Interchanging Mutation* - Si scelgono due posizioni in modo casuale e i bit di tali posizioni sono scambiati;

  Before: **1** 0 1 0 0 0 0 1 0 **0**
  After: **0** 0 1 0 0 0 0 0 0 **1**

- *Uniform Mutation* - L'operatore di mutazione cambia il valore del gene con un valore casuale uniforme scelto dall'utente, in un intervallo relativo al gene; usato nella rappresentazione reale;
- *Creep Mutation* - Si seleziona un gene in maniera casuale e il suo valore è scambiato con un altro valore casuale scelto all'interno di un intervallo; usato nella rappresentazione reale.

3.8 Schema Evolutivo

In seguito alla ricombinazione, i cromosomi risultanti sono passati alle generazioni successive. I processi di selezione e ricombinazione sono iterati fino ad ottenere una nuova generazione completa. A quel punto la popolazione successiva diventa una nuova popolazione sorgente per le nuove generazioni.

L'algoritmo genetico continua attraverso le generazioni fino a quando non si raggiungono gli obiettvi stabiliti: questi possono includere un numero prefissato di generazioni o la convergenza a partire da una soluzione con una fitness ottimale, o ancora fino alla generazione di una soluzione che soddisfa pienamente un insieme di vincoli.

Ci sono diversi schemi evolutivi che possono essere utilizzati, in base a quali cromosomi è consentito passare indenni da una generazione alla successiva. Le soluzioni comprendono *Sostituzione Completa* in cui tutti i membri di una popolazione sono generati tramite selezione e ricombinati ad una nuova generazione in cui gli individui meno adatti alla sopravvivenza sono sostituiti da altri una generazione alla volta; *Rimpiazzo con Elitismo* è un'altra possibilità ampiamente utilizzata, per garantire che alcuni individui più robusti siano mantenuti da una popolazione alla successiva. Quest'ultimo metodo permette di preservare gli elementi con fitness maggiore dalla selezione non deterministica che si attua ad ogni iterazione.

La ricombinazione può assumere diverse forme: quelle più comuni sono la ricombinazione *discreta* e *intermedia*: nel primo caso ogni componente del figlio è presa da uno dei genitori in maniera casuale, nel secondo caso invece si ottiene un figlio tramite

combinazione lineare, con un parametro casuale, delle componenti dei genitori.
Esistono due schemi alternativi, che definiscono due classi di strategie evolutive: *(n,m)* e *(n+m)*: nella prima classe da una popolazione di n elementi si producono $m > n$ figli e gli n migliori vanno a formare la generazione successiva; nella seconda classe invece, anche gli n genitori partecipano alla selezione, e dei totali $n+m$ individui solo gli n migliori compongono la generazione seguente, questo schema prende il nome di *sostituzione con elitismo*. In entrambi i casi la selezione è deterministica e funziona per *troncamento*, ovvero scarta gli individui peggiori, e la funzione obiettivo si considera direttamente come parametro dell'ottimizzazione da minimizzare o massimizzare.

Ci sono infine ulteriori opzioni di cui si può tenere conto nell'evoluzione dei cromosomi quali: gruppi separati di popolazioni che interagiscono in maniera non regolare, suddiviaioni di popolazioni in base al sesso maschile o femminile, identificare degli individui maschili "Alpha", procreare partendo da tre genitori, prevedere tre sessi, includere i geni recessivi, e molto altri, che risultano utili solo per funzioni di fitness molto specifiche.

4 Algoritmi genetici applicati a TSP

Si è scelto di riportare come esempio pratico un classico problema di ottimizzazione, il Travelling Salesman Problem. Il TSP è uno dei più conosciuti problemi di ottimizzazione combinatoria, studiato sia in ambito della Ricerca Operativa sia dell'Informatica.

Il problema si formula nella seguente maniera: dato un insieme di n città e le distanze che separano tutte le coppie di elementi, si cerca il percorso che li visita tutti una singola volta e di lunghezza minima. Il problema è NP-Hard, quindi non risolvibile tramite algoritmi in un tempo polinomiale, ma molte euristiche hanno proposto soluzioni quasi ottime: la complessità aumenta in proporzione al numero delle città tenute in considerazione.

