



UNIVERSITÀ DEGLI STUDI DI PERUGIA  
Dipartimento di Matematica e Informatica



## Progetto di Computational Intelligence

More crossover on tsp problem

Laureando:

Fabrizio Fagiolo

Professore:

Prof. Marco Baiocchi

---

Anno accademico 2021/2022

# Indice

<b>1</b>	<b>Obiettivo</b>	<b>4</b>
<b>2</b>	<b>Algoritmi Evolutivi</b>	<b>5</b>
2.1	Algoritmi Genetici . . . . .	6
2.1.1	Funzionamento . . . . .	6
2.1.2	Caratteristiche . . . . .	7
<b>3</b>	<b>Algoritmi Genetici</b>	<b>8</b>
3.1	Pseudo Codice GA . . . . .	8
3.1.1	Inizializzazione della popolazione . . . . .	9
3.1.2	Mating Pool . . . . .	9
3.1.3	Crossover . . . . .	13
3.1.4	Mutazione . . . . .	14
3.1.5	Selezionare la nuova popolazione . . . . .	15
3.1.6	Criteri di Terminazione . . . . .	16
<b>4</b>	<b>Algoritmo CreaTo</b>	<b>17</b>
4.1	Albero Progetto . . . . .	17
4.1.1	Instances . . . . .	18
4.1.2	Instances-Converter . . . . .	20
4.1.3	Log . . . . .	23
4.1.4	Result . . . . .	24
4.1.5	TSP . . . . .	25
<b>5</b>	<b>Risultati ottenuti</b>	<b>55</b>
5.1	Risultati . . . . .	55
5.1.1	Istanza Ulysses 16 (Istanza Piccola) . . . . .	55
5.1.2	Istanza Ulysses 22 (Istanza Medio-Piccola) . . . . .	63

5.1.3	Istanza Berlin 52 (Istanza Media) . . . . .	70
5.1.4	Istanza Gr 96 (Istanza Grande) . . . . .	77
5.1.5	Considerazioni finali . . . . .	84

# Capitolo 1

## Obiettivo

L'obiettivo del progetto è quello di implementare diversi crossover **nell'algoritmo genetico** che risolve **il problema del commesso viaggiatore** (Travelling salesman problem). Il problema del commesso viaggiatore è uno dei casi di studio tipici dell'informatica teorica e della teoria della complessità computazionale.

Il nome nasce dalla sua più tipica rappresentazione:

Dato un insieme di città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza.

# Capitolo 2

## Algoritmi Evolutivi

Un **algoritmo evolutivo** è un algoritmo euristico che si ispira al principio di evoluzione degli esseri viventi. Un algoritmo evolutivo quindi prevede di partire da una soluzione e di farla evolvere con una serie di modifiche casuali fino a giungere ad una soluzione **finale migliore**.

Gli Algoritmi evolutivi si dividono in 3 sottoclassi principali:

1. Algoritmi Genetici (GA)
2. Strategie Evolutive (ES)
3. Programmazione Evolutiva (EP)

Tra questi, i GA si sono rivelati i più popolari dei 3. Questi algoritmi sono simili in generale, ma ci sono grandi differenze tra loro.

### Somiglianze e differenze:

- Tutti e 3 operano su stringhe di lunghezza fissa, che contengono valori reali in ES e EP e numeri binari nel GA canonico.
- Tutti e 3 incorporano un operatore di mutazione: per ES e EP **la mutazione è la forza trainante**. GA e ES utilizzano anche un operatore di **ricombinazione**, che è l'operatore principale per GA (crossover).
- Tutti e 3 utilizzano **un operatore di selezione** che applica una pressione evolutiva, istintiva (in ES e EP, l'operatore determina quali individui saranno esclusi dalla nuova popolazione) o conservante (in GA l'operatore seleziona gli individui per la riproduzione).

- In GA e EP la **selezione è probabilistica**, mentre gli ES utilizzano una **selezione deterministica**. ES e meta-EP consentono l'autoadattamento, in cui i parametri che controllano la mutazione possono evolversi insieme alle variabili oggetto. Infine, vale la pena notare che l'implementatore è libero di modificare questi algoritmi.

## 2.1 Algoritmi Genetici

Un algoritmo genetico è **un algoritmo euristico** utilizzato per tentare di risolvere **problemi di ottimizzazione** per i quali non si conoscono altri algoritmi efficienti di **complessità lineare o polinomiale**. L'aggettivo "genetico", ispirato al principio della **selezione naturale** ed **evoluzione** biologica teorizzato nel 1859 da Charles Darwin, deriva dal fatto che, al pari del modello evolutivo darwiniano che trova spiegazioni nella branca della **biologia** detta genetica, gli algoritmi genetici attuano dei meccanismi concettualmente simili a quelli dei processi **biochimici** scoperti da questa scienza.

### 2.1.1 Funzionamento

Gli algoritmi genetici consistono in algoritmi che permettono di valutare **diverse soluzioni di partenza** (come se fossero diversi individui biologici) e che ricombinandole (come nella riproduzione biologica) ed introducendo elementi di disordine (come nelle mutazioni genetiche casuali) producono nuove soluzioni (nuovi individui) che vengono valutate scegliendo le migliori (selezione ambientale) nel tentativo di convergere verso **soluzioni "di ottimo"**. Ognuna di queste fasi di ricombinazione e selezione si può chiamare **generazione**. Nonostante questo utilizzo nell'ambito dell'ottimizzazione, data la natura casuale dell'algoritmo genetico, non vi è modo di sapere a priori se sarà effettivamente in grado di trovare una **soluzione accettabile al problema considerato**. Se si otterrà un soddisfacente risultato, non è detto che si capisca perché abbia funzionato, in quanto non è stato progettato da nessuno ma da **una procedura casuale**.

### 2.1.2 Caratteristiche

L'algoritmo genetico è l'algoritmo più famoso in letteratura e si applica per risolvere un problema di ottimizzazione. Viene considerata come più famosa **meta-euristica**.

Si basa **su una metafora** semplice da capire:

- L'algoritmo usa una popolazione di individui
- Ciascun individuo è una soluzione del problema ed è rappresentato in genere come una stringa

**Questa popolazione è modificata attraverso 4 operazioni:**

1. Selezione
2. Crossover
3. Mutazione
4. Rimpiazzamento

**Oltre a queste operazioni in piu' abbiamo:**

1.  $f$ : funzione obiettivo
2.  $X$ : Spazio di ricerca
3. Popolazione di  $N$  individui: chiamati cromosomi

Ciascun cromosoma codifica una soluzione, cioè un elemento di  $X$ . Se il cromosoma  $c$  è direttamente la soluzione,  $f$  può essere applicata al cromosoma  $c$ . Altrimenti  $f$  deve essere riscritta (ridefinita) in modo da poter essere applicata a  $c$ , oppure  $c$  deve essere decodificato in modo da ottenere la soluzione corrispondente. La rappresentazione degli individui è detta cromosomi. Ogni individuo è rappresentato come un cromosoma.

Il cromosoma è il patrimonio genetico di un individuo.

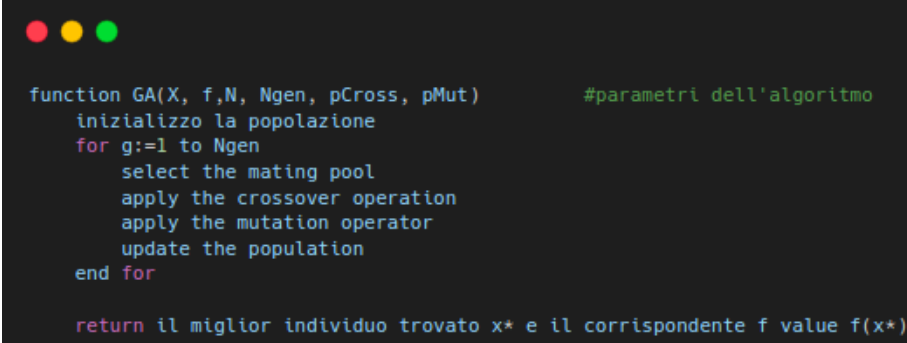
**Altre caratteristiche:**

- Gli GA (genetic algorithm) sono molto studiati in molti articoli scientifici. Sia dal punto di vista teorico che dal punto di vista applicativo.
- GA possono essere applicati a moltissimi problemi di ottimizzazione, sia discreti che continui.
- GA hanno una predisposizione per il discreto. Sono naturalmente applicati a problemi discreti perchè gli individui sono stringhe.

# Capitolo 3

## Algoritmi Genetici

### 3.1 Pseudo Codice GA



```
function GA(X, f, N, Ngen, pCross, pMut)           #parametri dell'algoritmo
  inizializzo la popolazione
  for g:=1 to Ngen
    select the mating pool
    apply the crossover operation
    apply the mutation operator
    update the population
  end for

  return il miglior individuo trovato x* e il corrispondente f value f(x*)
```

Cos'è il miglior individuo trovato?

Molte metauristiche possono produrre, in qualsiasi passaggio, individui che sono peggiori o migliori di quelli prodotti nei passaggi precedenti.

Quindi ha senso memorizzare il miglior individuo trovato fino a quel punto (stato corrente). C'è quindi una variabile nell'algoritmo che mi permette di controllare se il nuovo individuo è migliore di quello momentaneamente memorizzato. Li confronto e in caso positivo salvo quello nuovo. Ciò è molto utile se:

- L'algoritmo non sempre accetta miglioramenti
- L'algoritmo usa qualche meccanismo di restart (non riesco a migliorare allora provo a ripartire)



### 3.1.1 Inizializzazione della popolazione

La popolazione può essere inizializzata in 3 modi principali:

- **Completamente a caso**

- Se il problema non ha vincoli, tutte le soluzioni sono valide.

- **Creare solo cromosomi validi**

- Se il problema ha vincoli (ad esempio il problema dello zaino), significa che non tutti gli individui rappresentano una soluzione valida

- **Creare 'buoni' individui (non così scarsi)**

utilizzando un'euristica  $h$  che mi permette di farlo.

- se  $h$  è deterministica, può produrre un solo individuo. Di conseguenza gli altri  $N - 1$  vanno scelti a caso.
- in generale, usare  $h$  solo per generare soltanto alcuni individui e gli altri generati in modo casuale.

### Differenza tra Euristica e Meta-Euristica

La differenza sta nel fatto che l'euristica è dipendente dal problema mentre la metaeuristica no

### 3.1.2 Mating Pool

Il mating pool  $M$  è un insieme di  $N/2$  coppie di individui presi dalla popolazione (per esempio la popolazione attuale).

- $M$  è usato per il passaggio successivo (Crossover)
- L'idea principale è di scegliere i migliori individui

Per selezionare gli individui ci sono sostanzialmente 2 tecniche:

1. Roulette wheel
2. Tornei

## Roulette wheel (ruota della fortuna)



La ruota della fortuna, seleziona un individuo in modo casuale secondo la probabilità (in modo proporzionale) alla fitness  $F$  per ogni individuo. Quindi un algoritmo genetico ha l'obiettivo implicito di massimizzare il valore di fitness degli elementi della popolazione degli individui.

Si può accontentare anche di un singolo individuo con un alto valore di fitness  $F$ .

### Caratteristiche:

- Se il problema di ottimizzazione è un problema di **massimizzazione**:
  - $F$  può coincidere con la funzione obiettivo  $f$ .
  - Oppure  $F$  è una trasformazione crescente di  $f$ .
- Se il problema di ottimizzazione è un problema di **minimizzazione**:
  - $F$  deve essere una trasformazione decrescente.
  - In questo caso se  $f(x_1) < f(x_2)$  ( $x_1$  è migliore di  $x_2$ ), allora  $F(x_1) > F(x_2)$  ( $x_1$  ha un valore di fitness maggiore di  $x_2$ )

**Il TSP è un problema di minimizzazione.**

Supponiamo che le fitness  $F(x) > 0$  (siano tutte positive) per ogni  $x$ . La probabilità di pescare l'individuo  $x[i]$  è data da:

$$F(x[i]) / (F(x[1]) + F(x[2]) + \dots + F(x[N]))$$

In questo modo genero ogni numero  $x[i]$  con **una probabilità proporzionale** a  $F(x[i])$ .

