Algoritmo_esatto_Orienteering

July 3, 2022

1 Algoritmo esatto per un problema di Orienteering

1.1 Cos'è un problema di Orienteering?

Il problema dell'**Orienteering (OP)** consiste nel generare percorsi vincolati nel **tempo TMAX (o nella distanza DMAX)** attraverso un grafo, per un giocatore (nel caso di applicazione reale in esame un turista), in modo tale che lo **score** ottenuto visitando i nodi venga massimizzato. Ciascun tour di un giocatore inizia in un nodo (**start point**) diverso dal nodo di terminazione (**end point**).

Il problema dell'Orienteering (OP), presenta diverse generalizzazioni e varianti. Nel seguente elaborato si tratta il problema **deterministico**, in cui un giocatore sceglie l'insieme di nodi da visitare in maniera tale che il percorso di visita non ecceda l'intervallo di tempo specificato TMAX (o la distanza massima DMAX). Questo è un problema molto generale che si trova in numerose applicazioni, compreso il problema della **pianificazione di un tour turistico**.

1.2 Lo studio del problema di un'applicazione reale dell'Orienteering

Si studia il problema della generazione automatica ed efficiente di itinerari per turisti che vogliono visitare il **sito archeologico di Pompei**. Fare il piano di visita che include i **Point Of Interest** più interessanti da visitare, per la distanza o tempo a disposizione, è solitamente un problema complesso. Pertanto si creano sistemi che tendono a massimizzare il più possibile la soddisfazione del turista realizzando un piano di viaggio personalizzato.

Solitamente, i vincoli presi in considerazione dai sistemi di pianificazione in discussione sono: - ubicazione geografica dei POI (coordinate); - le distanze tra i POI; - punteggio di ciascun POI, ecc.

Il problema più semplice nella pianificazione del tour può essere studiato come un problema di Orienteering (OP), dove viene fornito un numero n di POI, ciascuno con un punteggio S_i. L'obiettivo è quello di avere un unico tour che includa il maggior numero di POI possibile, in modo da massimizzare il fattore di soddisfazione del viaggio. Si risolverà prima il problema con un metodo esatto, per poi valutarne le differenze attraverso degli approcci euristici.

1.3 Formulazione matematica del problema

L'obiettivo è pianificare un tour, con distanze massime variabili, che servirà al turista per la visita di un certo numero di ambienti del sito archeoligico/POI (Point Of Interest).

In generale, da un punto di vista descrittivo si possono definire i seguenti **vincoli**: - non tutti gli ambienti possono essere visitati durante la visita, poichè i km percorsi durante la visita è limitato ad un certo *Dmax*; - ogni ambiente può essere visitato al massimo una volta.

L'**obiettivo** è trovare una visita che includa il maggior valore di soddisfazione totale tenendo conto della distanza massima prefissata del viaggio.

Per dimostrare la bontà del modello, si è provato ad applicarlo a delle **istanze Benchmark** al modello esatto e alle euristiche confrontandone i risultati. Soltanto dopo, si è applicato il modello all'applicazione reale descritta finora.

1.3.1 Modellazione

Sulla base dei fatti sopra indicati, il problema di pianificazione può essere definito con le seguenti espressioni matematiche:

- **Variabili decisionali:**
  - $x_{ij} = 1$ se l'arco (,) appartiene al percorso seguito dal turista, 0 altrimenti
  - $= 1$ se il nodo è visitato, 0 altrimenti
- **Funzione Obiettivo:**

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

La funzione obiettivo (1) è massimizzare il valore di soddisfazione dell'ambiente (Score).

- **Vincoli:**

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

Il vincolo (2) garantisce che il percorso inizi da uno *start point* e termini ad un *end point.*

$$\sum_{i=1 i \neq j}^{n} x_{ij} = \sum_{i=1 i \neq j}^{n} x_{ji} \qquad j = 2..n-1 \tag{3}$$

Il vincolo (3) garantisce che ogni nodo abbia un arco entrante e un arco uscente, in modo tale che ogni POI (diverso dallo start point e dall'end'point) venga visitato soltanto una volta.

$$\sum_{i=1 i \neq j}^{n} x_{ij} = y_j \qquad j = 1..n \tag{4}$$

Il vincolo (4) garantisce che il generico nodo j è visitato se ha un arco entrante.

$$\sum_{i=1}^{n} \sum_{j=1 i \neq j}^{n} d_{ij} x_{ij} \leq DMax \tag{5}$$

Il vincolo (5) garantisce che il percorso non può avere una distanza maggiore di **DMAX**.

Per evitare che nella soluzione calcolata ci siano dei sottogiri sarà allora necessario eliminarli attraverso i vincoli di assenza di sottogiri:

Si applicano sia i *Lazy Constraint* utilizzando i seguenti vincoli:

$$\sum_{(i,j)\epsilon S} x_{ij} \leq |S| - 1 \qquad S \subset V \qquad 2 \leq |S| \leq |V| - 1 \tag{6}$$

e sia gli *MTZ*, aggiungendo un ulteriore variabile di decisione u_i, che determina l'ordine di visita dell'i-esimo nodo

$$u_1 = 0 \tag{7}$$

$$0 \leq u_i \leq n - 1 \qquad intera \qquad i = 1..n \tag{8}$$

$$u_j - u_i \geq 1 - n(1 - x_{ij}) \qquad i, j = 1..n, i \neq j, j \neq 1 \tag{9}$$

Ovviamente, dal punto di vista del risultato le differenze non sono apprezzabili, ma dal punto di vista dei tempi di esecuzione si notano notevoli differenze al crescere delle dimensioni del problema, come si vedrà in seguito.

Vincoli di fisica realizzabilità

$$x_{ij}, y_{ij} \quad \in 0, 1 \qquad i, j \in A \tag{10}$$

1.4 Algoritmi risolutivi

Per la risoluzione del problema ci avvaliamo dei seguenti algoritmi risolutivi utilizzando Python 3 ed eseguendo gli script su macchine che montano un processore Intel Core i7 (11 esima generazione, quad-core, 8 thread) e 16 GB di RAM alla frequenza di 3200MHz.

1.4.1 Algoritmo esatto

L'approccio seguito è mostrato nel seguente diagramma:

Lettura dati dal .txt Per il test del modello è possibile utilizzare le seguenti istanze Benchmark: - Bench32 - Bench52 - Bench102

In particolare, ogni riga dell'istanza Benchmark (Bench32, Bench52, Bench102) ha questo formato: - x y S

dove: x = x coordinate y = y coordinate S = score

Osservazioni: - La prima riga indica lo *start point* - L'ultima riga indica l' *end point*

Per il test del modello dell'applicazione reale è possibile utilizzare le seguenti istanze create dagli AUTORI: - Scavi_Archeologici_Pompei_Anfiteatro_Misteri - Scavi_Archeologici_Pompei_Marina_Misteri - Scavi_Archeologici_Pompei_Marina_Anfiteatro

N.B. Occorre inserire nella riga "namefile" il nome della relativa istanza *N.B.* Le istanze dell'applicazione reale sono state scelte sulla base di differenti start_point e end_point

