

# The Orienteering Problem: confronto tra algoritmo genetico ed esatto

Valerio Domenico Conte  
M63001606

## 1 Introduzione

Il **problema di orienteering (OP)** rappresenta una generalizzazione del noto problema del commesso viaggiatore (TSP). Il problema prende spunto da un gioco praticato in zone di montagna dove i concorrenti, armati di bussola e mappa, a partire da un centro di controllo iniziale, devono visitare quanti più centri possibili sulla mappa entro un certo limite di tempo. Ad ogni centro di controllo è infatti associato un punteggio, l'obiettivo è quello di terminare il gioco entro il limite di tempo ritrovandosi in una certa stazione prefissata e con il punteggio complessivo più alto possibile, sotto il vincolo che i centri possono essere visti al più una volta.

La popolarità di questo tipo di **problemi di ottimizzazione combinatoria** è dovuta al fatto che sono facili da formulare e, per dimensioni del problema piccole, sono anche facili da risolvere all'ottimo tramite un algoritmo esatto, cioè l'algoritmo di forza bruta; tuttavia, al crescere della dimensione del problema, l'algoritmo in questione esplode e la complessità di calcolo diventa inaccettabile anche per istanze di dimensioni non troppo grandi. In queste situazioni intervengono allora gli **algoritmi euristici** che riescono a fornire una buona soluzione, anche se questa non corrisponde con quella ottima che fornirebbe invece un algoritmo esatto, in tempi ragionevoli anche per istanze molto grandi. La questione cruciale nell'ambito della progettazione di euristiche è proprio riuscire a trovare il giusto trade-off tra bontà delle soluzioni e velocità di calcolo delle stesse. Le euristiche possibili sono diverse e vengono raggruppate in due grandi categorie:

- *costruttive*: costruiscono gradualmente una soluzione attraverso il passaggio per soluzioni parziali;
- *migliorative*: partono da una soluzione del problema e cercano di modificarla per ottenere una soluzione migliore.

Ai fini del progetto è stato realizzato un algoritmo genetico, che possiamo categorizzare come euristica migliorativa, per risolvere il problema di orienteering; confrontiamo poi i suoi risultati, in termini di bontà delle soluzioni e tempo impiegato per calcolarle, con quelli dell'algoritmo esatto ottenuto tramite la formulazione in PLI dello stesso problema.

## 2 Algoritmi genetici

Gli **algoritmi genetici** sono più correttamente definiti *meta-euristiche*, in quanto possono essere implementati per risolvere non solo istanze diverse di uno stesso problema, ma anche problemi diversi, rivelandosi dunque più generali rispetto alle euristiche specifiche. La peculiarità degli algoritmi genetici è rappresentata dal fatto che simulano un fenomeno fisico, cioè la selezione naturale in campo genetico, per la risoluzione di problemi. In effetti, se immaginiamo le possibili soluzioni di un problema come degli individui di una popolazione, ha senso far sopravvivere e riprodurre (con maggiore probabilità) quelle soluzioni che vengono valutate più promettenti al fine di raggiungere una soluzione finale di buona qualità. Per valutare la bontà di una soluzione si utilizza allora una funzione di fitness che viene calcolata per ogni individuo della popolazione. Il procedimento si ripete per un certo numero di generazioni: all'inizio di ogni generazione, avviene la riproduzione degli individui "migliori" dalla generazione precedente attraverso gli operatori genetici di crossover e gli individui figli prendono il posto dei genitori; i nuovi individui così ottenuti subiscono inoltre delle modifiche da parte degli operatori genetici di mutazione.

Gli algoritmi genetici dunque simulano il processo di selezione naturale partendo da una soluzione iniziale e applicando gli operatori genetici visti. Ogni soluzione ammissibile corrisponde ad

un individuo della generazione. Di seguito troviamo tutti i parallelismi tra la selezione naturale e gli algoritmi genetici:

- gene = variabile decisionale;
- allele = valore delle variabili decisionali;
- cromosoma = insieme delle variabili decisionali;
- genotipo = soluzione ammissibile;
- fitness = funzione obiettivo;
- accoppiamento = procedure di crossover;
- influenza dell'ambiente = altri operatori genetici;
- selezione naturale = algoritmo.

Le fasi caratterizzanti un algoritmo genetico sono:

1. *codifica del problema*: si rappresentano le soluzioni come stringa di variabili di decisione;
2. *inizializzazione e valutazione della fitness*: si genera una popolazione iniziale di soluzioni, alle quali viene associata il proprio valore di fitness;
3. *selezione*: dalla popolazione si selezionano coppie di soluzioni sulle quali verranno applicati gli operatori genetici;
4. *generazione di nuove soluzioni*: si applicano gli operatori genetici alle coppie per generare nuove soluzioni;
5. *sostituzione di elementi della popolazione*: alcuni elementi della nuova popolazione generata devono essere eliminati mentre altri vengono fatti sopravvivere nella generazione successiva;
6. *criterio di arresto*: può essere un numero prestabilito di iterazioni oppure un numero massimo di iterazioni nel corso delle quali la soluzione migliore della popolazione è rimasta la stessa.