$F$  può essere considerata anche come **rank** di  $x$  nella popolazione.

- $F=N$  per il miglior individuo
- $F=N-1$  per il secondo miglior individuo
- ...
- $F = 1$  per il peggior individuo

**Quindi Il costo computazionale della singola estrazione è  $O(N)$ .**

## Basata sui Tornei



Nella selezione basata sui suoi tornei **scelgo k individui a caso e scelgo il migliore tra di loro (come in una sfida)**, questo fa sì che conservando i migliori la tecnica sia **più veloce** rispetto a fare la roulette wheel.

il costo di selezionare  $N/2$  coppie è infatti di  $O(kN)$ , invece di  $O(N^2)$  per la roulette wheel

In questo modo il peggior individuo non verrà mai selezionato (non ha chance di essere selezionato perché prendendo anche solo due individui. Il peggiore non sarà mai scelto a meno che tra le selezioni degli individui io posso pescare più volte lo stesso individuo. In questo modo potrei prendere due peggiori e quindi viene selezionato). Questi metodi di selezione possono produrre un mating pool con individui identici

- I migliori individui possono essere rappresentati più volte.
  - Migliore è l'individuo e più coppie potrebbero esserci di lui.
  - I peggiori individui potrebbero anche essere assenti nel mating pool.
- Se un individuo è molto più buono degli altri:
  - con la roulette ci possono essere tante copie di lui a discapito degli altri
  - con i tornei non è detta ma potrebbe esserci comunque un numero abbastanza alto di copie

### 3.1.3 Crossover

L'operazione di crossover è un'operazione che prende due cromosomi  $s_1$  e  $s_2$ , e genera 1 o 2 nuovi cromosomi.

- $s_1$  e  $s_2$  sono chiamati genitori
- I due nuovi cromosomi  $c_1$  e  $c_2$  sono chiamati figli
- Si parte quindi da  $N/2$  coppie di individui:

- $p_1, p_2$
- $p_3, p_4$
- ...
- $p_{[N/2-1]}, p_{[N/2]}$

Ciascuna di queste coppie è copiata e inviata allo step successivo (con probabilità **1-pCross**) oppure è modificata utilizzando l'operatore di crossover (con probabilità **pCross**).

- Ciascuna coppia  $p_{[i]}, p_{[i+1]}$  produce 2 figli  $c_{[i]}, c_{[i+1]}$
- $c_{[i]} = p_{[i]}$  e  $c_{[i+1]} = p_{[i+1]}$  con probabilità 1-pCross
- $c_{[i]}, c_{[i+1]} = \text{crossover}(p_{[i]}, p_{[i+1]})$  con probabilità pCross

All'interno del progetto, sono state implementate varie operazioni di crossover, le andremo ad analizzare successivamente nel dettaglio.

### 3.1.4 Mutazione

Il processo di mutazione per definizione va ad alterare il cromosoma dei figli, dove i figli possono essere sia copie dei genitori o prodotti dal crossover.

Per eseguire una mutazione:

- Crea un nuovo individuo mutando/alterando un figlio appena prodotto dal crossover.
  - $c - - - - > c'$
  - Lo si altera ad esempio cambiando uno o più geni.
- Il crossover poi va ad usare il materiale genetico dalla popolazione
  - Il crossover ricombina tra loro cose che già esistono, non si hanno componenti nuovi per produrre individui. L'originalità è dovuta al fatto che li combino in modo diverso.

**Invece la mutazione può produrre nuove componenti**

#### MUtazione Standard

Può essere usato quando i cromosomi sono vettori o stringhe. Va ad alterare ogni gene con una probabilità  $p_{Mut}$  (probabilità di mutazione).

$c_{[i]} = 0101001110$  (stringa di bit binaria)

$$p_{Mut} = 0.1$$

Significa che in media solo un gene (bit) su 10 viene alterato.

- Con probabilità 1/10 lo altero
- Con probabilità 9/10 lo lascio invariato.

$$c'_{[i]} = 0101011110$$

Come regola generale  $p_{Mut}$  va tenuta bassa.

### 3.1.5 Selezionare la nuova popolazione

Arrivando alla fine del nostro algoritmo dopo tutte le operazioni di crossover e mutazione, dobbiamo andare a selezionare la nuova popolazione creata. Per scegliere la nuova popolazione si hanno questi elementi tra cui scegliere:

- N genitori (elementi della popolazione corrente)
- N figli (prodotti da crossover+mutazione)

Ci sono diverse tecniche per effettuare la scelta:

1. Valutare tutti gli N figli:

- La nuova popolazione è composta dagli N figli (sostituzione dei genitori con i figli). Questo è ciò che accade a lungo andare in natura. Potrebbe tuttavia verificarsi che non tutti i figli siano adatti a vivere in questo ambiente.

2. Elitismo:

La nuova popolazione è composta da

- (a) K migliori individui tra i genitori e i figli
- (b)  $N - K$  figli

3. Sopravvivono i migliori:

- È una condizione particolare del punto precedente in cui  $K = N$ .
- La nuova popolazione è composta dai N migliori individui tra i genitori e i figli.
- Non importa quindi l'età. È possibile che il miglior individuo rimanga sempre nella popolazione (immortale). Ciò è possibile anche nell'elitismo.

### 3.1.6 Criteri di Terminazione



Per la terminazione dell'algoritmo sono necessari dei criteri che facciano terminare affinché il nostro algoritmo possa terminare correttamente.

I principali sono:

1. Dopo *num\_gen* iterazioni/generazioni (criterio di iterazioni)
2. Dopo *num\_sec* secondi (criterio temporale)
  - Svantaggio:
    - dipende dalla velocità di esecuzione del programma. Il criterio ha senso se voglio una risposta velocemente. Il criterio non ha senso se voglio testare algoritmi testati su macchine diverse, è dipendente dalla macchina.
3. Termino quando la funzione obiettivo ha raggiunto un livello prefissato



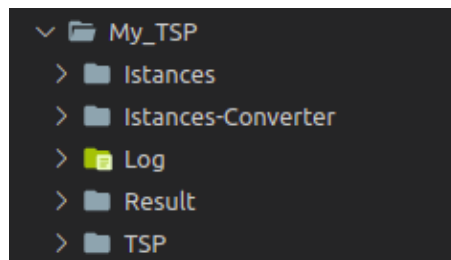
# Capitolo 4

## Algoritmo Creato

### 4.1 Albero Progetto

Per l'organizzazione del progetto ho utilizzato una struttura ad albero in modo da facilitare (spero) la lettura ad eventuali utenti.

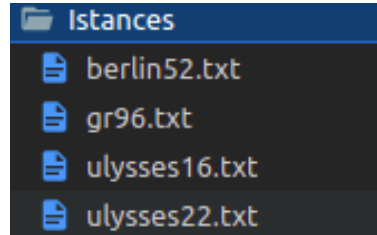
Il progetto ha la seguente struttura:



Com'è possibile vedere sono presenti diverse cartelle, ognuna delle quali contiene informazioni diverse:

- **Istances:** contiene le principali istanze ( non convertite di tsp lib)
- **Istances-Converter:** contiene le istanze di tsp-lib dopo la conversione da lat,lng a coordinate x,y
- **Log:** contiene la registrazione di tutte le soluzioni ed i run seguiti dall'orario
- **Result:** contiene i file.csv che contengono la soluzione
- **TSP:** è la cartella principale dov'è contenuto il codice del GA

### 4.1.1 Instances



Per la creazione dell'algoritmo per prima cosa ho dovuto scegliere le istanze dove andare a lavorare. Ovviamente i test dell'algoritmo sono stati effettuati sulle suddette istanze presenti nella foto.

Le istanze sono state prese dalla libreria:

<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>

Nella libreria sono presenti varie istanze, ad ogni istanza è associata una **best solution** che si può trovare in:

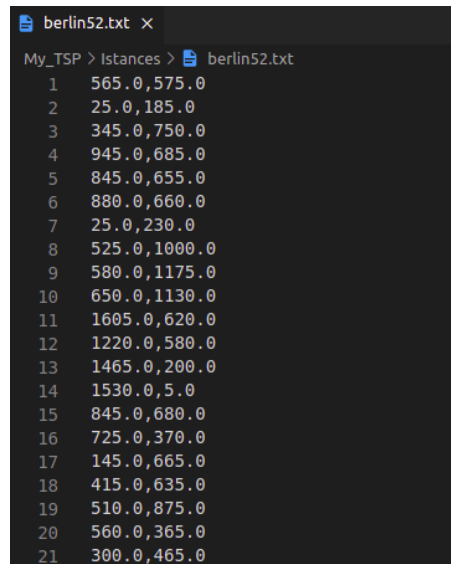
<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html>

A seconda del file selezionato, le coordinate delle istanze presentavano una forma diversa. Riporto nell'esempio sottostante solo quelle utilizzate per lo sviluppo del progetto, nel caso si voglia approfondire l'argomento si può trovare la documentazione al seguente link

<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/DOC.PS>

### Esempi di dicitura:

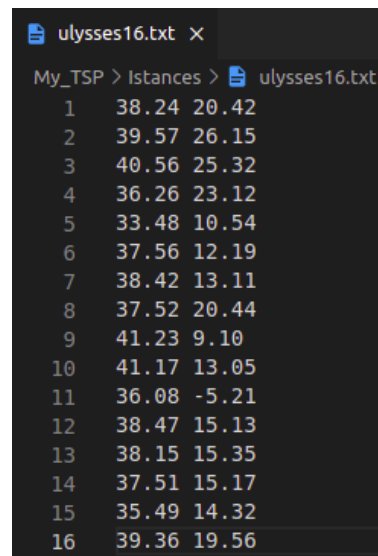
- **EUC\_2D:** i pesi sono riportati in distanza euclidea 2D (x,y)



```
berlin52.txt x
My_TSP > Instances > berlin52.txt
1 565.0,575.0
2 25.0,185.0
3 345.0,750.0
4 945.0,685.0
5 845.0,655.0
6 880.0,660.0
7 25.0,230.0
8 525.0,1000.0
9 580.0,1175.0
10 650.0,1130.0
11 1605.0,620.0
12 1220.0,580.0
13 1465.0,200.0
14 1530.0,5.0
15 845.0,680.0
16 725.0,370.0
17 145.0,665.0
18 415.0,635.0
19 510.0,875.0
20 560.0,365.0
21 300.0,465.0
```

Figura 4.1: l'istanza berlin52 presenta coordinate x,y

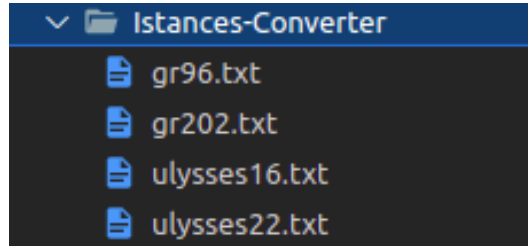
- **GEO:** i pesi vengono rappresentati come coordinate geografiche (lat,lng)



```
ulysses16.txt x
My_TSP > Instances > ulysses16.txt
1 38.24 20.42
2 39.57 26.15
3 40.56 25.32
4 36.26 23.12
5 33.48 10.54
6 37.56 12.19
7 38.42 13.11
8 37.52 20.44
9 41.23 9.10
10 41.17 13.05
11 36.08 -5.21
12 38.47 15.13
13 38.15 15.35
14 37.51 15.17
15 35.49 14.32
16 39.36 19.56
```

Figura 4.2: l'istanza ulysses16 presenta coordinate lat,lng

### 4.1.2 Istances-Converter



All'interno di questa cartella abbiamo quindi le istanze convertite da coordinate lat, lng a coordinate x,y. Ho avuto la necessità di andare a fare la conversione perchè nel codice che vedremo dopo andrò a calcolare la **distanza euclidea** tra 2 punti e ovviamente le coordinate devono essere di tipo x,y e non di tipo lat,lng.