[ ]:
```
import pandas as pd

siti = []
```

```
coordinate = {}
score = {}

#Parametri del problema

#Si scelga il metodo per l'eliminazione dei subtour:
MTZ=0 #se è 1 uso gli MTZ, altrimenti uso i lazy constraints

#Si scelga il DMAX
"""
    Per il test dei Benchmark usare i seguenti DMAX: 20,50,70
    Per il test dell'applicazione reale usare i seguenti DMAX: 0.02,0.03,0.04
    Tale differenza dei DMAX è dovuta dalla differenza delle distanze valutate␣
 ↪tra i vari nodi

"""
DMAX=0.04

#Si inserisca l'istanza indicata nel riquadro superiore
namefile = "Scavi_Archeologici_Pompei_Anfiteatro_Misteri"
with open('./IstanzeBenchmark/' + namefile + '.txt') as file:
    for i, line in enumerate(file):
            node = line.split(',')
            if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
                namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
                namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro" ):
                sito = str(node[0])
                siti.append(sito)
                coordinate[sito] = (float(node[2]),float(node[1]))
                score[sito] = int(node[3])
                zoom = 16
                raggio = 900
            else:
                sito = str(i)
                siti.append(sito)
                coordinate[sito] = (float(node[1]),float(node[0]))
                score[sito] = int(node[2])
                zoom = 5
                raggio = 5000000
```

Calcolo distanza tra due POI Per la valutazione delle distanze tra i POI si è utilizzata la metrica Euclidea. Siano due punti P_i e P_j allora:

$$d(P_i, P_j) = [(x_i - x_j)^2 + (y_i - y_j)^2]^{(\frac{1}{2})} \tag{11}$$

[ ]:
```
import math

def distance(sito1, sito2):
    c1 = coordinate[sito1]
    c2 = coordinate[sito2]

    #Metrica di Euclide per la valutazione della distanza tra due punti
    diff = (c1[0]-c2[0], c1[1]-c2[1])
    return (math.sqrt(diff[0]*diff[0]+diff[1]*diff[1]))

dist = {(c1, c2): distance(c1, c2) for c1 in siti for c2 in siti if c1 != c2 }
```

Calcolo del punto medio tra quelli dati per rappresentarli sulla mappa Utile per la visualizzazione ottimale della mappa su Folium.

[ ]:
```
media_lat=0
media_long=0

for sito in siti:
    media_lat = media_lat + coordinate[sito][0]
    media_long = media_long + coordinate[sito][1]

lat=media_lat/len(coordinate)
long=media_long/len(coordinate)
```

Visualizzazione dei nodi del problema sulla mappa

[ ]:
```
#Marker sulla mappa: I nodi del problema
import folium
map = folium.Map(location=[lat,long], zoom_start = zoom)
for sito in siti:
    folium.Marker(location = coordinate[sito], tooltip = sito, icon=folium.
 ↪Icon(color='darkred')).add_to(map)

# add search area circle
folium.Circle(radius=raggio, location=[lat,long], color='darkred').add_to(map)

map
```

Definizione del modello su Gurobi per l'algoritmo esatto Si inizializza il modello e si definiscono le variabili di decisione x_{ij} e y_i

[ ]:
```
#MODELLAZIONE GUROBI
import numpy as np
import gurobipy as gp
from gurobipy import GRB
```

```
if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro" ):
    model = gp.Model('Pompei_orienteering') #definizione del modello
else:
    model = gp.Model(str(namefile) + '_orienteering') #definizione del modello

#Definizione variabili di decisione
Xvars = model.addVars(dist.keys(), obj = score, vtype = GRB.BINARY, name = 'x')␣
 ↪#xij
Yvars = model.addVars(siti, obj = 0.0, vtype = GRB.BINARY, name = 'y')  #yi
```

Funzione obiettivo

$$max \sum_{i=1}^{n} s_i y_i \tag{12}$$

[ ]:
```
#Definizione della funzione obiettivo
obj = model.setObjective(gp.quicksum(Yvars[i]*score[i] for i in siti), gp.GRB.
 ↪MAXIMIZE)
```

Vincoli sullo start point e l'end point

$$\sum_{j=2}^{n} x_{1j} = \sum_{i=1}^{n-1} x_{in} = 1 \tag{13}$$

[ ]:
```
start_point = siti[0]
#Dal nodo di partenza (si indichi quale) esce soltanto un nodo
OutFirst = model.addConstr(Xvars.sum(start_point,'*') == 1)
```

[ ]:
```
end_point = siti[len(siti)-1]
#Dal nodo di uscita (si indichi quale) esce soltanto un nodo
InLast = model.addConstr(Xvars.sum('*',end_point) == 1)
```

Vincoli di bilancio

$$\sum_{i=1 i \neq j}^{n-1} x_{ij} = \sum_{i=2 i \neq j}^{n} x_{ji} \qquad j = 2..n-1 \tag{14}$$

[ ]:
```
#Vincolo di bilancio degli archi sui nodi intermedi
Balance = model.addConstrs((gp.quicksum(Xvars.sum(i,j) for i in siti if i !=␣
 ↪end_point)
                            == gp.quicksum(Xvars.sum(j,i) for i in siti if i !=␣
 ↪start_point)
                            for j in siti if i != j and j != start_point and j !
 ↪= end_point))
```

Vincoli di apertura di un nodo

$$\sum_{i=2 i \neq j}^{n} x_{ji} = y_j \qquad j = 1..n-1 \tag{15}$$

```
[ ]: #Vincolo di visita
Visited = model.addConstrs((gp.quicksum(Xvars.sum(j,i) for i in siti if i !=␣
 ↪start_point) == Yvars[j]
                           for j in siti if i != j and j != end_point))
```

Vincoli sulla distanza massima

$$\sum_{i=1}^{n} \sum_{j=1 i \neq j}^{n} d_{ij} x_{ij} \leq DMax \tag{16}$$

```
[ ]: #Vincolo sul DMAX (si indica il tempo di visita)
MaxDistanza = model.addConstr((gp.quicksum(Xvars[i,j]*dist[i,j] for i in siti␣
 ↪for j in siti if i != j) <= DMAX))
```

Vincoli di assenza di sottogiri MTZ Per una implementazione più semplice si è deciso di valutare gli MTZ nel seguente modo. Se $x_{i,j} = 1$ allora:

$$u_i + 1 = u_j \qquad i, j = 1..n, i \neq j, j \neq 1 \tag{17}$$

```
[ ]: #Vincoli di sottogiro MTZ
if MTZ == 1:
    #definizione della variabile ui
    Uvars = model.addVars(siti, vtype = GRB.CONTINUOUS, name = 'u')

    #Se Xij = 1 allora uj = ui + 1 (SE E SOLTANTO SE >>)
    Posizione = model.addConstrs((Xvars[i,j] == 1) >> (Uvars[i]+1==Uvars[j])␣
 ↪for i in siti for j in siti
                                 if j != start_point and i != j)
```

Vincoli di assenza di sottogiri 'Lazy Constraints' In alternativa ai vincoli di sottogiro MTZ, si possono aggiungere i **Lazy Constraints** (modificando opportunamente quelli offerti da GUROBI)

$$\sum_{(i,j) \epsilon S} x_{ij} \leq |S| - 1 \qquad S \subset V \qquad 2 \leq |S| \leq |V| - 1 \tag{18}$$

In particolare, rispetto all'implementazione classica, si è spostata l'istruzione per aggiungere i lazy constraints nella funzione subtour(), in quanto occorre aggiungere un vincolo del tipo appena visto,

ogni qualvolta si incontra nella soluzione intera restituita da Gurobi un sottogiro formato da nodi diversi dallo start ed end point.

La principale **differenza** rispetto ai vincoli MTZ è rappresentata dal fatto che i Lazy Constraint permettono di risolvere il modello 'rilassato', ovvero, senza i vincoli di assenza di sottogiri ed aggiungerli solo quando si arriva a trovare una soluzione corrente intera. Ciò permette di considerare un numero notevolmente inferiore di vincoli, che nel caso di MTZ sono in numero polinomiale.