Si applicano iterativamente i passi 3, 4 e 5 fin quando non è verificata la condizione di arresto scelta. L'algoritmo converge nel momento in cui gli elementi della popolazione sono tutti più o meno simili: ciò significa che l'operatore di crossover non riesce più a produrre nuovi genotipi e l'algoritmo esplora un sottoinsieme limitato dello spazio delle soluzioni.

### 3 Formulazione PLI del problema

Esistono diverse varianti dell'OP in letteratura scientifica, caratterizzate da regole diverse e quindi da vincoli differenti. In questo progetto ci focalizziamo sul problema di orienteering con nodo iniziale differente da quello finale, chiamato anche **Orienteering Problem with Mandatory Visits** in quanto i centri di partenza e di arrivo sono obbligatori da visitare. Iniziamo col modellare questa variante dell'OP come un **problema di PLI**.

Abbiamo un grafo  $G(V, A)$  pieno dove il nodo  $1 \in V$  rappresenta il punto di partenza mentre il nodo  $n \in V$  rappresenta il punto di arrivo. Ad ogni punto di controllo  $i \in V = \{1, 2, \dots, n\}$  è associato un punteggio  $s_i \geq 0$ . Gli altri dati del problema sono le lunghezze (o tempi di percorrenza)  $d_{ij} \geq 0$  degli archi  $(i, j) \in A$  e la lunghezza (o durata) massima TMAX del percorso soluzione. Come variabili decisionali usiamo:

- $x_{ij} \in \{0, 1\}$  variabile binaria per indicare se l'arco  $(i, j) \in A$  fa parte della soluzione o meno;
- $y_i \in \{0, 1\}$  variabile binaria per indicare se il nodo  $i \in V$  è visitato o meno lungo il percorso soluzione;
- $u_i \in \{0, 1, 2, \dots, n-1\}$  variabile intera che rappresenta l'ordine di visita del nodo  $i \in V$  all'interno della soluzione.

Lo scopo del problema è massimizzare il punteggio finale, dunque la funzione obiettivo è:

$$\max \sum_{i=1}^n s_i y_i$$

Affinché le regole di cui abbiamo parlato vengano rispettate, è necessario imporre i seguenti vincoli:

- il centro di partenza 1 deve avere solo un arco uscente mentre il centro di arrivo  $n$  deve avere solo un arco entrante:

$$\sum_{j=2}^n x_{1j} = \sum_{i=1}^{n-1} x_{in} = 1$$

- per tutti gli altri centri bisogna assicurare la conservazione del flusso, dunque ogni centro intermedio  $i$  visitato lungo il percorso deve avere un arco entrante e un arco uscente; in maniera equivalente, se  $y_i = 1$ , allora deve essere selezionato un arco entrante e un arco uscente per questo nodo:

$$\sum_{\substack{j \in V \\ i \neq j}} x_{ji} = y_i \quad i \in V \setminus \{1\}$$

$$\sum_{\substack{j \in V \\ i \neq j}} x_{ij} = y_i \quad i \in V \setminus \{n\}$$

- i centri di partenza e di arrivo devono essere obbligatoriamente visitati:

$$y_1 = 1, y_n = 1$$

- la lunghezza del percorso può essere al più pari a TMAX:

$$\sum_{i \in V} \sum_{\substack{j \in V \\ i \neq j}} d_{ij} x_{ij} \leq \text{TMAX}$$

Per quanto riguarda i vincoli di assenza di sottogiro, li esprimiamo usando la **formulazione MTZ** (Miller-Tucker-Zemlin) che ci assicura un numero polinomiale ( $n^2$ ) di vincoli. Innanzitutto, il nodo di partenza deve essere ovviamente visitato per primo, dunque gli assegniamo la posizione iniziale:

$$u_1 = 0$$

I vincoli di assenza di subtour sono:

$$u_j - u_i \geq 1 - n(1 - x_{ij}) \quad i, j \in V, i \neq j, j \neq 1$$

Come interpretiamo questi vincoli? Quando  $x_{ij} = 1$  significa che il nodo  $j$  è visitato subito dopo il nodo  $i$ , dunque la variabile  $u_j$  deve essere almeno più grande di uno rispetto a  $u_i$ :

$$x_{ij} = 1 \implies u_j \geq u_i + 1$$

D'altra parte, se  $x_{ij} = 0$  e dunque l'arco  $(i, j)$  non fa parte del percorso soluzione, i casi possibili sono due: l'ordine di visita di  $j$  è successivo a quello di  $i$  oppure la differenza tra l'ordine di visita di  $i$  e quello di  $j$  è al più pari a  $n - 1$ :

$$x_{ij} = 0 \implies u_j \geq u_i \text{ or } u_i - u_j \leq n - 1$$

Notiamo che una "soluzione" che presenta sottogiri ha sicuramente un subtour che non passa per il nodo iniziale: per tale percorso non sarà possibile assegnare un ordine di visita ad ogni nodo in modo tale da soddisfare i vincoli di eliminazione dei sottogiri.