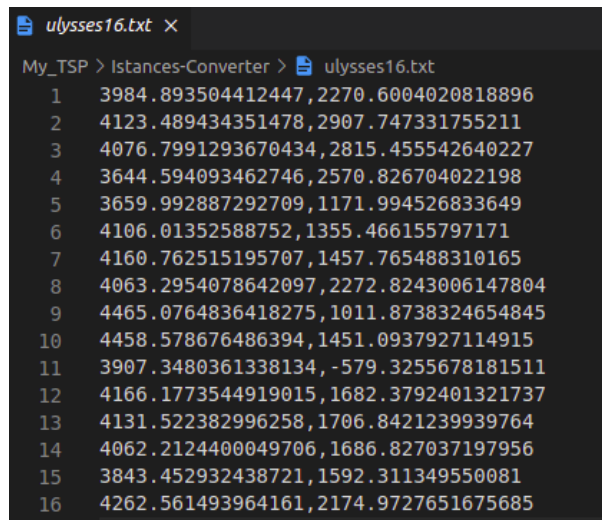


Figura 4.3: file ulysses16 convertito in coordinate x,y

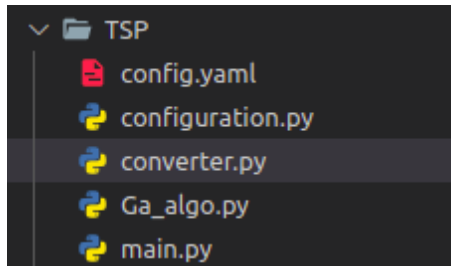
Per effettuare la conversione ho utilizzato uno script che applica la seguente formula:

$$x = r \lambda \cos(\phi_0)$$
$$y = r \phi$$

dove  $r = 6373$  km e  $\phi$ ,  $\lambda$  sono 2 costanti.

I risultati ottenuti, verranno poi approssimanti durante l'esecuzione dell'algoritmo.

## Converter.py



Lo script che si occupa della conversione viene salvato all'interno della cartella TSP insieme agli altri file.py.

Com'è possibile vedere dallo screen successivo

- Vado a leggere i valori non convertiti dal file.txt
- Creo una lista contenente tutti i valori non convertiti
- Applico la formula vista in precedenza a tutti i valori della lista
- Salvo il risultato all'interno della cartella **Instances-converter**.

```

import math

'''
    This script is use to convert the lat,lng coordinate in x,y coordinate.

    For Transformation:
    - Create list with all lat,lng point
    - Calculate function to transoform lat, lng in xy where:
       $x = r \lambda \cos(\varphi\theta)$ 
       $y = r \varphi$ 
    And after return the value.
'''

# Inizialize list of all_point
all_point = []

# Read file txt and point by istances with coordinates lat,lng
with open('/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Istances/gr202.txt') as file:
    for line in file.readlines():
        l = line.rstrip()
        p1 = float(l.split(",")[0])
        p2 = float(l.split(",")[1])
        all_point.append((p1, p2))

# Radio earth
r = 6373 # KM

# Calcolate  $\varphi$ 
phi_0 = all_point[0][1]

# Calcolate  $\cos(\varphi\theta)$ 
cos_phi_0 = math.cos(math.radians(phi_0))

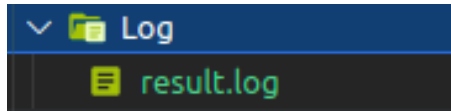
# Function to calcolate xy coordinates
def to_xy(point, r, cos_phi_0):
    lam = point[0]
    phi = point[1]
    return (r * math.radians(lam) * cos_phi_0, r * math.radians(phi))

# Write convert result in a txt file
with open('/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Istances-Converter/gr202.txt', 'w') as file:
    for point in all_point:
        point_xy = to_xy(point, r, cos_phi_0)
        file.write(str(point_xy[0])+' '+str(point_xy[1])+'\n')

```

Figura 4.4: file converter.py

### 4.1.3 Log

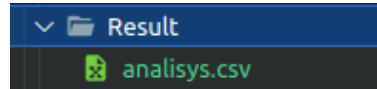


All'interno della cartella log, andremo a salvare:

- **Data e ora** dell'esecuzione
- Il file che contiene l'istanza di input
- Il tipo di **Crossover**
- La lunghezza dell'istanza
- Il numero di **iterazioni/generazioni**
- **Indici** del miglior percorso trovato
- La **best-fitness** relativa al percorso

```
2022-09-14 17:10:11.960 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:10:13.134 | INFO | configuration:configsetters:126 - [16, 15, 21, 17, 12, 0, 9, 0, 1, 18, 8, 11, 20, 3, 13, 4, 7, 2, 14, 16, 5, 19]
2022-09-14 17:10:13.134 | INFO | configuration:configsetters:129 - 12425
2022-09-14 17:10:13.141 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:10:14.322 | INFO | configuration:configsetters:126 - [15, 19, 18, 7, 11, 12, 3, 5, 8, 10, 14, 13, 16, 17, 4, 9, 0, 6, 21, 1, 2, 20]
2022-09-14 17:10:14.322 | INFO | configuration:configsetters:129 - 12234
2022-09-14 17:10:14.328 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:10:15.469 | INFO | configuration:configsetters:126 - [7, 8, 11, 4, 10, 18, 19, 12, 9, 20, 13, 6, 15, 2, 16, 5, 14, 17, 3, 21, 1, 0]
2022-09-14 17:10:15.469 | INFO | configuration:configsetters:129 - 12799
2022-09-14 17:12:26.226 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:13:00.963 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:13:02.187 | INFO | configuration:configsetters:126 - [13, 4, 14, 18, 6, 10, 16, 2, 0, 9, 8, 12, 20, 3, 17, 16, 21, 15, 3, 5, 11, 1]
2022-09-14 17:13:02.187 | INFO | configuration:configsetters:129 - 11898
2022-09-14 17:13:02.194 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:13:03.432 | INFO | configuration:configsetters:126 - [17, 18, 11, 5, 1, 10, 20, 15, 7, 3, 9, 8, 0, 21, 13, 6, 14, 4, 19, 12, 16, 2]
2022-09-14 17:13:03.432 | INFO | configuration:configsetters:129 - 12806
2022-09-14 17:13:03.446 | INFO | configuration:configsetters:80 - {'file_name': '/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Instances-Converter/ulysses22.txt', 'crossover': 'PMX', 'tsp_len': 22, 'iterations': 1000}
2022-09-14 17:13:04.718 | INFO | configuration:configsetters:126 - [11, 8, 17, 1, 5, 19, 18, 3, 16, 2, 4, 13, 14, 12, 10, 9, 21, 20, 7, 0, 6, 15]
2022-09-14 17:13:04.719 | INFO | configuration:configsetters:129 - 11325
```

#### 4.1.4 Result



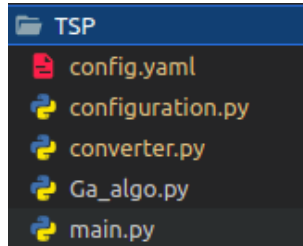
Nella cartella **Result** avremmo come dice il nome della cartella stessa, i risultati generati dall'algoritmo. I risultati sono visibili all'interno del file **analysis.csv**.

Riporto un'esempio di file **analysis.csv**

```
Istances,Geneation,Crossover,Population,Final Value
ulysses22.txt,1000,PMX,22,11890
ulysses22.txt,1000,PMX,22,12006
ulysses22.txt,1000,PMX,22,11325
Mean
11740
```



### 4.1.5 TSP



La cartella **TSP** è il centro del progetto, dove viene eseguito il GA e la maggior parte del codice python.

Andiamo ad analizzare i file:

#### File config.yaml

```
# Configuration file
main:
  file_name: "/home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Istances-Converter/ulysses22.txt"
  crossover: "PMX"
  tsp_len: 22
  iterations: 1000
```

In questo file andiamo a configurare le principali "caratteristiche" dell'algoritmo come:

- **File\_name:** corrisponde al file di input
- **Crossover:** corrisponde a uno dei crossover che è possibile selezionare
- **Tsp\_len:** corrisponde alla lunghezza del problema
- **Iterations:** corrisponde al numero di iterazioni/generazioni dell'algoritmo

## File configuration.py

All'interno del file **configuration.py** per prima cosa abbiamo gli import delle varie librerie utilizzate.

```
#####  
# Libraries used are:  
# numpy - To generate random integers as indices  
# matplotlib - To plot a graph  
# hydra - Create a hierarchical configuration  
# loguru - library which aims to bring enjoyable logging in Python.  
# pandas - for create df  
# pathlib - for split path  
# argparse - to read command from command line  
# csv - to create df in csv  
# statistics - to do a mean  
#####  
  
from Ga_algo import *  
import matplotlib.pyplot as plt  
import numpy as np  
import os  
from hydra import compose, initialize  
from loguru import logger  
import pandas as pd  
from pathlib import Path  
import argparse  
import csv  
from statistics import mean
```

Successivamente abbiamo la classe **Configuration** con il costruttore, dove per prima cosa mi vado a salvare il file di input ovvero il **file\_name** e anche **file di log**

```
class Configuration:  
  
    # Add file .log  
    logger.add('../Log/result.log')  
  
    # Input file  
    file_name = ('../Instances-Converter/ulysses22.txt')  
  
    def __init__(self):  
  
        # Dict for mean df  
        self.d_mean = {  
            'Mean': []  
        }  
  
        # Dict df  
        self.d = {  
            'Instances': [],  
            'Geneation': [],  
            'Crossover': [],  
            'Population': [],  
            'Final Value': [],  
        }  
  
        # Temp list for calc mean  
        self.temp = []  
  
        # List of mean value  
        self.average = []  
  
        # Init value define  
        self.init_values: float  
  
        # Ans value define  
        self.ans_values: list  
  
        # Final value define  
        self.final_value: float  
  
        # Recall create analys  
        self.create_analys()
```

In seguito all'interno del costruttore mi vado ad inizializzare tutti i valori che mi serviranno successivamente come

- Dizionario che contiene la **media**
- Dizionario che contiene:
  - Nome istanza
  - N di Generazioni
  - Il tipo di Crossover
  - Numero della Popolazione
  - Valore finale

Andremo ad usare i dizionari per creare il **DataFrame** finale che vederemo successivamente. Dopo i dizionari abbiamo l'inizializzazione di diverse liste e valori che andranno a contenere:

- **temp** una lista per valori temporanei
- **avarage** lista che conterrà il valore finale della media
- **init\_values** il valore iniziale
- **ans\_value** lista degli indici
- **final\_values** il valore finale

Infine andremo a richiamare la funzione **create\_analisis()** che andremo ad analizzare piu' tardi nel dettaglio.