L'esempio viene proposto tramite un implementazione rappresentativa e semplificata in Python, riportata di seguito e correlata di dettagli e spiegazioni. In questo approccio, le singole città, rappresentate come coppie di coordinate (x,y), identificano i geni, e vanno a formare gli individui, ovvero i cromosomi, sotto forma di percorsi che soddisfano la formulazione del problema. La qualità dei percorsi ottenuti viene valutata tramite una semplice funzione di fitness, basata sulla lunghezza del percorso. Si mostra dunque come varia il percorso, e soprattutto la distanza totale, dalla prima generazione all'ultima.

In primo luogo, viene creata la classe `City`, in cui è definito un metodo `distance` per calcolare la distanza tra due città, usando il teorema di Pitagora.

```
import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

class City:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, city):
        xDis = abs(self.x - city.x)
        yDis = abs(self.y - city.y)
        distance = np.sqrt((xDis ** 2) + (yDis ** 2))
        return distance
```

Si procede dunque a definire la classe `Fitness`: minore è la distanza tra due città, maggiore sarà la fitness dell'individuo, ovvero del percorso. Il punto di partenza e di arrivo devono coincidere, quindi sarà necessario tenere conto anche del segmento dalla città finale alla città iniziale al termine della somma.

```
class Fitness:
    def __init__(self, route):
        self.route = route
        self.distance = 0
        self.fitness= 0.0

    def routeDistance(self):
        if self.distance ==0:
            pathDistance = 0
            for i in range(0, len(self.route)):
```

```
                fromCity = self.route[i]
                toCity = None
                if i + 1 < len(self.route):
                    toCity = self.route[i + 1]
                else:
                    toCity = self.route[0]
                pathDistance += fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def routeFitness(self):
        if self.fitness == 0:
            self.fitness = 1 / float(self.routeDistance())
return self.fitness
```

Una volta definite le classi di base, si crea un singolo individuo, ovvero un singolo percorso possibile, tramite la funzione `createRoute`; dai singoli percorsi si crea la popolazione iniziale dell'algoritmo di dimensione `popSize`, tramite la funzione `initialPopulation`. Una volta definita la popolazione, si utilizza la funzione `rankRoutes` per ordinarle in base alla fitness.

```
def createRoute(cityList):
    route = random.sample(cityList, len(cityList))
return route

def initialPopulation(popSize, cityList):
    population = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))
return population

def rankRoutes(population):
    fitnessResults = {}
    for i in range(0,len(population)):
        fitnessResults[i] = Fitness(population[i]).routeFitness()
return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)
```

Per procedere con la riproduzione bisogna quindi selezionare i genitori che degli individui della popolazione successiva. Sono state implementate due alternative: la prima tramite *Roulette Wheel* e la seconda tramite *Rank Selection*, in entrambi i casi si applica una selezione con elitismo, in cui gli individui più adatti alla sopravvivenza vengono mantenuti nella generazione successiva.

Per definire delle popolazioni successive sono state usate due funzioni: `selection` e `matingPool`. `Selection` prende in input la lista di percorsi ordinati secondo la fitness generata da `rankRoutes` e il numero di individui da considerare "elitari". Nel caso di Roulette Wheel, nelle righe 3-5 viene costruita la ruota, calcolando il peso per ciascun individuo, che verrà paragonato ad un numero casuale per selezionare la lista di percorsi da inserire nel mating pool. La funzione `matingPool` genera come risultato gli individui selezionati dalla popolazione.

Nel caso invece di Rank Selection, si prendono gli individui ordinati e si assegna un peso inversamente proporzionale al loro Rank, basato sull'indice di ordinamento. Questo significa che l'individuo al primo posto avrà un pesò maggiore rispetto ai successivi

nella lista. Come per la Roulette Wheel, si compara il peso con un numero generato casualmente per selezionare o meno l'elemento in considerazione.

```
def rouletteSelection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])
    df['cum_sum'] = df.Fitness.cumsum()
    df['cum_perc'] = 100*df.cum_sum/df.Fitness.sum()

    for i in range(0, eliteSize):
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        pick = 100*random.random()
        for i in range(0, len(popRanked)):
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
```

```
def rankSelection(popRanked, eliteSize):
    selectionResults = []
    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])

    for i in range(0, eliteSize): #introduce elitism
        selectionResults.append(popRanked[i][0])
    for i in range(0, len(popRanked) - eliteSize):
        df["Rank"] = df.index.values.astype(int)
        df["Weight"] = 1/(df.Rank+1)

        pick = random.random() #select a random weight to compare to the weights
        for i in range(0, len(popRanked)): #and then select the mating pool in Results
            if pick <= df.iat[i,3]:
                selectionResults.append(popRanked[i][0])
                break
    return selectionResults
```