## 4 Risoluzione del problema tramite algoritmo genetico

Per la realizzazione dell'algoritmo è stata utilizzata la libreria **DEAP** (*Distributed Evolutionary Algorithms in Python*), che mette a disposizione strumenti per implementare in maniera semplice algoritmi genetici per differenti tipi di problemi, agevolando la codifica delle soluzioni, la creazione delle popolazioni e la loro evoluzione nel corso delle generazioni. La libreria mette a disposizione anche una varietà di metodi standard che implementano operatori genetici e di selezione. La libreria non limita però la personalizzazione delle procedure, aspetto importante in quanto ai fini del nostro specifico problema è stato necessario implementare delle funzioni ad-hoc per gli operatori genetici (crossover e mutazione), in quanto la codifica adottata per gli individui non è direttamente supportata dalle funzioni di libreria.

## 4.1 Dati del problema

Preliminarmente all'esecuzione dell'algoritmo, viene prelevata l'istanza del problema che contiene il tempo massimo  $Tmax$ , le coordinate cartesiane dei centri di controllo  $(x_i, y_i)$  e i punteggi ad essi associati  $s_i$ . A partire dalle coordinate calcoliamo la distanza  $d_{ij}$  (secondo la metrica euclidea) tra ogni possibile coppia di centri  $(i, j)$ , andando così a popolare una matrice delle distanze *distance\_map*. Dato che il grafo è pieno, ci conviene usare come struttura dati una matrice di adiacenza. Se il numero di centri del problema è  $n$ , allora:

$$distance\_map = \{d_{ij}\} \in \mathbb{R}^{n \times n}$$

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Tale matrice verrà utilizzata dalla funzione di fitness per il calcolo della distanza complessiva che ogni individuo richiede, cioè il tempo del percorso codificato dall'individuo. Le regole prevedono che il primo centro prelevato debba essere assunto come starting point mentre il secondo come ending point; gli score ad essi associati sono entrambi nulli. Al fine di rendere la rappresentazione più intuitiva, le liste contenenti coordinate e punteggi sono state modificate in modo tale che i dati del secondo nodo si trovino in realtà nelle posizioni finali delle liste.

## 4.2 Codifica delle soluzioni

È stata adottata una codifica che prevede individui, dunque soluzioni, di lunghezza variabile, rappresentati come liste di numeri che individuano l'ordine di visita dei nodi all'interno del percorso, differendo in tal modo dalla codifica a lunghezza fissa che invece è direttamente supportata dai metodi della libreria DEAP e che può essere utilizzata per il TSP. La scelta della codifica a lunghezza variabile è una diretta conseguenza delle regole del problema, in quanto l'OP prevede la possibilità che alcuni nodi del grafo non vengano visitati per poter rispettare il vincolo sul tempo massimo. La lunghezza massima degli individui *IND\_SIZE* sarà inferiore di due al numero totale di centri di controllo presenti, in quanto starting point ed ending point sono obbligatori da visitare e il loro ordine è prestabilito, quindi basterà considerarli sempre come se fossero rispettivamente all'inizio e alla fine degli individui.

## 4.3 Individui e popolazione

Il modulo *creator* di DEAP fornisce il metodo *create()* che consente di creare delle nuove classi utili all'algoritmo.

*creator.create(name, base)*

Il primo parametro consente di stabilire il nome della classe mentre il secondo da quale classe deve ereditare, si possono poi passare ulteriori valori utili per istanziare la classe. Il metodo è stato impiegato per creare una classe che rappresenta gli individui, passando come nome "*Individual*" e come classe base *list*; inoltre passiamo al metodo anche la classe della fitness che dovrà essere associata agli individui:

```
creator.create("Individual", list, fitness=creator.FitnessMax)
```

Il modulo *base* della libreria fornisce una classe chiamata *Toolbox* che consente di registrare gli "strumenti" evolutivi (cioè delle funzioni) utili all'algoritmo attraverso il metodo *register()*.