## Funzione configsetter()

```
def configsetters(self, cfg, plot=False):
    """
    Read configuration of algorithm by the file.yaml
    Plot the path of algorithm
    Writes the best path and the best solution and path in the log file
    """

    """
    Read the configuration by the file.yaml
    """

    tsp_len = cfg.main.tsp_len
    iterations = cfg.main.iterations
    file_name = cfg.main.file_name
    crossover = cfg.main.crossover

    logger.info(cfg.main)

    # Split input file
    split_file_name = Path(file_name).parts

    # Find .txt file extension
    instances = split_file_name[7]

    cwd = os.getcwd()

    # Open and read file input
    with open(file_name, 'r') as fp:
        data = fp.readlines()

    data = [[float(j) for j in i.replace("\n", "").split(',')
              for i in data]

    # Create original data
    original_points = data

    # Create pop_size
    pop_size = len(data)

    # Create the weights matrix
    weights = np.zeros((pop_size, pop_size), dtype=np.float64)

    # Calc euclidean distance of data
    for i in range(pop_size):
        for j in range(pop_size):
            weights[i][j] = (original_points[i][0] - original_points[j]
                             [0])**2 + (original_points[i][1] - original_points[j][1])**2
            weights[i][j] = weights[i][j]**0.5

    # Application of GA_ALGORITHM
    obj = GAalgo(tsp_len, weights, iterations, crossover)

    # Find init value
    init_value = 1/obj.cost(obj.population[0])

    final_value, ans_values = obj.run_algo()

    # Print best value select
    print(final_value)

    # Save log of cycle
    logger.info(ans_values)

    # Save log of best value
    logger.info(final_value)

    # Add value in dict
    d = {'instances': instances,
         'generation': iterations,
         'crossover': crossover,
         'population': pop_size,
         'final_value': final_value,
         }
```

In questa funzione per prima cosa andiamo a leggere il file di configurazione.yaml e ad assegnare all'interno delle variabili i valori letti. I valori passati dal file di configurazione sono quelli visti precedentemente. Vado poi a splittare il **file di input** in modo da ottenere solo la parte contenente il txt.

Esempio:

**file.txt**

Vado poi successivamente a leggere i **pesi** dai file delle istanze e con questi per ogni coppia di vertici vado a calcolare **la distanza euclidea**. Successivamente vado a richiamare l'algoritmo genetico definito nell'altro file e a popolare i valori:

- init\_value
- ans\_value
- final\_value

Stampo il valore della soluzione finale a video e vado ad aggiungere al file di log i rispettivi valori, in piu' con i medesimi vado a popolare il secondo dizionario definito in precedenza nel costruttore.

```

# Control if i would plot graph
if plot:
    # Plot cost graph
    obj.graph()

    # Fetching the best solution
    pts = np.array(original_points)

    # Create pts variable from ans_value that is a return path
    pts = pts[ans_values]

    joining_pts = np.zeros((2, 2))
    joining_pts[0] = pts[-1]
    joining_pts[1] = pts[0]

    # Plot graph of city path
    plt.title("Solution Tour crossover: " + crossover)

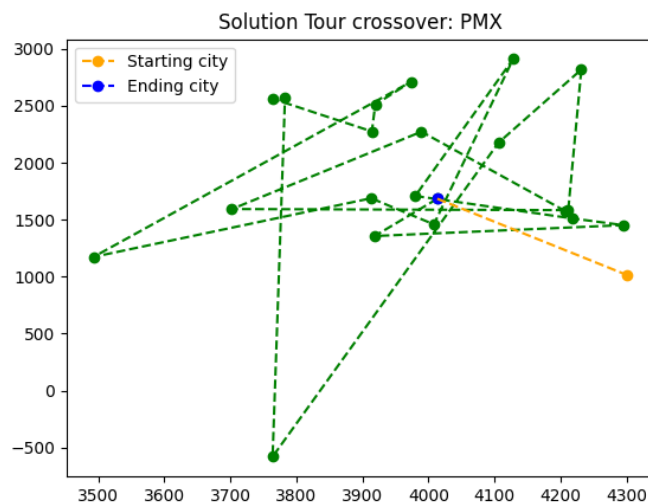
    plt.plot(pts[0][0], pts[0][1], color='orange', marker='o',
             linestyle='dashed', label="Starting city")
    plt.plot(pts[1:, 0], pts[1:, 1], color='green',
             marker='o', linestyle='dashed')
    plt.plot(pts[-1][0], pts[-1][1], color='blue', marker='o',
             linestyle='dashed', label="Ending city")
    plt.plot(joining_pts[:, 0], joining_pts[:, 1],
             color='orange', linestyle='dashed')
    plt.legend()
    plt.show()

return d

```

Sempre all'interno della funzione **configsetter**, vado a controllare se voglio **plottare** il grafico, se voglio plottare il grafico vado a richiamare la funzione **obj.graph()** definita nel altro file che mi mostra il grafico della funzione di **fitness**, successivamente vado a fare il plot delle città. La funzione mi ritorna i valori con cui ho popolato il dizionario.

Esempio di plot del grafico:



## Funzione create\_analisis()

```
def create_analisis(self):
    """
        Read the instruction for number of iteration and plot graph by terminal line
        Exaple:
        Only Iteration ---> python3 main.py -l 3 -o ../Result/
        Plot Graph Whit iteration ---> python3 main.py -l 3 -p -o ../Result/
    """

    # Define parser
    parser = argparse.ArgumentParser()

    # Add iteration with value and letter
    parser.add_argument('-l', '--iterations',
                        help='number of iterations', type=int)

    # Add plot with letter
    parser.add_argument(
        '-p', '--plot', help='plot fitness function and solution tour', action='store_true')

    # Add output with path result
    parser.add_argument(
        '-o', '--output', help="the csv's output path", type=str)

    # Recall parser
    args = parser.parse_args()

    # Initialize with a configuration path relative to the caller
    initialize(version_base=None, config_path=".", job_name="tsp")
    cfg = compose(config_name="config.yaml")

    """
        Create 2 DF.

        .1 DF contains the value of:

            instances, generation, crossover, population, final value

        that change based iterations

        .2 DF contains the mean of all best value

        The result of each other is save on folder /Result in file analisis.csv
    """

    for i in range(args.iterations):
        df = self.configsetters(cfg, plot=args.plot)

        # Populate df by value dict
        self.d['Instances'].append(df['instances'])
        self.d['Geneation'].append(df['geneation'])
        self.d['Crossover'].append(df['crossover'])
        self.d['Population'].append(df['population'])
        self.d['Final Value'].append(df['final_value'])

    # Create 1 DF
    df = pd.DataFrame(data=self.d)

    df.to_csv(os.path.join(args.output, 'analisis.csv'), index=False, encoding='utf-8',
              escapechar='\\t', mode='w')
```

Nella funzione `create_analysys()` andiamo prima a prendere gli argomenti da linea di comando, successivamente andiamo a popolare e creare il primo **dataframe**.

```
def create_analys(self):
    """
    Read the instruction for number of iteration and plot graph by terminal line

    Exaple:
    Only Iteration ---> python3 main.py -i 3 -o ../Result/
    Plot Graph Whit iteration ---> python3 main.py -i 3 -p -o ../Result/

    """

    # Define parser
    parser = argparse.ArgumentParser()

    # Add iteration with value and letter
    parser.add_argument('-i', '--iterations',
                        help='number of iterations', type=int)

    # Add plot with letter
    parser.add_argument(
        '-p', '--plot', help='plot fitness function and solution tour', action='store_true')

    # Add output with path result
    parser.add_argument(
        '-o', '--output', help="the csv's output path", type=str)

    # Recall parser
    args = parser.parse_args()

    # Initialize with a configuration path relative to the caller
    initialize(version_base=None, config_path=".", job_name="tsp")
    cfg = compose(config_name="config.yaml")

    """
    Create 2 DF.

    .1 Df contains the value of:
        instances, generation, crossover, population, final value
    that change based iterations

    .2 DF contains the mean of all best value

    The result of each other is save on folder /Result in file analysis.csv
    """

    for i in range(args.iterations):
        df = self.configsetters(cfg, plot=args.plot)

        # Populate df by value dict
        self.d['Instances'].append(df['instances'])
        self.d['Geneation'].append(df['geneation'])
        self.d['Crossover'].append(df['crossover'])
        self.d['Population'].append(df['population'])
        self.d['Final Value'].append(df['final_value'])

    # Create 1.DF
    df = pd.DataFrame(data=self.d)

    df.to_csv(os.path.join(args.output, 'analysis.csv'), index=False, encoding='utf-8',
              escapechar='\\t', mode='w')
```



Dopo aver creato il primo **Data Frame** vado ad aprire nuovamente il file creato in precedenza e fare la media di tutti i **best\_value** finali, e scrivo la media all'interno dello stesso file in un altro **Data Frame** infine vado stamparlo.

```
# Read the csv create and take the final value for mean
with open('../Result/analysis.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            line_count += 1
        else:
            self.temp.append(float(row[4]))
            mean_result = mean(self.temp)
            mean_result = round(mean_result)
            line_count += 1
            self.average.append(mean_result)

# Insert the mean value on df
self.d_mean['Mean'].append(mean_result)

# Create df
df2 = pd.DataFrame(data=self.d_mean)

# Write df
df2.to_csv('../Result/analysis.csv', index=False, mode='a+')

df2 = pd.read_csv('../home/fabrizio/Scrivania/Much-Cross-Little-Over/My_TSP/Result/analysis.csv')

# Print final df
print(df2.to_string(index=False))
```

Qua sotto riporto un'esempio di plot del **Data Frame**

Instances	Generation	Crossover	Population	Final Value
ulysses22.txt	1000.0	PMX	22.0	12555.0
ulysses22.txt	1000.0	PMX	22.0	12082.0
ulysses22.txt	1000.0	PMX	22.0	11583.0
Mean	NaN	NaN	NaN	NaN
12073	NaN	NaN	NaN	NaN

Figura 4.5: Plot Data Frame creato

## File GA\_algo

All'interno del file GA\_algo.py per prima cosa abbiamo gli import delle varie librerie utilizzate:

```
#####  
# Libraries used are:  
# numpy - To generate random integers as indices  
# random - To shuffle the list of integers  
# copy - To deepcopy a list  
# time - To generate seeds for random numbers generator  
#####  
  
import matplotlib.pyplot as plt  
import numpy as np  
import copy  
import time  
  
# Fixing the crossover and mutation probabilities  
p_crossover = 0.95  
p_mutation = 0.01
```

Insieme alle librerie andiamo a definire la p\_cross che come detto in precedenza deve avere un valore **alto** e la p\_mutation che invece deve avere un valore **basso**. I valori di entrambe le variabili sono stati presi dopo una serie di test, e sono quelli che si comportano in modo "migliore" all'interno dell'algoritmo.

In seguito abbiamo la definizione della classe **GAalgo** con il costruttore.

```
def __init__(self, tsp_len, weights, iterations, crossover_type):  
    # List of all fitness  
    self.all_fitness = []  
  
    # List of generation  
    self.generation = []  
  
    # Initialize iteration  
    self.iterations = iterations  
  
    # Initialize tsp_len  
    self.tsp_len = tsp_len  
  
    # Initialize type of crossover  
    self.crossover_type = crossover_type  
  
    # Enroll element in tsp  
    elements = [i for i in range(tsp_len)]  
  
    # List of population  
    population = []  
  
    # List of improvements  
    self.improvements = []  
  
    # List of cost parent and child  
    self.l_cost = []  
  
    # List of index parent and child  
    self.index = []  
  
    self.best_f = 1e300  
  
    # Select tsp_len randomly permutation return a permuted range.  
    p = np.random.permutation(tsp_len)  
  
    # Swap and create pop_size  
    pop_size = self.tsp_len  
  
    # Append element in population randomly  
    for i in range(pop_size):  
        population.append(list(np.array(elements)[p.astype(int)]))  
        p = np.random.permutation(tsp_len)  
  
    self.population = population  
    self.weights = weights
```

All'interno del costruttore andiamo ad inizializzare tutti i valori che andremo ad utilizzare successivamente nell'algoritmo, in più andiamo a prendere sia gli elementi all'interno del tsp che un valore random, che successivamente verranno trasformati in **np.array** e li andiamo ad inserire all'interno della lista **population**, che poi diventerà la nostra variabile **self.population**

## Funzione cost

```
'''
    Represents the objective function
    And
    Calculate the cost of solution
'''

def cost(self, sol):

    # Inizialize value
    value = 0.0

    # Inizialize weights
    weights = self.weights

    # Calculate the cost of path with the weights
    for i in range(self.tsp_len-1):

        value += weights[sol[i-1]][sol[i]]

    value += weights[sol[-1]][sol[0]]

    # Return value normalize
    return 1/value
```

La funzione `cost()` rappresenta la mia funzione obiettivo e serve per calcolare il costo del percorso dati i pesi. Visto che il tsp è un problema di **minimizzazione** ritorno la trasformazione decrescente: **1/value**

## Funzione Selection

```
'''  
    Function to select pop randomly  
'''  
  
def selection(self, res):  
  
    # Inizialize sum  
    sum = 0.0  
  
    # Create a matrix of the given shape and populate it with random samples from a uniform distribution on [0,1]  
    rvalue = np.random.rand()  
  
    # Create pop after comparison with rvalue  
    j = 0  
    for i in res:  
        sum += i[1]  
        if(sum >= rvalue):  
            ret = j  
            break  
        j += 1  
    return self.population[res[j]][0]]
```

Con la funzione **selection()** vado a selezione i padri dei figli dalla popolazione in base ad un **random value**.