Una volta creato il pool, è possibile attuare il crossover, scegliendo un punto a caso e dividendo la stringa selezionata in due parti per produrre i successori. Nel caso specifico del Travelling Salesman Problem, è necessario applicare il crossover a tutti i percorsi contemporaneamente: viene dunque selezionato un sottoinsieme della prima stringa genitore (riga 13) e si riempiono i rimanenti geni prendendoli dal secondo genitore, mantenendo l'ordine di apparizione, tenendo conto di non generare duplicati con i geni estrapolati dal primo genitore (riga 15). Si crea quindi una popolazione di successori, usando una funzione di elitismo per mantenere i percorsi migliori della popolazione corrente e generando il resto dei figli tramite la funzione `breed`.

```
def breed(parent1, parent2):
    child = []
    childP1 = []
    childP2 = []

    geneA = int(random.random() * len(parent1))
    geneB = int(random.random() * len(parent1))

    startGene = min(geneA, geneB)
    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):
        childP1.append(parent1[i])

```

```
    child = [item for item in parent2 if item not in childP1]
    child[startGene:startGene] = childP1
return child

def breedPopulation(matingpool, eliteSize):
    children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,eliteSize):
        children.append(matingpool[i])

    for i in range(0, length):
        child = breed(pool[i], pool[len(matingpool)-i-1])
        children.append(child)
return children
```

Una volta generato un insieme di successori, bisogna tenere conto di un ulteriore fattore: la mutazione. Questo operatore ci permette di evitare la convergenza locale e al tempo stesso esplorare nuove soluzioni nello spazio. Considerando il contesto del problema, verrà usato l'operatore di *Swap Mutation*, questo significa che due città assumeranno posizioni invertite nei percorsi selezionati. Una volta creata la funzione `mutate`, è necessario applicarla a tutta la popolazione.

```
def mutate(individual, mutationRate):
    for swapped in range(len(individual)):
        if(random.random() < mutationRate):
            swapWith = int(random.random() * len(individual))

            city1 = individual[swapped]
            city2 = individual[swapWith]

            individual[swapped] = city2
            individual[swapWith] = city1
return individual

def mutatePopulation(population, mutationRate):
    mutatedPop = []

    for ind in range(0, len(population)):
        mutatedInd = mutate(population[ind], mutationRate)
        mutatedPop.append(mutatedInd)
return mutatedPop
```

Di seguito la funzione per produrre una nuova generazione: innanzitutto si ordinano i percorsi usando la funzione `rankRoutes`, si selezionano i potenziali genitori e li si inserisce in un pool di riproduzione da cui si estraggono i futuri genitori. Si applicano quindi riproduzione, mutazione e si fornisce la nuova generazione.

```
def nextGeneration(currentGen, eliteSize, mutationRate):
    popRanked = rankRoutes(currentGen)
    selectionResults = selection(popRanked, eliteSize)
    matingpool = matingPool(currentGen, selectionResults)
    children = breedPopulation(matingpool, eliteSize)
    nextGeneration = mutatePopulation(children, mutationRate)
return nextGeneration
```

La popolazione iniziale è creata definendo diversi parametri, tramite cui si definiscono la dimensione della popolazione e delle generazioni da esplorare, oltre ai valori di mu-

tazione e di elitismo. L'algoritmo prende in ingresso una lista di 30 città, espresse in termini di coordinate (x,y), e genera tramite la funzione `printRoute` una rappresentazione grafica dei punti in un sistema di assi.