*toolbox.register(alias, function)*

Dopo aver istanziato un oggetto della classe *Toolbox*, possiamo registrare le funzioni di generazione di geni, individui e popolazione:

```
toolbox = base.Toolbox()
```

```
toolbox.register("indices", random.sample, range(IND_SIZE), IND_SIZE)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Il primo valore rappresenta l'alias dello strumento, cioè il nome che useremo per accedere alla funzione che registriamo; il secondo invece è la funzione che deve essere chiamata, dunque a seguire vengono specificati i parametri necessari per tale funzione. Abbiamo stabilito che la codifica delle soluzioni prevede stringhe di numeri senza ripetizioni; i geni sono proprio i numeri all'interno delle stringhe, per cui registriamo nel *toolbox* la funzione che li genera, cioè *sample()* del modulo *random*,

che effettua un campionamento senza ripetizioni degli indici all'interno del range `[0, IND.SIZE[`, scegliendo un numero di elementi pari alla dimensione massima di un individuo `IND.SIZE`. La funzione che crea gli individui è invece `initIterate()` del modulo `deap.tools`, che viene utilizzata per inizializzare un individuo iterando su una funzione generatrice. La funzione generatrice specificata verrà chiamata ripetutamente per produrre i valori che comporranno l'individuo.

`tools.initIterate(container, generator)`

Nel caso degli individui, il container è la classe "Individual" creata prima mentre la funzione generatrice è la funzione di generazione degli indici che abbiamo registrato nel `toolbox`. Infine, la funzione che crea la popolazione è `initRepeat()` del modulo `deap.tools`, che serve a inizializzare una popolazione ripetendo una funzione generatrice per un numero specificato di volte.

`tools.initRepeat(container, generator, n)`

Il container nel caso della popolazione è una lista generica mentre la funzione generatrice è quella di generazione degli individui che abbiamo registrato nel `toolbox`. Il valore per il parametro `n` verrà passato in seguito, quando istancieremo la popolazione, e serve a specificare la dimensione di quest'ultima (cioè il numero di volte che la funzione generatrice deve essere chiamata per riempire il container).

## 4.4 Valutazione della fitness

Utilizzando ancora il metodo `create()` del modulo `creator`, creiamo una classe per la nostra funzione di fitness, che deve essere massimizzata in quanto l'obiettivo del problema è la massimizzazione del punteggio finale (vince il concorrente che termina il gioco conseguendo lo score complessivo più alto, a patto che il percorso seguito non ecceda il limite di tempo assegnato). La classe `FitnessMax` eredita dalla classe `Fitness` del modulo `base` di DEAP, messa a disposizione per misurare la qualità delle soluzioni; come valore dell'attributo `weights` passiamo una tupla contenente dei pesi unitari positivi, stabilendo in tal modo che la funzione di fitness è a massimizzare.

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
```

L'algoritmo genetico calcola il valore della funzione di fitness per ogni individuo affinché possa valutarne la bontà in relazione all'obiettivo. La funzione di *fitness evaluation* implementata opera come segue:

1. inizializza il percorso aggiungendo in testa alla sequenza dell'individuo il nodo origine e in coda il nodo destinazione; inizializza poi sia la variabile *distance* (distanza complessiva) sia la variabile *score* (punteggio complessivo) a 0;
2. per ogni coppia di geni adiacenti all'interno dell'individuo, calcola la distanza tra i punti individuati dai geni della coppia e la aggiunge a *distance*, controllando che il suo valore non ecceda *Tmax*;
3. se *distance* non è ammissibile, termina restituendo 0, altrimenti aggiorna *score* col punteggio associato al secondo gene della coppia; verifica poi se ci sono punti del grafo che giacciono sul segmento avente per estremi le coordinate dei due geni e sottrai da *score* una penalizzazione *penalty* ogni volta che si verifica tale condizione;
4. termina restituendo una tupla il cui primo elemento è *score*.

Il metodo di valutazione della fitness deve essere sempre realizzato dal programmatore quando si utilizza DEAP; l'output deve essere una tupla perché la libreria gestisce i problemi mono-obiettivo come un particolare caso dei problemi multi-obiettivo e la tupla serve a conservare i valori delle due funzioni obiettivo da ottimizzare; è per questo che i pesi vengono inseriti in una tupla all'atto della creazione di `FitnessMax`. Per quanto riguarda il termine di penalizzazione *penalty*, esso è stato impostato pari a 5 e serve a migliorare le prestazioni dell'algoritmo, penalizzando genotipi dove:

1. ci sono archi che passano per dei punti senza che questi siano effettivamente presenti nella sequenza dell'individuo (soluzioni che hanno un punteggio inferiore a quello che potrebbero effettivamente avere);
2. ci sono punti della sequenza che hanno più di un arco entrante/uscente, cioè che vengono visitati più volte (soluzioni inammissibili).

Un punto  $z$  del grafo è collineare con due geni  $x$  e  $y$  (estremi del segmento) se la somma delle sue distanze dagli alleli dei geni è pari proprio alla distanza tra gli alleli stessi:

$$d_{xy} = d_{xz} + d_{yz}$$

## 4.5 Operatore di crossover

Per effettuare la procedura di accoppiamento e riproduzione tra individui che potrebbero essere di dimensioni diverse, si è resa necessaria l'implementazione di una funzione custom da usare come operatore di crossover. La funzione prende in input due individui che rappresentano i genitori e restituisce gli individui figli generati dall'accoppiamento dei genitori con un crossover singolo (cioè scegliamo un solo crosspoint). Vediamo come opera tale metodo:

$$cxVariableLen(ind1, ind2)$$

1. calcola la dimensione dei due individui, memorizzando la minore tra le due in *min\_size*;
2. estrae un indice *cxpoint* in maniera casuale dall'intervallo  $[0, \text{min\_size} - 1]$  da utilizzare come punto di crossover;
3. combina la parte a destra di *cxpoint* del primo genitore (*ind1*) con quella a sinistra di *cxpoint* del secondo genitore (*ind2*) per ottenere i geni del primo figlio; combina poi la parte a sinistra di *cxpoint* del primo genitore con quella a destra di *cxpoint* del secondo genitore per ottenere i geni del secondo figlio;
4. controlla e corregge eventualmente i geni degli individui figli;
5. utilizza il creatore di individui per istanziare i due figli con il corredo genetico ottenuto prima;
6. rimpiazza i genitori con i figli;
7. restituisce i nuovi individui.

Al passo 4 viene chiamata la funzione ausiliaria *fix\_child(child\_genes)*, che serve a correggere il genotipo ottenuto con il crossover; potrebbe verificarsi infatti che la sequenza di geni ottenuta per un figlio contenga degli indici duplicati, il problema però richiede che ogni punto di controllo sia visitato una sola volta per cui i duplicati devono essere eliminati. La funzione di correzione allora prende in input la sequenza di geni ottenuta e va a rimuovere eventuali duplicati preservando l'ordine di visita dei nodi secondo la loro prima apparizione. La funzione di crossover viene registrata nel *toolbox* con il nome "mate":

```
toolbox.register("mate", cxVariableLen)
```

## 4.6 Operatore di mutazione

Anche la funzione di mutazione degli individui è stata implementata appositamente, in particolare utilizza delle probabilità per stabilire quali mutazioni apportare alla sequenza genetica dell'individuo in ingresso.

$$mutVariableLen(individual, prob\_pop, prob\_swap)$$

Le tre operazioni possibili sono:

- rimozione di un gene: si estrae in maniera casuale l'indice della posizione del gene da rimuovere;
- aggiunta di un gene: si estrae in maniera casuale il valore del gene da aggiungere (assicurandoci poi che non sia già presente nella sequenza) e l'indice della posizione dove verrà inserito il gene;
- scambio di posizione tra due geni: si estraggono in maniera casuale due indici diversi e vengono scambiati i geni che si trovano in quelle posizioni.

Chiaramente la rimozione sarà possibile solo se l'individuo ha un numero di geni superiore a 1 mentre l'aggiunta sarà possibile se la lunghezza della sequenza genetica dell'individuo è strettamente inferiore alla massima consentita. Le probabilità di perdita/aggiunta di un gene e di scambio di posizione vengono impostate con la registrazione della funzione "mutate" nel *toolbox*:

```
prob_pop = 0.45
prob_swap = 0.4
```

```
toolbox.register("mutate", mutVariableLen, prob_pop=prob_pop, prob_swap=prob_swap)
```

## 4.7 Operatore di selezione

Nel modulo *tools* di DEAP troviamo diversi operatori di selezione che consentono di simulare la selezione all'interno di una popolazione. I metodi che implementano questi operatori ricevono sempre in ingresso un container iterabile di individui e un valore numerico intero che rappresenta il numero di individui da selezionare all'interno del container. Precisamente la funzione scelta è *selTournament()*, cioè una selezione a "torneo" che sceglie l'individuo migliore tra un numero di individui pari a *tournsize*, estratti casualmente dalla popolazione; questo processo di selezione viene iterato *k* volte. La scelta di *tournsize* individui casuali da cui prelevare successivamente quello migliore ci consente di ridurre il fenomeno di *convergenza prematura*, che potrebbe verificarsi invece quando la selezione è direttamente mirata verso la scelta degli individui con la fitness migliore.

```
tools.selTournament(individuals, k, tournsize, fit_attr='fitness')
```

- *individuals*: lista di individui tra cui scegliere;
- *k*: numero di individui da selezionare;
- *tournsize*: numero di individui che partecipa ad ogni torneo;
- *fit\_attr*: attributo degli individui da usare come criterio di selezione

L'output della funzione è una lista contenente gli individui scelti. I valori per i primi due parametri non dobbiamo fornirli noi, se ne occupa il metodo *eaSimple()* che passa la lista contenente la popolazione e la dimensione di tale lista; il valore per il terzo parametro invece è scelto da noi ed è stato impostato pari a 3 quando registriamo la funzione "select" nel *toolbox*:

```
toolbox.register("select", tools.selTournament, tournsize=3)
```

## 4.8 Algoritmo

Per prima cosa, istanziamo l'oggetto *population* tramite il metodo registrato appositamente nel *toolbox* in precedenza, specificando il numero di individui *n* di cui la popolazione deve essere composta. Definiamo inoltre il numero di generazioni *ngen* che si susseguiranno durante l'evoluzione e i valori delle probabilità di accoppiamento *cxpb* e mutazione *mutpb*. La probabilità di mutazione è del 5% perché è preferibile non alterare eccessivamente il corredo genetico che gli individui figli ereditano dai genitori. Per quanto riguarda la popolazione, si sceglie un valore sufficientemente piccolo per migliorare i tempi di convergenza, tenendo in considerazione il fatto che gli algoritmi genetici lavorano sorprendentemente bene con popolazioni piccole.

```
population = toolbox.population(n=50)
```

```
ngen = 2500  
cxpb = 0.75  
mutpb = 0.05
```