## Function roulette

```
def roulette_wheel(self):  
    # Create dict to append result  
    dict = {}  
  
    # Initialize sum  
    sum = 0.0  
  
    for ind, pop in enumerate(self.population):  
        # Calculate cost of pop  
        val = self.cost(pop)  
  
        # Normalize data and save dict  
        dict[ind] = 1/val  
  
    # Save result order min to max  
    res1 = sorted(dict.items(), key=lambda i: i[1])  
  
    # Create other dict to append result  
    dict = {}  
  
    for ind, pop in enumerate(self.population):  
        # Calculate the cost of pop  
        val = self.cost(pop)  
  
        # Save cost in dict  
        dict[ind] = val  
  
        # Update the sum  
        sum += val  
  
    for j in dict.keys():  
        # Divide key for sum  
        dict[j] = dict[j]/sum  
  
    # Order result save in a dict min to max  
    res = sorted(dict.items(), key=lambda i: i[1])  
  
    # List of pop  
    pop = []  
  
    # List of parents  
    parents = []  
  
    # Swap pop_size  
    pop_size = self.tsp_len  
  
    # Select couple of pop randomly  
    for i in range(int(pop_size/2)):  
        p1 = self.selection(res)  
        p2 = self.selection(res)  
        r = np.random.rand()  
  
        # control random value is better than p_crossover  
        if r <= p_crossover:  
            # execute crossover and mutation  
            c1, c2 = self.crossover(p1, p2, self.crossover_type)  
  
            c1 = self.mutation(c1)  
            c2 = self.mutation(c2)  
        else:  
            c1, c2 = p1, p2  
  
        # Append child1, child2 in new pop  
        pop.append(c1)  
        pop.append(c2)  
  
        # Append parents in parents list  
        parents.append(p1)  
        parents.append(p2)  
  
    self.population = pop  
  
    # Return pop value of child, parents and value of parents  
    return pop, res1, parents, res
```

In questa funzione andiamo a selezionare un individuo in modo casuale secondo la probabilità (in modo proporzionale) alla fitness  $F$  per ogni individuo.

Dove le fitness dei padri e dei figli sono contenute all'interno dei 2 dizionari, e successivamente andranno inserite dentro le variabili:

- `res1`: contiene le fitness dei figli
- `res`: contiene le fitness dei padri

Mentre le liste `pop` e `parents` contengono i relativi indici:

- `pop`: contiene indici dei figli
- `parents`: contiene indici dei padri

Se un **valore random** invece sarà minore della **`p_cross`** verranno eseguite le operazione di mutazione e crossover.

## Funzione mutation()

```
'''
Classic Mutation
Mutation means altering the chromosome of the children.
children can be either copies of the parents or produced by crossover.
Can be used when chromosomes are vectors or strings.
Alters each gene with a probability p_mutation
'''

def mutation(self, c):
    for i in range(self.tsp_len):
        # Create a matrix of the given shape and populate it with random samples from a uniform distribution on [0,1]
        r = np.random.rand()

        # Control if r value matrix is small than p_mutation
        if r <= p_mutation:
            # First index
            ind1 = i

            # Second index is randomly in tsp_len
            ind2 = np.random.randint(self.tsp_len)

            # Swap the index of child
            temp = c[ind1]
            c[ind1] = c[ind2]
            c[ind2] = temp

    return c
```

Qua abbiamo la funzione **mutation()** che esegue una mutazione classica e andiamo ad alterare un elemento in base alla probabilità **p\_mutation** definita precedentemente.

## Funzione crossover()

```
'''
Wrapper per crossover
Richiamo crossover selezionato
'''

def crossover(self, p1, p2, crossover_type):
    # Initialize tsp_len
    tsp_len = self.tsp_len

    # Create 2 cut point random to [0, tsp_len]
    while 1:
        cpoint_1 = np.random.randint(0, tsp_len)
        cpoint_2 = np.random.randint(0, tsp_len)

        # Control equality cut point
        if cpoint_1 == cpoint_2:
            continue

        # If not equal swap cpoint
        else:
            if cpoint_1 > cpoint_2:
                temp = cpoint_1
                cpoint_1 = cpoint_2
                cpoint_2 = temp
            break
```

Nella funzione **crossover** andiamo semplicemente a creare dei cut point di lunghezza randomica, e a richiamare il crossover selezionato dal file di configurazione. La funzione crossover contiene al suo interno tutti gli altri crossover ed in base a quello selezionato andiamo a richiamare l'apposita funzione,

In seguito andiamo a esaminare gli altri operatori di crossover implementati.



## Crossover PMX

```
# Crossover PMX
def crossover_PMX(p1, p2):

    # Create child1
    child1 = copy.deepcopy(p1)

    # Create child2
    child2 = copy.deepcopy(p2)

    # Select cut point by the child that represent the tour of parent
    cpoint_1:cpoint_2+1 = p2[cpoint_1:cpoint_2+1]
    cpoint_1:cpoint_2+1 = p1[cpoint_1:cpoint_2+1]

    # Initialize indices of child1
    child1_indices = [-1 for i in range(tsp_len)]

    # Enroll cpoint1 to cpoint2
    for i in range(cpoint_1, cpoint_2+1):
        # Save new index of child1
        child1_indices[p2[i]] = i

    # Check that i is included in the cut points
    for i in range(tsp_len):
        ind = child1[i]
        if i >= cpoint_1 and i <= cpoint_2:
            continue

        # Transfer index of child1 to parent q and child
        while child1_indices[ind] != -1:
            ind = child1_indices[ind]
            ind = p1[ind]

        child1[i] = ind
        child1_indices[ind] = i

    # Initialize indices of child2
    child2_indices = [-1 for i in range(tsp_len)]

    # Enroll cpoint1 to cpoint2
    for i in range(cpoint_1, cpoint_2+1):
        # Save new index of child2
        child2_indices[p1[i]] = i

    # Check that i is included in the cut points
    for i in range(tsp_len):
        ind = child2[i]

        if i >= cpoint_1 and i <= cpoint_2:
            continue

        # Transfer index of child2 to parent q and child2
        while child2_indices[ind] != -1:
            ind = child2_indices[ind]
            ind = p2[ind]

        child2[i] = ind
        child2_indices[ind] = i

    return child1, child2
```

L'operatore di crossover parzialmente mappato è stato proposto da Gold-berg e Lingle (1985). Esso trasmette le informazioni sull'ordine e sul valore dai tour dei genitori ai tour della genitore. Una parte della stringa di un genitore viene mappata su una parte della stringa dell'altro genitore e le informazioni rimanenti vengono scambiate. Si considerino, ad esempio, i seguenti due tour di genitori:

(12345678) e  
(37516824))

L'operatore PMX crea una genitore nel modo seguente. Innanzitutto, seleziona in modo uniforme e casuale due punti di taglio lungo le stringhe, che rappresentano i tour dei genitori. Supponiamo che il primo punto di taglio sia selezionato tra il terzo e il quarto elemento della stringa e il secondo tra il sesto e il settimo elemento della stringa.

Ad esempio

(123j456j78) e  
(375j168j24))

Le sottostringhe tra i punti di taglio sono chiamate sezioni di mappatura. Nel nostro esempio, esse definiscono le mappature  $4 + -1$ ,  $5 + -6$  e  $6 + -8$ . Ora la sezione di mappatura del primo genitore viene copiata nella seconda discendenza e la sezione di mappatura del secondo genitore viene copiata nella prima discendenza, crescendo:

- figlio 1: (xxj168jxx)
- figlio 2: (xxxj456jxx)

Quindi il genitore  $i$  ( $i = 1,2$ ) viene riempita copiando gli elementi del genitore  $i$ -esimo. Nel caso in cui una città sia già presente nella genitore, viene sostituita in base alle mappature. Ad esempio, il primo elemento della genitore 1 sarà un 1 come il primo elemento del primo genitore. Tuttavia, nella genitore 1 è già presente un 1. Quindi, a causa della mappatura  $1 + -4$ , scegliamo che il primo elemento della genitore 1 sia un 4. Il secondo, il terzo e il settimo elemento della genitore 1 possono essere presi dal primo genitore. Tuttavia, l'ultimo elemento della genitore 1 sarebbe un 8, che è già presente. A causa delle mappature  $8 + -6$  e  $6 + -5$ , si sceglie che sia un 5.

Quindi:

- genitore 1: (4 2 3j1 6 8j7 5))

Con lo stesso procedimento troviamo

- genitore 2: (3 7 8j4 5 6j2 1))

Si noti che le posizioni assolute di alcuni elementi di entrambi i genitori vengono conservate.

## Crossover Cycle

```
# Crossover Order1 (OX1)
def crossover_Order1(p1, p2):

    # Inizialize child1
    c1 = [-1 for i in range(tsp_len)]

    # Create 2 cat point child1
    c1[cpoint_1:cpoint_2+1] = p1[cpoint_1:cpoint_2+1]

    # Create start point child1
    st_point = cpoint_1+1

    for i in range(tsp_len):

        # Control out Index
        if(c1[i] == -1):

            while p2[st_point] in c1:

                # update start point
                st_point += 1

                # Control the end
                if(st_point == tsp_len):
                    st_point = 0

            # Assign child 1 to new parent q start point
            c1[i] = p2[st_point]

    # Inizialize child2
    c2 = [-1 for i in range(tsp_len)]

    # Create cat point child2
    c2[cpoint_1:cpoint_2+1] = p2[cpoint_1:cpoint_2+1]

    # Create start point child2
    st_point = cpoint_1+1

    for i in range(tsp_len):

        # Control out Index
        if(c2[i] == -1):

            while p1[st_point] in c2:

                # update start point
                st_point += 1

                # Control the end
                if(st_point == tsp_len):
                    st_point = 0

            # Assign child2 to new parent p start point
            c2[i] = p1[st_point]

    return c1, c2
```

L'operatore di crossover ciclico è stato proposto da Oliver et al. (1987). Cerca di creare una progenie dai genitori in cui ogni posizione è occupata da un elemento corrispondente di uno dei genitori. Ad esempio, si considerino nuovamente i genitori

- (123456784) e
- (24687531)

Ora scegliamo che il primo elemento della progenie sia il primo elemento del primo tour dei genitori o il primo elemento del secondo tour dei genitori. Quindi, il primo elemento della progenie deve essere un 1 o un 2. Supponiamo di sceglierlo come 1,

(1 \* \* \* \* \*)

Consideriamo ora l'ultimo elemento della discendenza. Poiché questo elemento deve essere scelto da uno dei genitori, può essere solo un 8 o un 1. Tuttavia, se si scegliesse un 1, la progenie non rappresenterebbe un giro legale. Pertanto, si sceglie un 8,

(1 \* \* \* \* \* 8)

Analogamente, troviamo che anche il quarto e il secondo elemento della progenie devono essere selezionati dal primo genitore, il che risulta in

(12 \* 4 \* \* \* 8)

Le posizioni degli elementi scelti finora sono dette un ciclo. Consideriamo ora il terzo elemento della progenie. Questo elemento può essere scelto da uno qualsiasi dei genitori.