L'implementazione mostrata prevede come paramentri: `popSize`= 100, `eliteSize`= 20, `mutationRate`= 0.01, `generations`= 300.

```
def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):
    pop = initialPopulation(popSize, population)
    print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))
    initialRouteIndex=rankRoutes(pop)[0][0]
    initialRoute=pop[initialRouteIndex]
    printRoute(initialRoute)

    for i in range(0, generations):
        pop = nextGeneration(pop, eliteSize, mutationRate)

    print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))
    bestRouteIndex = rankRoutes(pop)[0][0]
    bestRoute = pop[bestRouteIndex]
    printRoute(bestRoute)
```

In base al percorso iniziale e finale elaborato, si fornisce quindi un'indicazione della distanza iniziale e finale, mostrati sulla griglia. Si mostra inoltre come diminuisce la distanza totale generazione dopo generazione: dopo un certo numero di iterazioni si raggiunge una convergenza, dopo cui non si ottengono cambiamenti significativi. Si possono inoltre notare alcune generazioni in la distanza finale calcolata sia addirittura peggiore rispetto a quella determinata in precedenza: gli algoritmi genetici, come spiegato nell'introduzione, analizzano più alternative possibili, e per questo sono estremamente flessibili, ma ciò non garantisce che la possibilità presa in esame sia la migliore in assoluto.

La differenza principale nell'utilizzo di Roulette Wheel e Rank Selection, risiede nel fatto che Rank Selection affida la stessa probabilità di essere selezionati a tutti gli individui in esame, mentre nel caso di Roulette Wheel, gli elementi con fitness maggiore hanno più probabilità degli altri. Nel caso di Roulette Wheel, se gli elementi hanno fitness molto diverse, si privilegiano sin dall'inizio chi ha valori di funzione obiettivo più elevati, giungendo ad una convergenza prematura e perdita di diversità. Tramite Rank Selection invece, si evita questo rischio, cercando di dare una possibilità equa a tutti gli individui, e quindi anche potenziali soluzioni migliori.

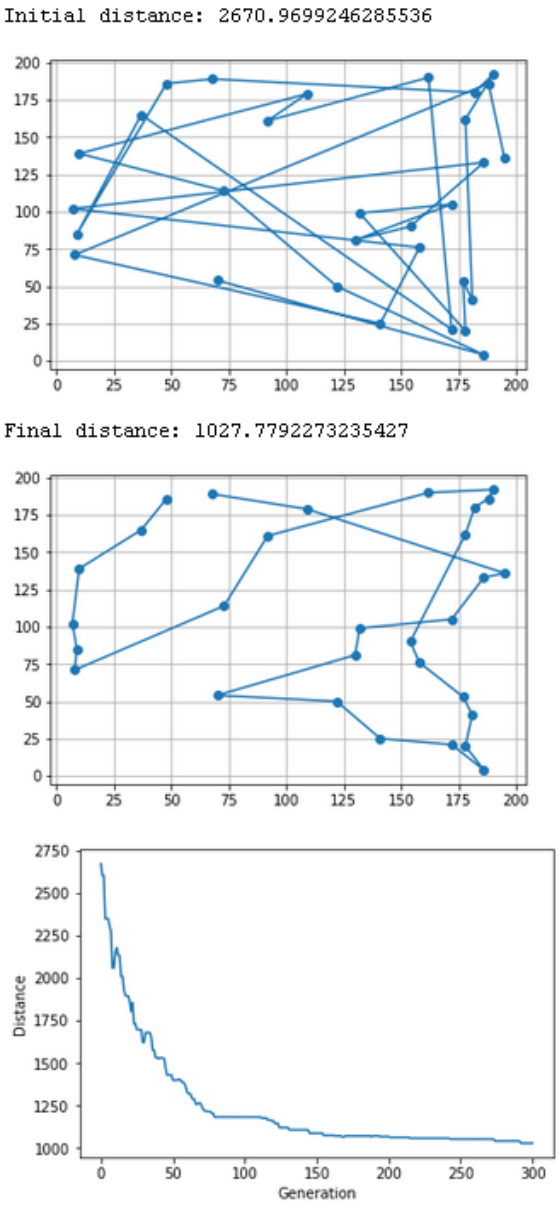

Figure 1: Percorso iniziale e finale ottenuti tramite Roulette Wheel Selection.