Il modulo *deap.algorithms* fornisce un metodo per avviare un semplice algoritmo genetico, costituito dai passi necessari a eseguire tutte le operazioni di cui abbiamo parlato.

```
algorithms.eaSimple(population, toolbox, cxpb, mutpb, ngen, verbose=False)
```

Possiamo schematizzare il suo funzionamento con il seguente pseudo-codice:

```
evaluate(population)  
for g in range(ngen):  
    population = select(population, len(population))  
    offspring = varAnd(population, toolbox, cxpb, mutpb)  
    evaluate(offspring)  
    population = offspring
```

Per prima cosa il metodo valuta la fitness degli individui nella popolazione iniziale *population* che gli viene fornita. Successivamente, la funzione itera sul numero di generazioni *ngen* che abbiamo fornito, applicando l'operatore di selezione per ottenere gli individui necessari per l'evoluzione. Viene generata la prole della popolazione fatta dagli individui scelti applicando gli operatori di crossover e mutazione registrati nel *toolbox*, con probabilità di accoppiamento *cxpb* e mutazione *mutpb* scelte da noi in precedenza. Si applica quindi la funzione di fitness evaluation sui nuovi individui così generati e la popolazione precedente viene sostituita con la prole. Dopo un certo numero di generazioni, l'algoritmo termina restituendo la popolazione finale e delle statistiche

riguardanti l'evoluzione. Usando il metodo *selBest()* del modulo *deap.tools*, possiamo farci restituire una lista contenente i *k* individui migliori dalla popolazione che gli forniamo:

$$\text{tools.selBest}(\text{individuals}, k)$$

Nel nostro caso, vogliamo ottenere unicamente il migliore individuo della popolazione finale, del quale ci interessa il valore di fitness e il genotipo per poter ricostruire il percorso soluzione del problema.

```
final_pop, _ = algorithms.eaSimple(population, toolbox, cxpb, mutpb, ngen, verbose=False)

best_ind = tools.selBest(final_pop, 1)[0]

print(f"Fitness migliore: {best_ind.fitness.values[0]}")

final_solution = []
final_solution.append(start_node)

for i in range(len(best_ind)):
    final_solution.append(best_ind[i] + 1)

final_solution.append(end_node)
```

Il genotipo dell'individuo migliore rappresenta dunque la migliore soluzione trovata dal nostro algoritmo genetico per l'OP.

## 4.9 Note

La versione utilizzata di DEAP è la 1.4.1. La libreria è open-source e la sua documentazione è consultabile sul sito del progetto [1]. L'algoritmo visto ha bisogno anche delle librerie *random*, *math*, *time* e *matplotlib*. Il codice completo è disponibile su Github [2].



## 5 Confronto tra i risultati

Per valutare le prestazioni dell'algoritmo genetico e confrontare le soluzioni dell'euristica con quelle dell'algoritmo esatto sono state utilizzate due istanze dell'OP di dimensione diversa. Le istanze sono presenti online sul sito della community di ricerca *KU Leuven* [3].

### 5.1 Prima istanza

La prima istanza appartiene al Set 1 di T. Tsiligirides, prevede un valore di  $T_{max}$  pari a 60 e contiene  $n = 32$  punti di controllo; per tale istanza l'algoritmo esatto è in grado di trovare la soluzione ottima. La soluzione migliore calcolata dall'algoritmo genetico è visualizzabile in Figura 1; il valore della funzione obiettivo è **175** e il tempo di calcolo è pari a 3.1 secondi.

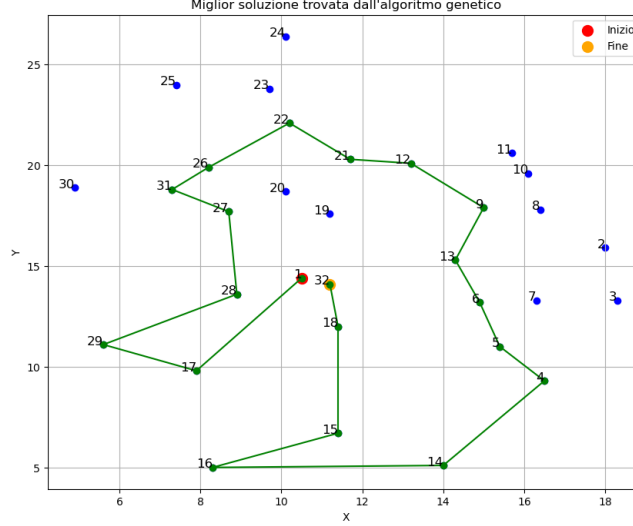


Figura 1: Soluzione migliore per l'istanza 1 fornita dall'algoritmo genetico

La soluzione ottima calcolata dall'algoritmo esatto è invece visualizzabile in Figura 2. Il valore della funzione obiettivo è **225** e il tempo di calcolo è pari a 0.87 secondi.