Supponiamo di sceglierlo dal genitore 2. Ciò implica che anche il quinto, il sesto e il settimo elemento della discendenza devono essere scelti dal secondo genitore, poiché formano un altro ciclo. Si ottiene quindi la seguente discendenza:

(12647538)

La posizione assoluta della metà degli elementi di entrambi i genitori viene conservata.

Oliver et al. (1987) hanno concluso, sulla base di risultati teorici ed empirici, che l'operatore CX fornisce risultati migliori per il Travelling Salesman Problem rispetto all'operatore PMX.

## Crossover Order1 (OX1)

```
# Crossover Order1 (OX1)
def crossover_Order1(p1, p2):

    # Inizialize child1
    c1 = [-1 for i in range(tsp_len)]

    # Create 2 cut point child1
    c1[cpoint_1:cpoint_2+1] = p1[cpoint_1:cpoint_2+1]

    # Create start point child1
    st_point = cpoint_1+1

    for i in range(tsp_len):

        # Control out index
        if(c1[i] == -1):

            while p2[st_point] in c1:

                # update start point
                st_point += 1

                # Control the end
                if(st_point == tsp_len):
                    st_point = 0

            # Assign child 1 to new parent q start point
            c1[i] = p2[st_point]

    # Inizialize child2
    c2 = [-1 for i in range(tsp_len)]

    # Create cut point child2
    c2[cpoint_1:cpoint_2+1] = p2[cpoint_1:cpoint_2+1]

    # Create start point child2
    st_point = cpoint_1+1

    for i in range(tsp_len):

        # Control out index
        if(c2[i] == -1):

            while p1[st_point] in c2:

                # update start point
                st_point += 1

                # Control the end
                if(st_point == tsp_len):
                    st_point = 0

            # Assign child2 to new parent p start point
            c2[i] = p1[st_point]

    return c1, c2
```

L'operatore order crossover è stato proposto da Davis (1985). L'OX1 sfrutta una proprietà della rappresentazione dei percorsi, secondo cui l'ordine delle città (e non la loro posizione) è importante. Costruisce una progenie scegliendo una città di un sottotour di un genitore e preservando l'ordine relativo delle città dell'altro genitore.

Ad esempio, si considerino i seguenti due tour di genitori:

(12345678) e

(24687531)

e supponiamo di selezionare un primo punto di taglio tra il secondo e il terzo bit e un secondo tra il quinto e il sesto bit. Quindi,

$(12j345j678)$  e  
 $(24j687j531)$

La progenie viene creata nel modo seguente. In primo luogo, i segmenti del tour tra il punto di taglio vengono copiati nella progenie, il che dà come risultato

$(**j345j**)$  e  
 $(**j687j**)$

Quindi, a partire dal secondo punto di taglio di un genitore, si copiano le altre città nell'ordine in cui appaiono nell'altro genitore, sempre a partire dal secondo punto di taglio e omettendo le città già presenti. Quando si raggiunge la fine della stringa del genitore, si continua dalla sua prima posizione. Nel nostro esempio si ottengono i seguenti figli:

$(87j345j126)$  e  
 $(45j687j123)$

## Crossover Order2 (OX2)

```
# Crossover Order2 (OX2)

def crossover_Order2(p1, p2):
    # Select random position
    inds = np.random.randint(tsp_len)

    while inds == 0:
        inds = np.random.randint(tsp_len)

    # List of index
    ind = []

    # Select random index
    for i in range(inds):
        temp = np.random.randint(tsp_len)
        while temp in ind:
            temp = np.random.randint(tsp_len)
        ind.append(temp)

    # Copy p in child1
    c1 = copy.deepcopy(p1)

    # Copy p in child 2
    c2 = copy.deepcopy(p2)

    # Create permute cities by parent p1
    permute_cities = [p1[i] for i in ind]

    for i in range(tsp_len):
        # Control child1 in permute cities
        if(c1[i] in permute_cities):
            # Decrement value
            c1[i] = -1

    k = 0
    for i in range(tsp_len):
        # Control first element
        if c1[i] == -1:
            # Assign child1 new permute cities
            c1[i] = permute_cities[k]

            # Increment k
            k += 1

    # Create permute cities by parent p2
    permute_cities = [p2[i] for i in ind]

    for i in range(tsp_len):
        # Control child2 in permute cities
        if(c2[i] in permute_cities):
            # Decrement value
            c2[i] = -1

    k = 0
    for i in range(tsp_len):
        # Control first element
        if c2[i] == -1:
            # Assign child2 new permute cities
            c2[i] = permute_cities[k]

            # Increment k
            k += 1

    return c1, c2
```

L'operatore di crossover basato sull'ordine (Syswerda 1991) seleziona a caso diverse posizioni in un giro di genitori e l'ordine delle città nelle posizioni selezionate di questo genitore viene imposto all'altro genitore. Ad esempio, consideriamo nuovamente i genitori

(12345678) e  
(24687531)

e supponiamo che nel secondo genitore vengano selezionate la seconda, la terza e la sesta posizione. Le città presenti in queste posizioni sono rispettivamente città 4, città 6 e città 5. Nel primo genitore queste città sono presenti nelle posizioni quarta, quinta e sesta. Ora la progenie è uguale al genitore 1 tranne che per la quarta, quinta e sesta posizione:

(123 \* \* \* 78)

Aggiungiamo le città mancanti alla progenie nello stesso ordine in cui appaiono nel secondo tour dei genitori. Il risultato è

(12346578)

Scambiando il ruolo del primo genitore e del secondo genitore si ottiene, utilizzando le stesse posizioni selezionate,

(24387561)



## Crossover Position

```
# Crossover Position
def crossover_Position(p1, p2):
    # Select random index
    inds = np.random.randint(tsp_len)

    while inds == 0:
        inds = np.random.randint(tsp_len)

    # Create list of index
    ind = []

    # Enroll all inds create
    for i in range(inds):
        # Append all temp i ind
        temp = np.random.randint(tsp_len)

        while temp in ind:
            temp = np.random.randint(tsp_len)

        ind.append(temp)

    # Copy p in child1
    c1 = copy.deepcopy(p1)

    # Copy q in child2
    c2 = copy.deepcopy(p2)

    for i in range(tsp_len):
        # Control all index select in child1 are the same of city parent p2
        if i in ind:
            c1[i] = p1[i]
        else:
            c1[i] = -1

    k = 0
    for i in range(tsp_len):
        # Control all index select in child1 are the same of city parent p1
        if c1[i] == -1:
            while k < tsp_len and p1[k] in c1:
                k += 1
            c1[i] = p1[k]

    for i in range(tsp_len):
        # Control all index select in child1 are the same of city parent p1
        if i in ind:
            c2[i] = p1[i]
        else:
            c2[i] = -1

    k = 0
    for i in range(tsp_len):
        # Control all index select in child1 are the same of city parent p2
        if c2[i] == -1:
            while k < tsp_len and p2[k] in c2:
                k += 1
            c2[i] = p2[k]

    return c1, c2
```

Anche l'operatore basato sulla posizione (Syswerda 1991) inizia selezionando un insieme casuale di posizioni nei tour dei genitori. Tuttavia, questo operatore impone la posizione delle città selezionate alle città corrispondenti dell'altro genitore.

Ad esempio, si considerino i tour dei genitori

(12345678) e  
(24687531)

e supponiamo che vengano selezionate la seconda, la terza e la sesta posizione.

Questo porta alla seguente progenie:

(14623578) e  
(42387651)

## Crossover Alternation

```
# Crossover a posizione alternata (AP)
def crossover_Alternation(p1, p2):

    # Initialize child1
    c1 = [-1 for i in range(tsp_len)]

    k = 0
    for i in range(tsp_len):

        # Control k enroll all element
        if k == tsp_len:
            break

        # Control the element of parent p2 not in in childern1
        if p1[i] not in c1:
            c1[k] = p1[i]
            k += 1

        # Control the element of parent p2 not in in childern1
        if p2[i] not in c1:
            c1[k] = p2[i]
            k += 1

    # Initialize child2
    c2 = [-1 for i in range(tsp_len)]

    k = 0
    for i in range(tsp_len):

        # Control k enroll all element
        if k == tsp_len:
            break

        # Control the element of parent p2 not in in childern2
        if p2[i] not in c2:
            c2[k] = p2[i]
            k += 1

        # Control the element of parent p1 not in in childern2
        if p1[i] not in c2:
            c2[k] = p1[i]
            k += 1

    return c1, c2
```

L'operatore di crossover a posizione alternata (Larranaga et al. 1996) crea semplicemente una progenie selezionando alternativamente l'elemento successivo del primo genitore e l'elemento successivo del secondo genitore, omettendo gli elementi già presenti nella progenie.

Ad esempio, se il genitore 1 è

(12345678)

e il genitore 2 è

(37516824)

l'operatore AP dà la seguente discendenza

(13275468)

Scambiando i genitori si ottiene

(31725468)

```

"""
    Plot graph of best fitness
    by the iteration
"""

def graph(self):

    # Plot graph where on y: all_fitness x: number of generation

    plt.plot(self.generation, self.all_fitness, c='blue')
    plt.xlabel('Generations')
    plt.ylabel('Best Fitness')
    plt.title('Fitness Function')
    plt.show()

```

Alla fine della funzione **Crossover** abbiamo quindi il richiamo dei crossover dopo averli inseriti nel file di configurazione.