[ ]:
```
#Funzioni per eliminare i subtour (Callback)
def subtourelim(model, where):
    if where == GRB.Callback.MIPSOL:
        # preleva la soluzione corrente
        Xvals = model.cbGetSolution(model._Xvars)
        selected = gp.tuplelist((i,j) for i, j in model._Xvars.keys() if␣
 ↪Xvals[i,j] > 0.5)
        # cerca il ciclo di lunghezza minima nella soluzione
        tour = subtour(selected)

def subtour(edges):
    unvisited = list(siti)
    while unvisited:
        thiscycle = []
        neighbors = unvisited
        while neighbors:
            current = neighbors[0]
            thiscycle.append(current)
            unvisited.remove(current)
            neighbors = [j for i, j in edges.select(current, '*')
                           if j in unvisited]
        #aggiunto vincolo come nella formulazione
        if len(thiscycle) >= 2 and len(thiscycle) <= len(siti)-1:
            model.cbLazy(gp.quicksum(model._Xvars[i,j] for i in thiscycle for j␣
 ↪in thiscycle if i != j )
                         <= len(thiscycle)-1)
```

Scrittura del modello completo su .lp ed esecuzione del solutore

[ ]:
```
if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro" ):
    model.write(str(namefile) + '_orienteering.lp')
else:
    model.write(str(namefile) + '_orienteering.lp')
```

Se l'esecuzione del solutore impiega molto tempo (al max 120s) rimuovere il commento: *#model.Params.MIPGap = 0.30* e scegliere la percentuale di MIPGap di Gurobi (in questo caso del 30%)

Gurobi infatti "misura" l'ottimalità della soluzione attraverso il rapporto tra:

$$MIPGap = \frac{|best_{objective} - best_{bound}|}{best_{objective}} \tag{19}$$

Se non gli viene fornita una percentuale, Gurobi continuerà ad eseguire fino a quando il MIPGap non raggiungerà lo 0.00% e purtroppo essendo che le dimensioni del problema sono rilevanti impiegherà molto tempo per valutare l'ottima soluzione, soprattutto con l'utilizzo degli MTZ.

[ ]:
```
import time
start = time.time() #Start-Esecuzione

if MTZ == 1:
    #model.Params.MIPGap = 0.10 #**50 *30 ***10
    model.optimize()
else:
    model._Xvars = Xvars
    model.Params.lazyConstraints = 1
    model.optimize(subtourelim)

end = time.time()
time_exec = round(end - start,3)
print('Tempo di esecuzione ' + str(time_exec))
```

Stampa dei risultati Il risultato indica il percorso da seguire:

[ ]:
```
#Funzione per costruire il percorso a partire dalla soluzione
def createtour(solution):
    unvisited = list(siti)
    neighbors = unvisited
    tour = []
    while neighbors:
            current = neighbors[0]
            tour.append(current)
            unvisited.remove(current)
            neighbors = [j for i, j in solution.select(current, '*')
                         if j in unvisited]
    return tour
```

[ ]:
```
#Stampa dei risultati
if model.status == GRB.OPTIMAL:

    foundOptimalSol = True
    Xvals = model.getAttr('x', Xvars)
    #ottengo la soluzione
    solution = gp.tuplelist((i,j) for i,j in Xvals.keys() if Xvals[i,j] > 0.5)
    #prelevo il punteggio
    punteggio = model.ObjVal
```

```
    #Stampa del percorso ricavato con le MTZ
    if MTZ == 1:
        Uvals = model.getAttr('x', Uvars)

        optTour=createtour(solution)
        percorso=list(optTour)

        print('Valore di soddisfazione TOTALE - MTZ: %g' % punteggio)

    #Stampa del percorso ricavato con i Lazy Constraints
    else:
        optTour=createtour(solution)
        percorso=list(optTour)

        punteggio=0
        for i in percorso:
            punteggio+=score[i]

        print('Valore di soddisfazione TOTALE - LAZY: %g' % punteggio)

    #Stampa del numero dei Point of Interests
    print("N.POI: %g" % len(percorso))

    #Stampa della distanza totale
    lunghezza=0
    for i in solution:
        lunghezza+=dist[i]
    print('Distanza percorsa in km (lda): ' + str(lunghezza))

    #Stampa del percorso
    print(percorso)
```

Stampa del tour in linea d'aria

```
[ ]: # Stampa del tour degli scavi di Pompei in linea d'aria
import folium
import folium.plugins as plugins
map = folium.Map(location=[lat,long], zoom_start = zoom)

points = []
for sito in percorso:
    points.append(coordinate[sito])
    if sito == start_point: #start point
            folium.Marker(location = coordinate[sito], tooltip = sito,
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
 ↪icon_shape="marker",
                                                    border_color= '#b22222',
```

```
                                                        ␣
↪background_color='#b22222')).add_to(map)
    elif sito == end_point: #end point
            folium.Marker(location = coordinate[sito], tooltip = sito,
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    border_color= '#ffd700',
                                                        ␣
↪background_color='#ffd700')).add_to(map)
    else:
            folium.Marker(location = coordinate[sito], tooltip = sito,␣
↪icon=folium.Icon(color='darkred')).add_to(map)



folium.PolyLine(points, color='darkred').add_to(map)

map
```

[ ]:
```
conv_coord = []
mytour=[]
i=0
for sito in percorso:
    conv_coord.append((coordinate[sito][1],coordinate[sito][0]))
    mytour.append(list(conv_coord[i]))
    i+=1
```

Stampa del percorso a piedi per l'applicazione reale (Scavi di Pompei)

[ ]:
```
#Stampa del percorso per vie REALI
#(ovviamente la distanza in km sarà leggermente diversa da quella in linea␣
↪d'aria)
if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro" ):
        import openrouteservice as ors
        import folium
        import folium.plugins as plugins

        # API Key di Open Route Service
        ors_key = '5b3ce3597851110001cf6248435cfcfbcf0c42858fde19dccf6f9c0f'

        # Richiesta dei servizi tramite API Key di ORS
        # Apro un Client per effettuare le richieste al Server di ORS
        client = ors.Client(key=ors_key)

        # Traccio il percorso
```

```
        route = client.directions(coordinates=mytour,
                                  profile='foot-walking',
                                  format='geojson')


        map = folium.Map(location=[lat,long], zoom_start = zoom)
        #for sito in percorso:
            #folium.Marker(location = coordinate[sito], tooltip = sito).
↪add_to(map)

        num = 0
        for sito in percorso:
            if sito == start_point: #start point
                folium.Marker(location = coordinate[sito], tooltip = sito,
                              icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                        number=num,
                                                        border_color= '#b22222',
                                                       ␣
↪background_color='#b22222')).add_to(map)
            elif sito == end_point: #end point
                folium.Marker(location = coordinate[sito], tooltip = sito,
                              icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                        number=num,
                                                        border_color= '#ffd700',
                                                       ␣
↪background_color='#ffd700')).add_to(map)
            else:
                folium.Marker(location = coordinate[sito], tooltip = sito,
                              icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                        number=num,
                                                        border_color= '#b22222',
                                                       ␣
↪background_color='#ffffff')).add_to(map)
            num+=1

        # Aggiungo il GeoJson alla mappa
        folium.GeoJson(route, name=('Itinerario Scavi di Pompei con ' +␣
↪str(DMAX) + ' ore'),
                       style_function=lambda feature: {'color': 'darkred'}).
↪add_to(map)

        # Aggiungo il livello del percorso alla mappa
        folium.LayerControl().add_to(map)
```

```
        print('Distanza percorsa in km (reale): ' +␣
↪str((route['features'][0]['properties']['summary']['distance'])/1000))

map
```