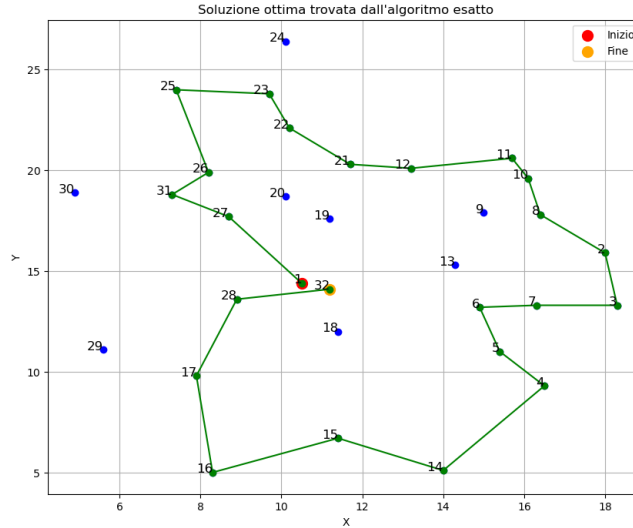


Figura 2: Soluzione ottima per l'istanza 1

La dimensione dell'istanza  $I1$  è sufficientemente piccola affinché l'algoritmo esatto possa risolvere all'ottimo il problema e i risultati vanno a suo vantaggio anche in termini temporali. Lo **scarto percentuale** tra soluzione euristica fornita dall'algoritmo genetico e soluzione ottima fornita dall'algoritmo esatto vale:

$$gap = \frac{|OPT(I1) - EUR(I1)|}{|OPT(I1)|} \cdot 100 = \frac{|225 - 175|}{|225|} \cdot 100 = \frac{50}{225} \cdot 100 = 22.22\%$$

## 5.2 Seconda istanza

La seconda istanza appartiene invece al Set 66 di I. Chao, prevede un valore di  $Tmax$  pari a 120 e contiene  $n = 66$  punti di controllo. Per tale istanza l'algoritmo esatto non riesce a fornire una soluzione ottima. La migliore soluzione trovata dall'algoritmo genetico, mostrata in Figura 3, ha valore di funzione obiettivo pari a **950** e il tempo di calcolo pari a 15.5 secondi.

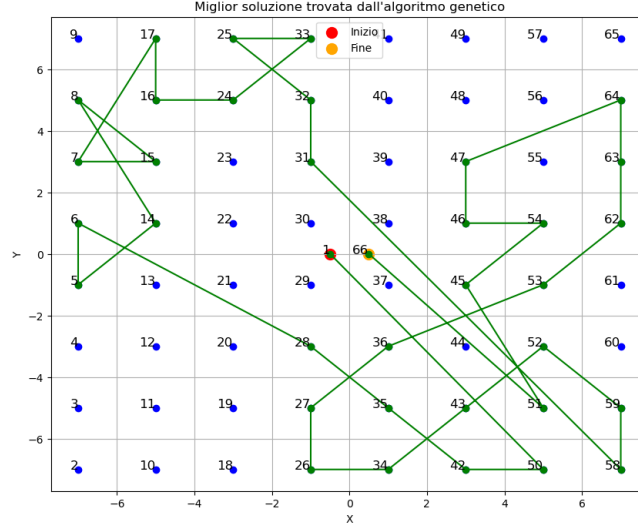


Figura 3: Soluzione migliore per l'istanza 2 fornita dall'algoritmo genetico

Per quanto riguarda l'algoritmo esatto, sebbene non converga all'ottimo in un tempo definito, l'ottimizzatore di *Gurobi* riesce comunque a trovare dei bounds abbastanza stretti per il valore della funzione obiettivo; interrompendo l'ottimizzazione dopo 17.8 secondi otteniamo come valore della funzione obiettivo **1625** e come soluzione quella in Figura 4.