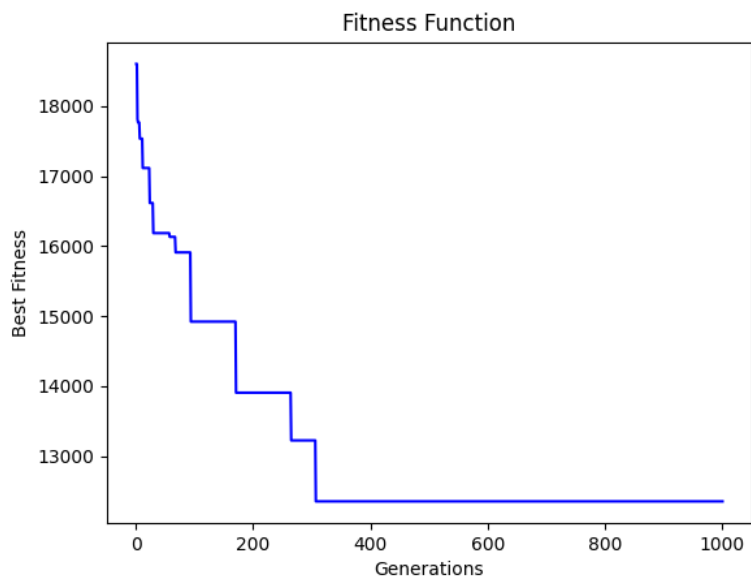
```

if crossover_type == "PMX":
    c1, c2 = crossover_PMX(p1, p2)
elif crossover_type == "Cycle":
    c1, c2 = crossover_Cycle(p1, p2)
elif crossover_type == "Order1":
    c1, c2 = crossover_Order1(p1, p2)
elif crossover_type == "Order2":
    c1, c2 = crossover_Order2(p1, p2)
elif crossover_type == "Position":
    c1, c2 = crossover_Position(p1, p2)
elif crossover_type == "Alternation":
    c1, c2 = crossover_Alternation(p1, p2)
else:
    print("Bad Choice")
    print(
        "Choose from 'PMX', 'Cycle', 'Order1', 'Order2', 'Position', 'Alternation'"
    )
    exit()
return c1, c2

```

### Funzione graph

La funzione **graph** serve per plottare il grafico del **best fitness**



### Funzione `update_best`

La funzione `update_best` si occupa di andare ad aggiornare i valori della **miglior fitness** trovata.

```
'''  
    Function that update the best value  
'''  
  
def update_best(self, x, fx):  
    fx = fx[1]  
  
    if fx < self.best_f:  
        self.best_f = fx  
        self.best = x  
        self.improvements.append((fx))
```

## Funzione run\_algo

```
def run_algo(self):
    pop_size = self.tsp_len

    # Start run algo in seconds
    start = time.time()

    for i in range(1, self.iterations+1):
        # Append generation
        self.generation.append(i)

        # Extract child and parent
        child, res, parents, res2 = self.roulette_wheel()

        # Append index parents
        self.index.append(parents)

        # Append index child
        self.index.append(child)

        # Append cost child
        self.l_cost.append(res)

        # Append cost parents
        self.l_cost.append(res2)

        # Create variable with parents and child index
        l = parents + child

        # Create variable with all cost
        f = res + res2

        # Create list with all pop
        ll = list(range(2*pop_size))

        # order list of index in base at value o function f
        ll.sort(key=lambda i: f[i])

        # Take first part of list
        llbest = ll[:pop_size]

        # Reconstruction new population with l value only for best index and value of f
        po = [l[i] for i in llbest]
        f_obj = [f[i] for i in llbest]

        # Recall function update best
        self.update_best(po[0], f_obj[1])

        path = po[0]

        # Save the best min values
        best_values = min(self.improvements)

        # Approximation best values
        best_values = round(best_values)

        # Save best on all best fitness
        self.all_fitness.append(best_values)

        print("Genetation: {}".format(i),
              "-- Population Size: {}".format(len(self.index)),
              "-- BestFitness: {}".format(best_values))

        # Stop the time of algorithm
        end = time.time()

        # Calculate time of execution for best solution
        total_time = round(end-start, 1)

        # Control total time is lower than 5 minutes
        if total_time > 500:
            break

    print("-----")
    print("Total time: {}s".format(total_time))
    print("-----")
    print("BEST SOLUTION: {}".format(best_values))
    print("-----")

    # Return the best path and the best cost of GA_Algo
    return (best_values, path)
```

La funzione **run\_algo** è la funzione che si occupa dell'esecuzione dell'algoritmo vado a:

1. mettere insieme i padri e i figli.
2. calco la funzione obiettivo per ogni figlio.
3. Ho messo insieme, in un'unica lista, le funzioni obiettivo dei padri (che già avevo) e dei figli.
4. Ho creato una lista che contiene gli indici di tutti (sia padre che figli)
5. Ho ordinato questa lista di indici in base al valore della funzione f.
6. Ho ricostruito la nuova popolazione prendendo i valori di l soltanto per indici migliori e i valori di f.
7. Infine ho chiamato la funzione **update\_best**.

Infine vado a printare la popolazione, il tempo di esecuzione e la **best fitness**, in più la funzione mi ritorna il miglior valore di **fitness** a cui è associato il miglior path.

Ho implementato 2 criteri di terminazione ovvero

1. l'algoritmo termina quando sono terminate **le iterazioni**
2. l'algoritmo termina se viene eseguito per più di 5 min

# Capitolo 5

## Risultati ottenuti

### 5.1 Risultati

Per ogni istanza ho effettuato 2 test per crossover sulle stesse 3 istanze andando a variare la dimensione delle istanze in modo da capire il comportamento dei vari crossover sui problemi. Per questioni di spazio ho omesso i grafici relativi al percorso effettuato e alla fitness.

#### 5.1.1 Istanza Ulysses 16 (Istanza Piccola)

##### Run 1

Generazioni	1000
Crossover	PMX
Lunghezza	16
Solution Instances	6859

Instances	Generation	Crossover	Population	Final Value
ulysses16.txt	1000.0	PMX	16.0	8848.0
ulysses16.txt	1000.0	PMX	16.0	8691.0
ulysses16.txt	1000.0	PMX	16.0	8191.0
Mean				
				8577



## Run 2

Generazioni	10000
Crossover	PMX
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	10000.0	PMX	16.0	7704.0
ulysses16.txt	10000.0	PMX	16.0	6986.0
ulysses16.txt	10000.0	PMX	16.0	7446.0
Mean				
7379				

### Run 3

Generazioni	1000
Crossover	Cycle
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	1000.0	Cycle	16.0	9155.0
ulysses16.txt	1000.0	Cycle	16.0	8495.0
ulysses16.txt	1000.0	Cycle	16.0	8607.0
Mean				8752

### Run 4

Generazioni	10000
Crossover	Cycle
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	10000.0	Cycle	16.0	6989.0
ulysses16.txt	10000.0	Cycle	16.0	7226.0
ulysses16.txt	10000.0	Cycle	16.0	7397.0
Mean				7204

### Run 5

Generazioni	1000
Crossover	Order1
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	1000.0	Order1	16.0	9689.0
ulysses16.txt	1000.0	Order1	16.0	10141.0
ulysses16.txt	1000.0	Order1	16.0	10009.0
Mean				9946

### Run 6

Generazioni	10000
Crossover	Order1
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	10000.0	Order1	16.0	8226.0
ulysses16.txt	10000.0	Order1	16.0	8645.0
ulysses16.txt	10000.0	Order1	16.0	8933.0
Mean				8601

## Run 7

Generazioni	1000
Crossover	Order2
Lunghezza	16
Solution Instances	6859

Instances	Geneation	Crossover	Population	Final Value
ulysses16.txt	1000.0	Order2	16.0	9766.0
ulysses16.txt	1000.0	Order2	16.0	11127.0
ulysses16.txt	1000.0	Order2	16.0	10421.0
Mean				
	10438			

## Run 8

Generazioni	10000
Crossover	Order2
Lunghezza	16
Solution Instances	6859

Instances	Geneation	Crossover	Population	Final Value
ulysses16.txt	10000.0	Order2	16.0	9205.0
ulysses16.txt	10000.0	Order2	16.0	9671.0
ulysses16.txt	10000.0	Order2	16.0	9014.0
Mean				
	9297			

### Run 9

Generazioni	1000
Crossover	Position
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	1000.0	Position	16.0	9286.0
ulysses16.txt	1000.0	Position	16.0	10215.0
ulysses16.txt	1000.0	Position	16.0	9348.0
Mean				9616

### Run 10

Generazioni	10000
Crossover	Position
Lunghezza	16
Solution Instances	6859

Istances	Geneation	Crossover	Population	Final Value
ulysses16.txt	10000.0	Position	16.0	8603.0
ulysses16.txt	10000.0	Position	16.0	8254.0
ulysses16.txt	10000.0	Position	16.0	8095.0
Mean				8317

## Run 11

Generazioni	1000
Crossover	Alternation
Lunghezza	16
Solution Instances	6859

Instances	Geneation	Crossover	Population	Final Value
ulysses16.txt	1000.0	Alternation	16.0	9776.0
ulysses16.txt	1000.0	Alternation	16.0	8298.0
ulysses16.txt	1000.0	Alternation	16.0	10172.0
Mean				9415

## Run 12

Generazioni	10000
Crossover	Alternation
Lunghezza	16
Solution Instances	6859

Instances	Geneation	Crossover	Population	Final Value
ulysses16.txt	10000.0	Alternation	16.0	7970.0
ulysses16.txt	10000.0	Alternation	16.0	8129.0
ulysses16.txt	10000.0	Alternation	16.0	8793.0
Mean				8297

### 5.1.2 Istanza Ulysses 22 (Istanza Medio-Piccola)

Run 1

Generazioni	1000
Crossover	PMX
Lunghezza	22
Solution Istances	7013

Istances	Geneation	Crossover	Population	Final Value
ulysses22.txt	1000.0	PMX	22.0	11687.0
ulysses22.txt	1000.0	PMX	22.0	12554.0
ulysses22.txt	1000.0	PMX	22.0	10849.0
Mean				
11697				

## Run 2

Generazioni	10000
Crossover	PMX
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	10000.0	PMX	22.0	9845.0
ulysses22.txt	10000.0	PMX	22.0	10039.0
ulysses22.txt	10000.0	PMX	22.0	10620.0
Mean				
10168				



### Run 3

Generazioni	1000
Crossover	Cycle
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	1000.0	Cycle	22.0	10075.0
ulysses22.txt	1000.0	Cycle	22.0	8910.0
ulysses22.txt	1000.0	Cycle	22.0	9311.0
Mean				
9432				

### Run 4

Generazioni	10000
Crossover	Cycle
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	10000.0	Cycle	22.0	9012.0
ulysses22.txt	10000.0	Cycle	22.0	9407.0
ulysses22.txt	10000.0	Cycle	22.0	7986.0
Mean				
8802				

### Run 5

Generazioni	1000
Crossover	Order1
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	1000.0	Order1	22.0	13442.0
ulysses22.txt	1000.0	Order1	22.0	13514.0
ulysses22.txt	1000.0	Order1	22.0	11970.0
Mean				12975

### Run 6

Generazioni	10000
Crossover	Order1
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	10000.0	Order1	22.0	12176.0
ulysses22.txt	10000.0	Order1	22.0	12023.0
ulysses22.txt	10000.0	Order1	22.0	11781.0
Mean				11993

### Run 7

Generazioni	1000
Crossover	Order2
Lunghezza	22
Solution Instances	7013

Istances	Geneation	Crossover	Population	Final Value
ulysses22.txt	1000.0	Order2	22.0	12995.0
ulysses22.txt	1000.0	Order2	22.0	13724.0
ulysses22.txt	1000.0	Order2	22.0	13710.0
Mean				
	13476			

### Run 8

Generazioni	10000
Crossover	Order2
Lunghezza	22
Solution Instances	7013

Istances	Geneation	Crossover	Population	Final Value
ulysses22.txt	10000.0	Order2	22.0	12224.0
ulysses22.txt	10000.0	Order2	22.0	12792.0
ulysses22.txt	10000.0	Order2	22.0	12580.0
Mean				
	12532			

### Run 9

Generazioni	1000
Crossover	Position
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	1000.0	Position	22.0	12069.0
ulysses22.txt	1000.0	Position	22.0	11641.0
ulysses22.txt	1000.0	Position	22.0	10872.0
Mean				
11527				

### Run 10

Generazioni	10000
Crossover	Position
Lunghezza	22
Solution Instances	7013

Instances	Geneation	Crossover	Population	Final Value
ulysses22.txt	10000.0	Position	22.0	10188.0
ulysses22.txt	10000.0	Position	22.0	9563.0
ulysses22.txt	10000.0	Position	22.0	10652.0
Mean				
10134				

## Run 11

Generazioni	1000
Crossover	Alternation
Lunghezza	22
Solution Instances	7013

Istances	Geneation	Crossover	Population	Final Value
ulysses22.txt	1000.0	Alternation	22.0	13369.0
ulysses22.txt	1000.0	Alternation	22.0	14159.0
ulysses22.txt	1000.0	Alternation	22.0	12481.0
Mean				13336

## Run 12

Generazioni	10000
Crossover	Alternation
Lunghezza	22
Solution Instances	7013

Istances	Geneation	Crossover	Population	Final Value
ulysses22.txt	10000.0	Alternation	22.0	12136.0
ulysses22.txt	10000.0	Alternation	22.0	11913.0
ulysses22.txt	10000.0	Alternation	22.0	11783.0
Mean				11944