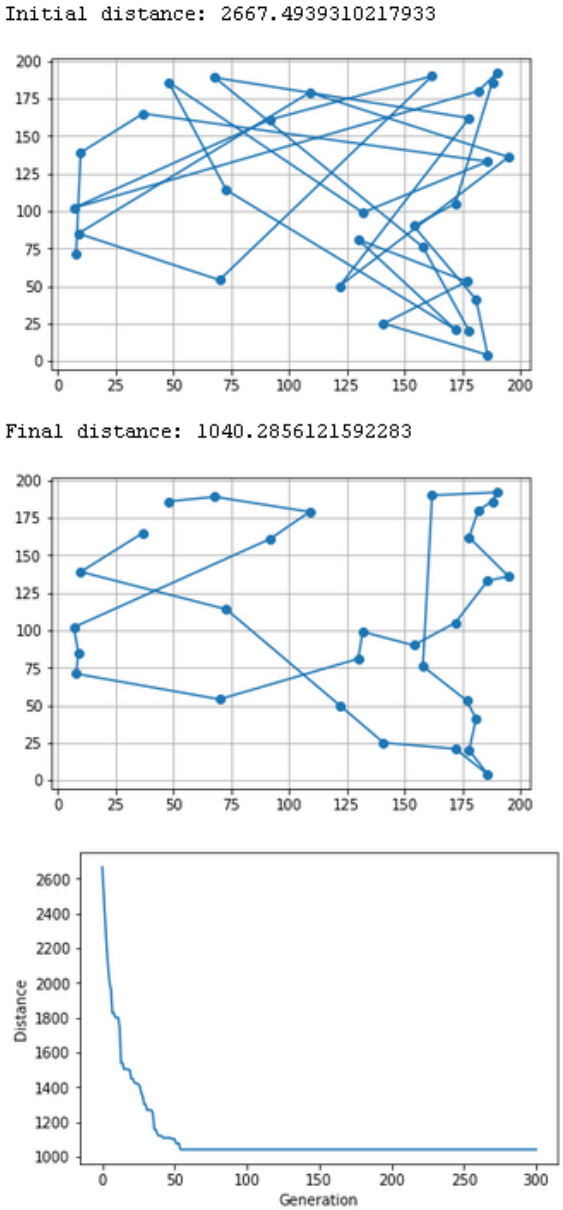

Figure 2: Percorso iniziale e finale ottenuti tramite Rank Selection.

5 Conclusioni

Gli algoritmi genetici offrono uno strumento flessibile e robusto per affrontare problemi di ottimizzazione, esplorando lo spazio di ricerca tramite un approccio che favorisce gli individui più adatti alla sopravvivenza. Seppur non limitati dai problemi che affligono le tecniche di ottimizzazione tradizionali, non esiste la garanzia che un algoritmo genetico trovi la soluzione ottima ad ogni iterazione. Tramite selezione, mutazione e crossover si esplora in maniera più esaustiva possibile lo spazio delle soluzioni, ma la ricerca è un processo stocastico, il che implica che sia possibile mancare la migliore soluzione in assoluto. Per questa ragione bisogna trovare il giusto compromesso tra il numero di generazioni e la probabilità di trovare una soluzione ottima, usando i parametri a disposizione. Tramite l'applicazione mostrata si vuole mostrare come il processo di ottimizzazione sia complesso ed articolato e preveda molteplici possibilità di sviluppo, in base alla scelta di come rappresentare il problema stesso, la codifica dei cromosomi, la tipologia di mutazione o crossover, di come valutare la fitness di un elemento.

Per applicazioni più realistiche si potrebbero usare le città non come punti, ma come coppie (latitudine, longitudine) che vengono inserite in una specifica funzione (Haversine), per calcolare la reale distanza terrestre tra di esse. In generale tuttavia, per determinare in maniera completa l'efficacia di una applicazione, bisogna condurre uno studio estensivo e approfondito su tutte le possibili varianti dell'algoritmo genetico utilizzato.

Bibliografia

[1] David E. Golberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1989

[2] John McCall, *Genetic algorithms for modelling and optimisation*, Journal of Computational and Applied Mathematics 184 (2005) 205–222

[3] Benjamin J. Lynch, *Optimizing with Genetic Algorithms*, Minnesota Supercomputing Institute, 2006.

[4] Andrea G. B. Tettamanzi, *Algoritmi Evolutivi: concetti e applicazioni*, 2005.

[5] Marek Obitko, *Introduction to Genetic Algorithms*