[ ]:
```
if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro" ):
    model.write(str(namefile) + '_orienteering_sol.lp')
else:
    model.write(str(namefile) + '_orienteering_sol.lp')
```

Salvataggio dei risultati

[ ]:
```
import csv

colonne = ['Istanza','DMAX','Metodo','N.POI','Lunghezza percorso (km)','Valore␣
↪di soddisfazione (TOTALE)','Tempo di esecuzione (s)']
with open('./Confronti.csv', mode='a', newline='') as csv_file:

    writer = csv.DictWriter(csv_file, fieldnames=colonne)

    if MTZ==1:
        writer.writerow({'Istanza': namefile,'DMAX': DMAX,'Metodo': 'OPT␣
↪MTZ','N.POI': len(percorso),'Lunghezza percorso (km)':␣
↪round(lunghezza,5),'Valore di soddisfazione (TOTALE)': punteggio, 'Tempo di␣
↪esecuzione (s)': time_exec})
    else:
        writer.writerow({'Istanza': namefile,'DMAX': DMAX,'Metodo': 'OPT L.C.
↪','N.POI': len(percorso),'Lunghezza percorso (km)':␣
↪round(lunghezza,5),'Valore di soddisfazione (TOTALE)': punteggio, 'Tempo di␣
↪esecuzione (s)': time_exec})
```

GeneticAlgorithm_Orienteering

July 3, 2022

1 Genetic Algorithm per la risoluzione del problema di Orienteering

1.1 Accenno alla metaeuristica Genetic Algorithm

In un algoritmo genetico, una popolazione di **soluzioni candidate (chiamate individui)** a un problema di ottimizzazione evolvono verso soluzioni migliori. Ogni soluzione candidata ha un insieme di proprietà (i suoi cromosomi o genotipo) che possono essere mutate e alterate.

L'evoluzione di solito inizia da una popolazione di individui generati casualmente ed è un processo iterativo, con **la popolazione in ogni iterazione chiamata generazione**. In ogni generazione viene valutata la forma fisica (**fitness**) di ogni individuo della popolazione; l'idoneità è solitamente il valore della funzione obiettivo nel problema di ottimizzazione da risolvere. Gli individui più in forma vengono selezionati stocasticamente dalla popolazione attuale e il genoma di ogni individuo viene **mutato** (in modo casuale o secondo un criterio) per formare una **nuova generazione**. La nuova generazione di soluzioni candidate viene quindi utilizzata nell'iterazione successiva dell'algoritmo. Comunemente, **l'algoritmo termina quando è stato prodotto un numero massimo di generazioni** o quando è stato raggiunto un livello di fitness soddisfacente per la popolazione dove ogni soluzione candidata presenterà la stessa fitness.

Un tipico algoritmo genetico richiede:

- una rappresentazione genetica del dominio della soluzione;
- una funzione di fitness per valutare il dominio della soluzione;
- una rappresentazione standard di ciascuna soluzione candidata è un array di bit (chiamato anche set di bit o stringa di bit).

Una volta definita la rappresentazione genetica e la funzione di fitness, un **GA procede ad inizializzare una popolazione di soluzioni e quindi a migliorarla attraverso l'applicazione ripetitiva degli operatori di mutazione, crossover, inversione e selezione.**

1.1.1 Parametri usati per il Genetic Algorithm

Per l'esecuzione dell'algoritmo di GA, si sono scelti in via sperimentale, i seguenti parametri: - population=100 (dimensione della popolazione); - max_number_of_generation=10 (numero di generazioni create); - K=4 (numero di sottoinsiemi della popolazione creati ad ogni generazione); - tsize=4 (numero di individui generati)

Lettura dati dal .txt Per il test del modello è possibile utilizzare le seguenti istanze Benchmark: - Bench32 - Bench52 - Bench102

In particolare, ogni riga dell'istanza Benchmark (Bench32, Bench52, Bench102) ha questo formato: - x y S

dove: x = x coordinate y = y coordinate S = score

Osservazioni: - La prima riga indica lo *start point* - L'ultima riga indica l' *end point*

Per il test del modello dell'applicazione reale è possibile utilizzare le seguenti istanze create dagli AUTORI: - Scavi_Archeologici_Pompei_Anfiteatro_Misteri - Scavi_Archeologici_Pompei_Marina_Misteri - Scavi_Archeologici_Pompei_Marina_Anfiteatro

N.B. Occorre inserire nella riga "namefile" il nome della relativa istanza *N.B.* Le istanze dell'applicazione reale sono state scelte sulla base di differenti start_point e end_point

[ ]:
```
import pandas as pd
import numpy as np


#Parametri del problema
DMAX=0.03
population=100
max_number_of_generation=10
K=4
tsize=4

namefile = 'Scavi_Archeologici_Pompei_Marina_Anfiteatro'

node_list = []
with open('./IstanzeBenchmark/' + namefile + '.txt') as file:

    if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
        namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
        namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro"):
        zoom=16
        for i, line in enumerate(file):
            node = line.split(',')
            node_list.
 ↪append([str(node[0]),float(node[1]),float(node[2]),int(node[3])])

    else:
        zoom=5
        for i, line in enumerate(file):
            node = line.split(',')
            node_list.
 ↪append([str(i),float(node[0]),float(node[1]),int(node[2])])

node_info = np.asarray(node_list,dtype=object)
node = pd.
 ↪DataFrame(data=node_info,columns=['Nodo','Longitudine','Latitudine','Score'])
```

[ ]:
```
import random

#Definizione della classe individuo e i suoi metodi
class individual:
    """
    Definisco una classe relativi agli individui della popolazione.
    - node_list: Lista dei nodi.
    - visited: Nodi visitati.
    - unvisited: Nodi non visitati.
    - fitness: Fitness dell'individuo.
    - distance: Distanze adiacenti nel percorso.
    - TotalDistance: Distanza totale del percorso.
    - TotalPoint: Punteggio totale del percorso.
    """
    def __init__(self):
        self.node_list=np.array([node.iloc[0],node.iloc[-1]])
        self.visited=self.node_list.copy()
        self.unvisited=np.array(node.iloc[1:-1])
        self.fitness=0
        self.distance=self.cal_distance()
        self.TotalDistance=self.distance.sum()
        self.TotalPoint=0

    def␣
↪reset(self,node_list,visited,unvisited,distance,TotalDistance,TotalPoint):
        self.node_list=node_list
        self.visited=visited
        self.unvisited=unvisited
        self.distance=distance
        self.TotalDistance=TotalDistance
        self.TotalPoint=TotalPoint

    #Calcolo della distanza tra due punti
    def path_length(self,x1,x2,y1,y2):
        x=(x1-x2)**2
        y=(y1-y2)**2
        return np.sqrt(x+y)

    def cal_distance(self):
        distance=[]
        number_of_nodes = self.node_list.shape[0]
        for i in range(number_of_nodes - 1):
            D=self.path_length(self.node_list[i,1],self.node_list[i+1,1],self.
↪node_list[i,2],self.node_list[i+1,2])
            distance.append(D)
        distance=np.array(distance)
        return distance
```

```
    def cal_TotalDistance(self):
        self.TotalDistance=self.distance.sum()

    def cal_TotalPoint(self):
        number_of_nodes = self.node_list.shape[0]
        for i in range(number_of_nodes):
            self.TotalPoint+=self.node_list[i,3]

    #Calcolo della fitness
    #La valutazione è Fitness_i = TotalPoint^3 / TotalDistance
    def cal_fitness(self):
        if self.TotalPoint==0:
            self.fitness=0
        else:
            self.fitness=self.TotalPoint**3/self.TotalDistance

    def insert_repeat_node(self):
        index=random.randint(0,len(self.visited)-1)
        insert_node=self.visited[index,:]
        insert_loc=len(self.node_list)-1
        self.node_list=np.insert(self.node_list,insert_loc,insert_node,axis=0)

        self.distance=self.cal_distance()
        self.cal_TotalDistance()

        self.cal_fitness()
```