### 5.1.3 Istanza Berlin 52 (Istanza Media)

Run 1

Generazioni	1000
Crossover	PMX
Lunghezza	52
Solution Instances	7542

Istances	Geneation	Crossover	Population	Final Value
berlin52.txt	1000.0	PMX	52.0	23857.0
berlin52.txt	1000.0	PMX	52.0	24063.0
berlin52.txt	1000.0	PMX	52.0	23996.0
Mean				
23972				

## Run 2

Generazioni	10000
Crossover	PMX
Lunghezza	52
Solution Instances	7542

```
Instances      Generation  Crossover  Population  Final Value
berlin52.txt   10000.0     PMX        52.0        23007.0
berlin52.txt   10000.0     PMX        52.0        22472.0
berlin52.txt   10000.0     PMX        52.0        20987.0

      Mean
    22155
```

### Run 3

Generazioni	1000
Crossover	Cycle
Lunghezza	52
Solution Instances	7542

Istances	Geneation	Crossover	Population	Final Value
berlin52.txt	1000.0	Cycle	52.0	21524.0
berlin52.txt	1000.0	Cycle	52.0	21912.0
berlin52.txt	1000.0	Cycle	52.0	22783.0
Mean				22073

### Run 4

Generazioni	10000
Crossover	Cycle
Lunghezza	52
Solution Instances	7542

Istances	Geneation	Crossover	Population	Final Value
berlin52.txt	10000.0	Cycle	52.0	20800.0
berlin52.txt	10000.0	Cycle	52.0	20935.0
berlin52.txt	10000.0	Cycle	52.0	19620.0
Mean				20452



### Run 5

Generazioni	1000
Crossover	Order1
Lunghezza	52
Solution Instances	7542

Instances	Geneation	Crossover	Population	Final Value
berlin52.txt	1000.0	Order1	52.0	24427.0
berlin52.txt	1000.0	Order1	52.0	24741.0
berlin52.txt	1000.0	Order1	52.0	24661.0
Mean				24610

### Run 6

Generazioni	10000
Crossover	Order1
Lunghezza	52
Solution Instances	7542

Instances	Geneation	Crossover	Population	Final Value
berlin52.txt	10000.0	Order1	52.0	22907.0
berlin52.txt	10000.0	Order1	52.0	22925.0
berlin52.txt	10000.0	Order1	52.0	23460.0
Mean				23097

### Run 7

Generazioni	1000
Crossover	Order2
Lunghezza	52
Solution Instances	7542

Instances	Geneation	Crossover	Population	Final Value
berlin52.txt	1000.0	Order2	52.0	24234.0
berlin52.txt	1000.0	Order2	52.0	24431.0
berlin52.txt	1000.0	Order2	52.0	24438.0
Mean				24368

### Run 8

Generazioni	10000
Crossover	Order2
Lunghezza	52
Solution Instances	7542

Instances	Geneation	Crossover	Population	Final Value
berlin52.txt	10000.0	Order2	52.0	21989.0
berlin52.txt	10000.0	Order2	52.0	23744.0
berlin52.txt	10000.0	Order2	52.0	23522.0
Mean				23085

### Run 9

Generazioni	1000
Crossover	Position
Lunghezza	52
Solution Instances	7542

```

Instances      Generation  Crossover  Population  Final Value
berlin52.txt   1000.0      Position    52.0        23645.0
berlin52.txt   1000.0      Position    52.0        23526.0
berlin52.txt   1000.0      Position    52.0        23797.0

      Mean
      23656

```

### Run 10

Generazioni	10000
Crossover	Position
Lunghezza	52
Solution Instances	7542

```

Instances      Generation  Crossover  Population  Final Value
berlin52.txt   10000.0     Position    52.0        22615.0
berlin52.txt   10000.0     Position    52.0        21903.0
berlin52.txt   10000.0     Position    52.0        21865.0

      Mean
      22128

```

## Run 11

Generazioni	1000
Crossover	Alternation
Lunghezza	52
Solution Instances	7542

Instances	Geneation	Crossover	Population	Final Value
berlin52.txt	1000.0	Alternation	52.0	22055.0
berlin52.txt	1000.0	Alternation	52.0	24480.0
berlin52.txt	1000.0	Alternation	52.0	23493.0
Mean				23343

## Run 12

Generazioni	10000
Crossover	Alternation
Lunghezza	52
Solution Instances	7542

Instances	Geneation	Crossover	Population	Final Value
berlin52.txt	10000.0	Alternation	52.0	23213.0
berlin52.txt	10000.0	Alternation	52.0	21725.0
berlin52.txt	10000.0	Alternation	52.0	23516.0
Mean				22818

### 5.1.4 Istanza Gr 96 (Istanza Grande)

Run 1

Generazioni	1000
Crossover	PMX
Lunghezza	96
Solution Instances	55209

Instances	Geneation	Crossover	Population	Final Value
gr96.txt	1000.0	PMX	96.0	301072.0
gr96.txt	1000.0	PMX	96.0	302354.0
gr96.txt	1000.0	PMX	96.0	311364.0
Mean				
304930				

## Run 2

Generazioni	10000
Crossover	PMX
Lunghezza	96
Solution Instances	55209

Istances	Geneation	Crossover	Population	Final Value
gr96.txt	10000.0	PMX	96.0	291205.0
gr96.txt	10000.0	PMX	96.0	301402.0
gr96.txt	10000.0	PMX	96.0	292795.0
Mean				
295134				

### Run 3

Generazioni	1000
Crossover	Cycle
Lunghezza	96
Solution Instances	55209

Instances	Geneation	Crossover	Population	Final Value
gr96.txt	1000.0	Cycle	96.0	295963.0
gr96.txt	1000.0	Cycle	96.0	300863.0
gr96.txt	1000.0	Cycle	96.0	301946.0
Mean				
299591				

### Run 4

Generazioni	10000
Crossover	Cycle
Lunghezza	96
Solution Instances	55209

Instances	Geneation	Crossover	Population	Final Value
gr96.txt	10000.0	Cycle	96.0	289328.0
gr96.txt	10000.0	Cycle	96.0	281303.0
gr96.txt	10000.0	Cycle	96.0	282497.0
Mean				
284376				

### Run 5

Generazioni	1000
Crossover	Order1
Lunghezza	96
Solution Instances	55209

```
Instances  Geneation  Crossover  Population  Final Value
gr96.txt   1000.0      Order1      96.0        300766.0
gr96.txt   1000.0      Order1      96.0        299936.0
gr96.txt   1000.0      Order1      96.0        307158.0

      Mean
    302620
```

### Run 6

Generazioni	10000
Crossover	Order1
Lunghezza	96
Solution Instances	55209

```
Instances  Geneation  Crossover  Population  Final Value
gr96.txt   10000.0    Order1      96.0        298108.0
gr96.txt   10000.0    Order1      96.0        290897.0
gr96.txt   10000.0    Order1      96.0        279975.0

      Mean
    289660
```



### Run 7

Generazioni	1000
Crossover	Order2
Lunghezza	96
Solution Instances	55209

Istances	Geneation	Crossover	Population	Final Value
gr96.txt	1000.0	Order2	96.0	307125.0
gr96.txt	1000.0	Order2	96.0	304045.0
gr96.txt	1000.0	Order2	96.0	302703.0
Mean				
	304624			

### Run 8

Generazioni	10000
Crossover	Order2
Lunghezza	96
Solution Instances	55209

Istances	Geneation	Crossover	Population	Final Value
gr96.txt	10000.0	Order2	96.0	296482.0
gr96.txt	10000.0	Order2	96.0	301258.0
gr96.txt	10000.0	Order2	96.0	282324.0
Mean				
	293355			

### Run 9

Generazioni	1000
Crossover	Position
Lunghezza	96
Solution Instances	55209

```

Instances  Generation Crossover  Population  Final Value
gr96.txt   1000.0   Position      96.0      295514.0
gr96.txt   1000.0   Position      96.0      296540.0
gr96.txt   1000.0   Position      96.0      295816.0

      Mean
295957

```

### Run 10

Generazioni	10000
Crossover	Position
Lunghezza	96
Solution Instances	55209

```

Instances  Generation Crossover  Population  Final Value
gr96.txt   10000.0   Position      96.0      298291.0
gr96.txt   10000.0   Position      96.0      297454.0
gr96.txt   10000.0   Position      96.0      294295.0

      Mean
296680

```

### Run 11

Generazioni	1000
Crossover	Alternation
Lunghezza	96
Solution Instances	55209

Instances	Geneation	Crossover	Population	Final Value
gr96.txt	1000.0	Alternation	96.0	315249.0
gr96.txt	1000.0	Alternation	96.0	299851.0
gr96.txt	1000.0	Alternation	96.0	297793.0
Mean				
	304298			

### Run 12

Generazioni	10000
Crossover	Alternation
Lunghezza	96
Solution Instances	55209

Instances	Geneation	Crossover	Population	Final Value
gr96.txt	10000.0	Alternation	96.0	291596.0
gr96.txt	10000.0	Alternation	96.0	295024.0
gr96.txt	10000.0	Alternation	96.0	291968.0
Mean				
	292863			

### 5.1.5 Considerazioni finali

Andando ad analizzare i dati possiamo fare una stima sui comportamenti dei crossover andando a capire quali si comportano in modo migliore o peggiore in base alle istanze. Vado quindi a fare delle classifiche in base al comportamento che è dato da quanto i valori medi si avvicinano al **gol** (cioè la soluzione dell'istanza).

#### Piccole Istanze (16 città)

1. **Cycle**: 7204
  2. **PMX**: 7379
  3. **Alternation**: 8297
  4. **Position**: 8317
  5. **Order1**: 8601
  6. **Order2**: 9297
- **Best solution possible**: 6859

I risultati sono ottenuti sono la media di **3 esecuzioni** su **10 mila** generazioni

#### Medio-Piccole Istanze (22 città)

1. **Alternation**: 8297
  2. **Cycle**: 8802
  3. **Position**: 10134
  4. **PMX**: 10618
  5. **Order1**: 11993
  6. **Order2**: 11944
- **Best solution possible**: 7013

I risultati sono ottenuti sono la media di **3 esecuzioni** su **10 mila** generazioni

### Medio-Grandi Istanze (52 città)

1. **Cycle:** 20452
2. **Position:** 22128
3. **PMX:** 22155
4. **Alternation:** 22818
5. **Order2:** 23085
6. **Order1:** 23097

- **Best solution possible:** 7542

I risultati sono ottenuti sono la media di **3 esecuzioni** su **10 mila** generazioni

### Grandi Istanze (96 città)

1. **Cycle:** 284376
2. **Position:** 296680
3. **Order1:** 289660
4. **Alternation:** 292863
5. **Order2:** 293355
6. **PMX:** 295134

- **Best solution possible:** 55209

I risultati sono ottenuti sono la media di **3 esecuzioni** su **10 mila** generazioni

Possiamo quindi notare come il crossover **Cycle** ha le prestazioni migliori su istanze sia grandi che piccole come anche il crossover **Position**, mentre sicuramente i crossover **Order1** e **Order2** hanno i comportamenti peggiori in entrambi i casi, ovvero sia istanze piccole che grandi. Possiamo anche notare, come nelle istanze piccole o medio-piccole andiamo ad avvicinarci di molto alla migliore soluzione possibile, mentre nelle altre istanze (medie e grandi) siamo molto lontani dal raggiungimento del **goal**

**Importante:**

Ovviamente i risultati, riportati non sono **ottimizzati** e sono anche soggetti a **molte conversioni**, quindi rappresentano una stima **MOLTO** approssimativa del comportamento.