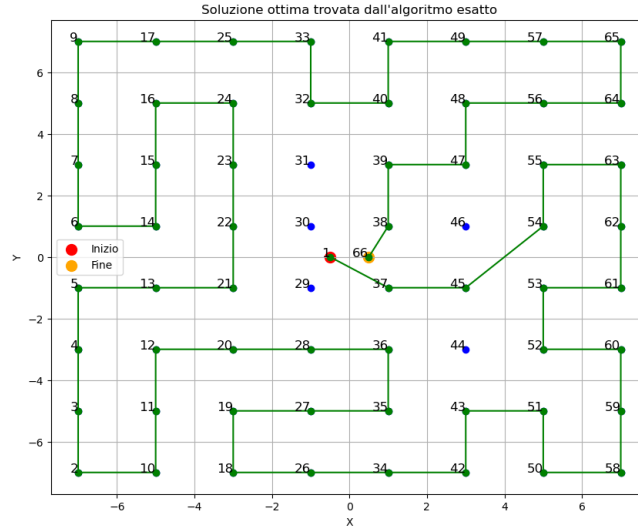


Figura 4: Soluzione migliore per l'istanza 2 fornita dall'algoritmo esatto dopo 17.8 secondi

Per quanto riguarda i limiti al valore della funzione obiettivo, come possiamo vedere in Figura 5, dopo 17.8 secondi il lower bound per l'istanza  $I2$  è  $LB(I2) = 1625$  mentre l'upper bound vale  $UB(I2) = 1645$ . A partire da queste informazioni possiamo calcolare la **stima per eccesso dell'errore** come:

$$gap \leq \frac{|UB(I2) - EUR(I2)|}{|UB(I2)|} \cdot 100 = \frac{|1645 - 950|}{|1645|} \cdot 100 = \frac{695}{1645} \cdot 100 = 42.25\%$$

```

mod.optimize()
[10] ✓ 17.8s
... Gurobi Optimizer version 11.0.1 build v11.0.1rc0 (mac64[arm] - Darwin 23.5.0 23F79)

CPU model: Apple M1
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 4361 rows, 4488 columns and 25678 nonzeros
Model fingerprint: 0x4c8cf083
Variable types: 0 continuous, 4488 integer (4422 binary)
Coefficient statistics:
  Matrix range      [1e+00, 7e+01]
  Objective range   [5e+00, 4e+01]
  Bounds range      [1e+00, 6e+01]
  RHS range         [1e+00, 1e+02]
Presolve removed 5 rows and 70 columns
Presolve time: 0.03s
Presolved: 4356 rows, 4418 columns, 25476 nonzeros
Variable types: 0 continuous, 4418 integer (4353 binary)
Found heuristic solution: objective -0.0000000
Found heuristic solution: objective 35.0000000
Found heuristic solution: objective 50.0000000
Found heuristic solution: objective 70.0000000

Root relaxation: objective 1.62500e+03, 2312 iterations, 0.06 seconds (0.15 work units)

   Nodes |   Current Node |   Objective Bounds |   Work
...
Solution count 10: 1625 1620 1615 ... 1480

Solve interrupted
Best objective 1.62500000000e+03, best bound 1.64500000000e+03, gap 1.2308%

```

Figura 5: Bounds trovati da Gurobi dopo 17.8 secondi per la soluzione ottima dell'istanza 2

## 6 Conclusioni

Come abbiamo potuto constatare, nel caso di istanze di dimensione sufficientemente piccola, come la prima istanza, l'algoritmo esatto batte nettamente quello genetico, in quanto riesce a trovare la soluzione ottima richiedendo un tempo di calcolo comunque inferiore rispetto a quello che impiega l'euristica per trovare una soluzione approssimata. Nel caso di istanze di dimensione più grande, invece, la situazione non è la stessa. Per la seconda istanza abbiamo ottenuto che la stima per eccesso dell'errore è  $gap \leq 42.25\%$ ; se da una parte il valore della stima è abbastanza alto, dall'altra dobbiamo considerare che l'euristica implementata continua a convergere in un tempo finito e accettabile. Il numero di generazioni dell'algoritmo genetico per I1 è 2500 mentre per I2 è 4500; considerando anche il fatto che il I2 ha dimensione due volte più grande di I1 si comprende come mai l'algoritmo risolve I2 impiegando un tempo 5 volte superiore rispetto a quello necessario per I1. Differentemente, l'algoritmo esatto per I1 è estremamente veloce, mentre per I2 non converge proprio e deve essere interrotto.

Gli algoritmi genetici rappresentano una tra tante possibilità di implementazione di euristiche per problemi di ottimizzazione combinatoria; la loro forza è proprio dovuta al fatto che non sono specifici per un certo problema e dunque i loro step di risoluzione possono essere adattati a problemi di tipologie differenti. Lo svantaggio chiaramente sta nella bontà delle soluzioni, che spesso sono abbastanza distanti dall'ottimalità, anche se nei casi analizzati non risultano mai essere peggiori del doppio della soluzione ottima e possiamo ritenerle di buona qualità.

## Riferimenti bibliografici

- [1] DEAP Project. *DEAP Documentation*. URL: <https://deap.readthedocs.io/en/master/index.html>.
- [2] Valerio D. Conte. *The Orienteering Problem. Risoluzione tramite algoritmo genetico*. URL: [https://github.com/valeriooconte/orienteering\\_problem\\_ga](https://github.com/valeriooconte/orienteering_problem_ga).
- [3] KU Leuven - Division CIB. *The Orienteering Problem: Test Instances*. URL: <https://www.mech.kuleuven.be/en/cib/op#autotoc-item-autotoc-0>.