[ ]:
```
import copy


#Funzione che calcola la distanza euclidea tra 2 individui
def cal_distant(x1,y1,x2,y2):
    x=(x1-x2)**2
    y=(y1-y2)**2
    return np.sqrt(x+y)

#Funzione di inserimento di un nodo
def␣
 ↪insert_vertex(insert_location,node_list_temp,node_distance_temp,node_visited_temp,node_unvi
 ↪array([-1,-1,-1,-1])):
    delete=False

    #Scelgo casualmente un nodo non sul percorso
    if insert_node[0]==-1:
        insert_node_index = random.randint(0, node_unvisited_temp.shape[0] - 1)
        insert_node=node_unvisited_temp[insert_node_index,:].copy()
```

```
        delete=True
    #Se il nodo è visitato, aggiorno il suo indice con il valore dell'ordine di␣
↪visita
    else:
        i=0
        while i<len(node_unvisited_temp):
            if node_unvisited_temp[i,0]==insert_node[0]:
                insert_node_index=i
                delete=True
                break
            i+=1

    #Inserisco il punto nella posizione specificata
    node_list_temp=np.insert(node_list_temp,insert_location,insert_node,axis=0)

    #Aggiorno il valore di visited/unvisited/totalpoint
    if delete:
        node_visited_temp=np.insert(node_visited_temp,0,insert_node,axis=0)
        node_unvisited_temp=np.
↪delete(node_unvisited_temp,insert_node_index,axis=0)
        node_TotalPoint_temp+=insert_node[3]

    #Aggiorno la distanza
    ␣
↪dis_front=cal_distant(insert_node[1],insert_node[2],node_list_temp[insert_location-1,1],nod
    ␣
↪dis_back=cal_distant(insert_node[1],insert_node[2],node_list_temp[insert_location+1,1],node
    temp_list=np.array([dis_front,dis_back])
    node_distance_temp=np.insert(node_distance_temp,insert_location,temp_list)
    node_distance_temp=np.delete(node_distance_temp,insert_location-1)
    node_TotalDistance_temp=np.sum(node_distance_temp)

    return␣
↪node_list_temp,node_visited_temp,node_unvisited_temp,node_TotalPoint_temp,node_distance_tem


def crossover(parent1,parent2,i,j):
    # Genero un elenco di nodi figli
    child1_node=parent1.node_list[1:i,:]
    child1_node=np.append(child1_node,parent2.node_list[j:-1,:],axis=0)
    child2_node = parent2.node_list[1:j, :]
    child2_node = np.append(child2_node, parent1.node_list[i:-1, :], axis=0)

    #Creo due nuovi individui
    child1=individual()
    child2=individual()
```

```
    child1_node_list, child1_distance,child1_node_visited,␣
↪child1_node_unvisited, child1_TotalPoint, \
    child1_TotalDistance=child1.node_list,child1.distance,child1.visited,child1.
↪unvisited,\
                          child1.TotalPoint,child1.TotalDistance

    child2_node_list, child2_distance, child2_node_visited,␣
↪child2_node_unvisited, child2_TotalPoint, \
    child2_TotalDistance = child2.node_list, child2.distance, child2.visited,␣
↪child2.unvisited, \
                           child2.TotalPoint, child2.TotalDistance

    for x in range(len(child1_node)):
        ␣
↪child1_node_list,child1_node_visited,child1_node_unvisited,child1_TotalPoint,\
        child1_distance,child1_TotalDistance =␣
↪insert_vertex(x+1,child1_node_list, child1_distance,child1_node_visited,
                                                                        ␣
↪child1_node_unvisited, child1_TotalPoint,child1_node[x,:])

    child1.reset( child1_node_list,child1_node_visited,child1_node_unvisited,\
        child1_distance,child1_TotalDistance,child1_TotalPoint)
    child1.cal_fitness()

    for y in range(len(child2_node)):
        ␣
↪child2_node_list,child2_node_visited,child2_node_unvisited,child2_TotalPoint,\
        child2_distance,child2_TotalDistance =␣
↪insert_vertex(y+1,child2_node_list, child2_distance, child2_node_visited,␣
↪child2_node_unvisited,
                                                                        ␣
↪child2_TotalPoint,child2_node[y,:])

    child2.reset(child2_node_list,child2_node_visited,child2_node_unvisited,\
                    child2_distance,child2_TotalDistance,child2_TotalPoint)
    child2.cal_fitness()

    return child1,child2

#TwoOpt per attuare la mutazione
def TwoOpt(old_individual,DMAX):
    length=len(old_individual.node_list)
    if length<=4:
        return old_individual
    s_node_index=random.randint(1,length-3)
    e_node_index=random.randint(2,length-2)
```

```
    while e_node_index<=s_node_index:
        e_node_index=random.randint(2,length-2)

    new_individual=individual()

    #Sottostringa da scambiare
    sub_sequence=copy.deepcopy(old_individual.node_list[s_node_index:
↪e_node_index+1,:])
    new_node_list,  new_distance,new_node_visited, new_node_unvisited,␣
↪new_TotalPoint,\
    new_TotalDistance=new_individual.node_list,new_individual.
↪distance,new_individual.visited,new_individual.unvisited,\
                        new_individual.TotalPoint,new_individual.TotalDistance

    #Il primo e l'ultimo elemento del nuovo individuo sono gli stessi
    for i in range(s_node_index-1):
        insert_node=copy.deepcopy(old_individual.node_list[i+1])
        new_node_list,new_node_visited,  new_node_unvisited,␣
↪new_TotalPoint,new_distance, \
        ␣
↪new_TotalDistance=insert_vertex(i+1,new_node_list,new_distance,new_node_visited,␣
↪ new_node_unvisited, new_TotalPoint, \
                        insert_node)

    #Scambio della prima sequenza
    i=e_node_index-s_node_index
    while i>=0:

        temp_length=len(new_node_list)
        insert_location=temp_length-1
        insert_node=copy.deepcopy(sub_sequence[i,:])
        new_node_list,  new_node_visited, new_node_unvisited, new_TotalPoint,␣
↪new_distance,\
        new_TotalDistance = insert_vertex(insert_location, new_node_list,␣
↪new_distance, new_node_visited, new_node_unvisited,
                                            new_TotalPoint, \
                                            insert_node)
        i-=1

    #Scambio della seconda sequenza
    i=e_node_index+1
    while i<=length-2:
        insert_node=copy.deepcopy(old_individual.node_list[i,:])
        temp_length = len(new_node_list)
        insert_location = temp_length - 1
```

```
        new_node_list,  new_node_visited, new_node_unvisited, new_TotalPoint,␣
↪new_distance,\
        new_TotalDistance = insert_vertex(insert_location, new_node_list,␣
↪new_distance, new_node_visited,
                                          new_node_unvisited,
                                          new_TotalPoint, \
                                          insert_node)
        i+=1

    #Verifico che il nuovo individuo rispetti il vincolo sulla distanza massima
    if new_TotalDistance>DMAX:
        return old_individual

    new_individual.reset(new_node_list,new_node_visited,␣
↪new_node_unvisited,new_distance,new_TotalDistance,new_TotalPoint)
    new_individual.cal_fitness()

    return new_individual


def inserting_mutation(old_solution,DMAX):
    from random import choice
    in_element_index= random.randint(0, old_solution.unvisited.shape[0] - 1)
    in_element=old_solution.unvisited[in_element_index,:].copy()
    in_value=in_element[3]
    loc=0
    best_increase_value=0
    old_solution_TotalDistance=old_solution.TotalDistance
    for i in range(1,len(old_solution.node_list)):
        old_node_list,  old_node_visited, old_node_unvisited,␣
↪old_TotalPoint,old_distance, \
        old_TotalDistance = old_solution.node_list,  old_solution.visited,␣
↪old_solution.unvisited, \
                                old_solution.TotalPoint, old_solution.
↪distance,old_solution.TotalDistance
        old_node_list,old_node_visited,old_node_unvisited,old_TotalPoint,\
        old_distance,old_TotalDistance= insert_vertex(i,old_node_list,␣
↪old_distance, old_node_visited, old_node_unvisited, old_TotalPoint, \
        in_element)
        increase=(old_TotalDistance-old_solution_TotalDistance)
        insertion_value=in_value**2/increase
        if insertion_value>best_increase_value and old_TotalDistance<DMAX:
            best_increase_value=insertion_value
            loc=i
    old_node_list,  old_node_visited, old_node_unvisited,␣
↪old_TotalPoint,old_distance, \
```

```
    old_TotalDistance = old_solution.node_list,  old_solution.visited,␣
↪old_solution.unvisited, \
                        old_solution.TotalPoint, old_solution.
↪distance,old_solution.TotalDistance
    if loc!=0:
        new_node_list, new_node_visited, new_node_unvisited, new_TotalPoint,␣
↪new_distance,\
        new_TotalDistance=insert_vertex(loc,old_node_list, old_distance,␣
↪old_node_visited, old_node_unvisited, old_TotalPoint, \
    in_element)
        new_individual=individual()
        new_individual.reset(new_node_list, new_node_visited,␣
↪new_node_unvisited, new_distance, \
        new_TotalDistance, new_TotalPoint)
        new_individual.cal_fitness()
        return new_individual
    return old_solution


def delete_mutation(old_individual,DMAX):
    #Verifico se ci sono elementi duplicati nell'individuo mutato
    temp_list=old_individual.node_list
    node_index_list=[]

    #Variabili di conteggio
    i=1
    k=0
    flag=0
    repeat_loc1=0
    repeat_loc2=0

    while i<=len(temp_list)-2:
        if k==0:
            node_index_list.append([temp_list[i,0],i])
            k+=1
        else:
            l=0
            for j in node_index_list:
                if temp_list[i,0]==j[0] and l!=k-1:
                    l+=j[1]
                    flag+=1
                    break

        if flag==0:
            node_index_list.append([temp_list[i,0],i])
            k+=1
        else:
```

```
            repeat_loc1+=i
            repeat_loc2+=l
            break

        i+=1

    #Elimina gli elementi duplicati nell'individuo
    if flag==1:
        new_individual1=individual()
        new_individual2=individual()

        new1_node_list, new1_node_visited, new1_node_unvisited,␣
↪new1_TotalPoint, new1_distance, \
        new1_TotalDistance = new_individual1.node_list, new_individual1.
↪visited, new_individual1.unvisited, \
                             new_individual1.TotalPoint, new_individual1.
↪distance, new_individual1.TotalDistance

        new2_node_list, new2_node_visited, new2_node_unvisited,␣
↪new2_TotalPoint, new2_distance, \
        new2_TotalDistance = new_individual2.node_list, new_individual2.
↪visited, new_individual2.unvisited, \
                              new_individual2.TotalPoint, new_individual2.
↪distance, new_individual2.TotalDistance

        #Rimuove la prima posizione duplicata
        m=1
        loc=1
        while m<=len(temp_list)-2:

            if m!=repeat_loc1:
                new1_node_list, new1_node_visited, new1_node_unvisited,␣
↪new1_TotalPoint, new1_distance, \
                new1_TotalDistance = insert_vertex(loc, new1_node_list,␣
↪new1_distance, new1_node_visited, new1_node_unvisited,
                                                   new1_TotalDistance, \
                                                   temp_list[m,:])
                loc+=1
            m+=1

        new_individual1.reset(new1_node_list, new1_node_visited,␣
↪new1_node_unvisited, new1_distance, \
        new1_TotalDistance, new1_TotalPoint)
        new_individual1.cal_fitness()

        #Rimuove la seconda posizione duplicata
```

```
        n=1
        loc=1
        while n <= len(temp_list) - 2:
            if n != repeat_loc2:
                new2_node_list, new2_node_visited, new2_node_unvisited,␣
↪new2_TotalPoint, new2_distance, \
                new2_TotalDistance = insert_vertex(loc, new2_node_list,␣
↪new2_distance, new2_node_visited,
                                                   new2_node_unvisited,
                                                   new2_TotalDistance, \
                                                   temp_list[n, :])
                loc+=1
            n+=1

        new_individual2.reset(new2_node_list, new2_node_visited,␣
↪new2_node_unvisited, new2_distance, \
                              new2_TotalDistance, new2_TotalPoint)
        new_individual2.cal_fitness()

        if new_individual1.TotalDistance<new_individual2.TotalDistance:
            return new_individual1
        else:
            return new_individual2

    else:
        #Memorizzo il risultato
        len_decrease = 0
        delete_location = 0
        if (old_individual.TotalDistance > DMAX):
            for i in range(1, len(old_individual.node_list)):
                old_node_list, old_node_visited, old_node_unvisited,␣
↪old_TotalPoint, old_distance, \
                old_TotalDistance = old_individual.node_list, old_individual.
↪visited, old_individual.unvisited, \
                                    old_individual.TotalPoint, old_individual.
↪distance, old_individual.TotalDistance
                #Elimino la posizione rimossa
                deletion_point = copy.deepcopy(i)
                j = 1
                #Nuovo individuo
                new_individual = individual()
                new_node_list, new_distance, new_node_visited,␣
↪new_node_unvisited, new_TotalPoint, \
                new_TotalDistance = new_individual.node_list, new_individual.
↪distance, new_individual.visited, new_individual.unvisited, \
```

```
                                    new_individual.TotalPoint, new_individual.
↪TotalDistance
                loc = 1
                while j < len(old_individual.node_list - 1):
                    #Inserisco i valori del nuovo individuo
                    insertion_value = old_individual.node_list[j, :].copy()

                    if (j != deletion_point):
                        new_node_list, new_node_visited, new_node_unvisited,␣
↪new_TotalPoint, \
                        new_distance, new_TotalDistance = insert_vertex(loc,␣
↪new_node_list, new_distance,
                                                                        ␣
↪new_node_visited,
                                                                        ␣
↪new_node_unvisited, new_TotalPoint,
                                                                        ␣
↪insertion_value)

                        new_individual.reset(new_node_list, new_node_visited,␣
↪new_node_unvisited, \
                                             new_distance, new_TotalDistance,␣
↪new_TotalPoint)
                        loc += 1
                    j += 1
                decreaseValue = old_TotalDistance - new_TotalDistance

                if decreaseValue!=0:
                    temp_len_decrease = old_individual.
↪node_list[deletion_point, 3] ** 2 / decreaseValue
                else:
                    temp_len_decrease= +999999999999

                if (len_decrease == 0 and temp_len_decrease > 0):
                    len_decrease = copy.deepcopy(temp_len_decrease)
                    delete_location = copy.deepcopy(deletion_point)
                elif (temp_len_decrease > 0 and temp_len_decrease <␣
↪len_decrease):
                    len_decrease = copy.deepcopy(temp_len_decrease)
                    delete_location = copy.deepcopy(deletion_point)

            if (len_decrease != 0 and delete_location != 0):
                res_individual = individual()
                res_node_list, res_distance, res_node_visited,␣
↪res_node_unvisited, res_TotalPoint, \
```

```
                res_TotalDistance = res_individual.node_list, res_individual.
↪distance, res_individual.visited, res_individual.unvisited, \
                                        res_individual.TotalPoint,␣
↪res_individual.TotalDistance
                m = 1
                loc = 1
                while m < len(old_individual.node_list)-1:
                    insertion_value = old_individual.node_list[m, :].copy()

                    if (m != delete_location):
                        res_node_list, res_node_visited, res_node_unvisited,␣
↪res_TotalPoint, \
                        res_distance, res_TotalDistance = insert_vertex(loc,␣
↪res_node_list, res_distance,
                                                                        ␣
↪res_node_visited,
                                                                        ␣
↪res_node_unvisited, res_TotalPoint,
                                                                        ␣
↪insertion_value)
                        res_individual.reset(res_node_list, res_node_visited,␣
↪res_node_unvisited, \
                                                    res_distance,␣
↪res_TotalDistance, res_TotalPoint)
                        loc += 1
                    m += 1

            return res_individual

        else:
            return old_individual
```

1.1.2 Diagramma di flusso del Genetic Algorithm

I passi seguiti per l'esecuzione del **Genetic Algorithm** sono i seguenti:

[ ]:
```
import time


#Inizializzo la popolazione
pop_list=[]

_start = time.time()
for i in range(population):

    temp_node=individual()
```

```
    while True:
        insert_location=temp_node.node_list.shape[0]-1
        node_list_temp=temp_node.node_list.copy()
        node_distance_temp=temp_node.distance.copy()
        node_TotalDistance_temp=temp_node.TotalDistance
        node_visited_temp=temp_node.visited.copy()
        node_unvisited_temp=temp_node.unvisited.copy()
        node_TotalPoint_temp=temp_node.TotalPoint

        node_list_temp, node_visited_temp, node_unvisited_temp,␣
↪node_TotalPoint_temp, node_distance_temp, \
      ␣
↪node_TotalDistance_temp=insert_vertex(insert_location,node_list_temp,node_distance_temp,nod
↪array([-1,-1,-1,-1]))

        if node_TotalDistance_temp>DMAX:
            break

        temp_node.reset(node_list_temp, node_visited_temp,␣
↪node_unvisited_temp,node_distance_temp,node_TotalDistance_temp,node_TotalPoint_temp)
        temp_node.cal_fitness()

    pop_list.append(temp_node)


for count in range(max_number_of_generation):
    #Seleziono alcuni individui della popolazione
    selected_pop_list=[]

    for i in range(int(population/K)):

        for j in range(K):
            #la scelta nella lista è fatta con questa notazione (j*n_population/
↪K:(j+1)*population/K)
            temp_pop_list=pop_list[j*int(population/K):(j+1)*int(population/K)].
↪copy()
            best_fitness=0
            best_individual=None
            index=-1

            for k in range(tsize):

                while True:
                    tem_a=random.randint(0,int(population/K)-1)
                    if index!=tem_a:
                        index=tem_a
```

```
                    break

                selected_individual=copy.deepcopy(temp_pop_list[index])
                if selected_individual.fitness>=best_fitness:
                    best_fitness=selected_individual.fitness
                    best_individual=copy.deepcopy(selected_individual)

            selected_pop_list.append(best_individual)

    pop_list=selected_pop_list.copy()
    crossover_child=[]

    #Operatore di Crossover
    for i in range(int(population/2)):
        parent1_index=random.randint(0,len(pop_list)-1)
        parent2_index=copy.deepcopy(parent1_index)

        #Mi assicuro che i due genitori non siano la stessa persona
        while True:
            a_temp = random.randint(0, len(pop_list)-1)
            if a_temp!=parent2_index:
                parent2_index=a_temp
                break

        parent1=copy.deepcopy(pop_list[parent1_index])
        parent2=copy.deepcopy(pop_list[parent2_index])
        best_individual1=copy.deepcopy(parent1)
        best_individual2=copy.deepcopy(parent2)

        for j in range(1,len(parent1.node_list)-1):

            for k in range(1,len(parent2.node_list)-1):
                if parent1.node_list[j,0]==parent2.node_list[k,0]:
                    child1,child2=crossover(parent1,parent2,j,k)

                    #Sostituisco uno dei due genitori con il figlio1 se␣
↪rispetto il vincolo sulla distanza massima
                    if child1.TotalDistance<DMAX:
                        if best_individual1.fitness<best_individual2.fitness:
                            worse_of_best=copy.deepcopy(best_individual1)
                            tag=1
                        else:
                            worse_of_best=copy.deepcopy(best_individual2)
                            tag=2
                        if child1.fitness>worse_of_best.fitness and tag==1:
                            best_individual1=child1
                        elif child1.fitness>worse_of_best.fitness and tag==2:
```

```
                            best_individual2=child1

                    #Sostituisco uno dei due genitori con il figlio2 se␣
↪rispetto il vincolo sulla distanza massima
                    if child2.TotalDistance<DMAX:
                        if best_individual1.fitness<best_individual2.fitness:
                            worse_of_best=copy.deepcopy(best_individual1)
                            tag=1
                        else:
                            worse_of_best=copy.deepcopy(best_individual2)
                            tag=2
                        if child2.fitness>worse_of_best.fitness and tag==1:
                            best_individual1=child2
                        elif child2.fitness>worse_of_best.fitness and tag==2:
                            best_individual2=child2

        crossover_child.append(best_individual1)
        crossover_child.append(best_individual2)

    pop_list=crossover_child.copy()

    total_select=random.sample(range(0,population),int(population*0.3))

    for i in total_select:

        #Uso 2-opt per effettuare la mutazione del nuovo individuo
        old_individual=copy.deepcopy(pop_list[i])
        new_individual=TwoOpt(old_individual,DMAX)

        #Probabilità di scelta del nuovo individuo
        choice=random.randint(1,2)

        if choice==1:
            new_individual=inserting_mutation(new_individual,DMAX)
        else:
            new_individual=delete_mutation(new_individual,DMAX)

        pop_list[i]=new_individual

best_point=0
best_indi=None
for indi in pop_list:
    if indi.TotalPoint>best_point:
        best_point=indi.TotalPoint
        best_indi=indi

time_exec = round((time.time() - _start),5)
```

```
print('Punteggio: ' + str(best_individual.TotalPoint))
print('Distanza percorsa in km: ' + str(best_individual.TotalDistance))
print('Tempo impiegato: ' + str(time.time() - _start))
print('Percorso: ' + str(best_individual.node_list))

mytour=[]
for i in range(len(best_individual.node_list)):
    mytour.append([best_individual.node_list[i][1], best_individual.
 ↪node_list[i][2]])
```

[ ]:
```
media_lat=0
media_long=0

for node in range(len(node_list)):
    media_lat = media_lat + float(node_list[node][1])
    media_long = media_long + float(node_list[node][2])

lat=media_lat/len(node_info)
long=media_long/len(node_info)
```

[ ]:
```
# Stampa del tour degli scavi di Pompei in linea d'aria
import folium


map = folium.Map(location=[long,lat], zoom_start = zoom)
for i in range(len(best_individual.node_list)):
    folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip =␣
 ↪best_individual.node_list[i][0],
                  icon=folium.Icon(color='darkred')).add_to(map)

points = []
for i in range(len(best_individual.node_list)):
    points.append((mytour[i][1],mytour[i][0]))

folium.PolyLine(points, color='darkred').add_to(map)

map
```

[ ]:
```
#Stampa del percorso per vie REALI
#(ovviamente la distanza in km sarà leggermente diversa da quella in linea␣
 ↪d'aria)
if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro"):
    import openrouteservice as ors
    import folium
    import folium.plugins as plugins
```

```
    # API Key di Open Route Service
    ors_key = '5b3ce3597851110001cf6248435cfcfbcf0c42858fde19dccf6f9c0f'

    # Richiesta dei servizi tramite API Key di ORS
    # Apro un Client per effettuare le richieste al Server di ORS
    client = ors.Client(key=ors_key)

    # Traccio il percorso
    route = client.directions(coordinates=mytour,
                              profile='foot-walking',
                              format='geojson')

    map = folium.Map(location=[long, lat], zoom_start = zoom)
    for i in range(len(best_individual.node_list)):
        if i == 0:
            folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip =␣
↪best_individual.node_list[i][0],
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    number=i,
                                                    border_color= '#b22222',
                                                    ␣
↪background_color='#b22222')).add_to(map)
        elif i == len(best_individual.node_list)-1:
            folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip =␣
↪best_individual.node_list[i][0],
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    number=i,
                                                    border_color= '#ffd700',
                                                    ␣
↪background_color='#ffd700')).add_to(map)
        else:
            folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip =␣
↪best_individual.node_list[i][0],
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    number=i,
                                                    border_color= '#b22222',
                                                    ␣
↪background_color='#ffffff')).add_to(map)

    # Aggiungo il GeoJson alla mappa
```

```
    folium.GeoJson(route, name=('Itinerario Scavi di Pompei con ' + str(DMAX) +␣
 ↪' ore'),
                   style_function=lambda feature: {'color': 'darkred'}).
 ↪add_to(map)

    # Aggiungo il livello del percorso alla mappa
    folium.LayerControl().add_to(map)

    print('Distanza percorsa in km: ' +␣
 ↪str((route['features'][0]['properties']['summary']['distance'])/1000))

map
```

[ ]:
```
import csv


colonne = ['Istanza','DMAX','Metodo','N.POI','Lunghezza percorso (km)','Valore␣
 ↪di soddisfazione (TOTALE)','Tempo di esecuzione (s)']
with open('./Confronti.csv', mode='a', newline='') as csv_file:

    writer = csv.DictWriter(csv_file, fieldnames=colonne)

    writer.writerow({'Istanza': namefile,'DMAX': DMAX,'Metodo': 'G.A.','N.POI':␣
 ↪len(best_individual.node_list),'Lunghezza percorso (km)':␣
 ↪round(best_individual.TotalDistance,5),'Valore di soddisfazione (TOTALE)':␣
 ↪best_individual.TotalPoint, 'Tempo di esecuzione (s)': time_exec})
```

Heuristic_Orienteering

July 3, 2022

1 Euristica per la risoluzione del problema di Orienteering

1.1 Algoritmo Simeuristico (Panadero Juan)

Per applicare un algoritmo euristico, si è trovato un **Algoritmo simeuristico (Juan e Panadero del 2017)** che combina simulazione e metaeuristica per risolvere l'OP, ed in cui il **tuning dei parametri** avviene tramite le osservazioni dell'algoritmo in esecuzione.

1.1.1 Approccio euristico di apprendimento per il problema di Orienteering

L'algoritmo scelto per risolvere il problema di Orienteering usa un approccio di tipo **Biased Randomisation Approach (BRA)**. In generale, questa euristica di apprendimento integra la metaeuristica con la simulazione allo scopo di risolvere problemi di ottimizzazione combinatoria come dei problemi di apprendimento.

L'algoritmo di **Biased Randomisation Heuristic** svolge il ruolo dell'**algoritmo di ottimizzazione**, mentre la **simulazione** si occupa di effettuare il **tuning dei parametri** α **e** β, attraverso un semplice meccanismo di apprendimento dalle osservazioni. Combinando la simulazione con i metodi euristici, l'algoritmo simheuristico risultante mira a estendere e migliorare le capacità dei classici algoritmi di randomizzazione parziale nella modellazione e nella gestione delle dimensioni di un problema.

Le ragioni di tale miglioramento sono le seguenti: - tramite la **parametrizzazione dell'approccio randomizzato** si possono costruire un'**ampia gamma di soluzioni** con caratteristiche diverse fra di loro e quindi, avere una **maggior probabilità di ottenere una soluzione subottima**; - tramite la **procedura greedy** di generazione della soluzione di base si ha la capacità di **ridurre le dimensioni dell'intorno della soluzione da ispezionare**, infatti, si considererà solo l'insieme delle soluzioni non dominate come soluzioni candidate.

[ ]:
```
import functools
import heapq
import collections


#------------------------CLASSI DELL'ALGORITMO--------------------------
class Edge:
    def __init__(self, inode, jnode, cost):
        """
        Definisco una classe relativi agli archi del grafo.
        - inode: Il nodo di partenza.
```

```
        - jnode: Il nodo di terminazione.
        - cost: Distanza tra inode e jnode.
        """
        self.inode = inode
        self.jnode = jnode
        self.cost = cost
        self.savings = dict()


class Route:
    """
    Un'istanza di questa classe rappresenta un percorso dal nodo di
    partenza al nodo di arrivo effettuato da un giocatore.
    """
    def __init__(self, source, depot, starting_node):
        """
        - source: Nodo di partenza del percorso.
        - depot: Nodo di arrivo del percorso.
        - starting_node: Primo nodo incluso nel percorso.

        - nodes: I nodi che formano il percorso.
        - score: Il punteggio totale del percorso.
        - cost: Il costo totale (distanza) del percorso.
        """
        self.source = source
        self.depot = depot
        self.nodes = collections.deque([starting_node])
        self.score = starting_node.score
        self.cost = starting_node.from_source + starting_node.to_depot

    def merge (self, other, edge, dists):
        """
        Questa funzione unisce due percorsi in un solo percorso.

        - other: Percorso alternativo da unire con il percorso attuale.
        - edge: Gli archi usati per unire i due percorsi.
        - dists: La matrice delle distanze tra i nodi.
        """
        # Ottengo il costo e i nodi dell'arco
        inode, jnode = edge.inode, edge.jnode
        edge_cost = dists[inode.id, jnode.id]
        # Aggiorna l'elenco dei nodi nel percorso
        self.nodes.extend(other.nodes)
        # inode non è più connesso al nodo di arrivo
        # e jnode non è più connesso al nodo di partenza
        inode.link_right = False
        jnode.link_left = False
```

```
        # Aggiorna il punteggio e il costo di questo percorso
        self.cost += edge_cost - inode.to_depot + (other.cost - jnode.
↪from_source)
        self.score += other.score
        # Aggiorna il percorso a cui appartengono i nodi
        for node in other.nodes:
            node.route = self


class Node:
    def __init__(self, id, x, y, score, *, color='#FDDD71', issource=False,␣
↪isdepot=False):
        """
        Definisco la classe dei nodi del grafo (ovvero i siti da visitare).
        - id: ID del nodo.
        - x: Longitudine del nodo.
        - y: Latitudine del nodo.
        - score: Lo score relativo alla visita di quel nodo.
        - issource: Una variabile booleana che dice se il nodo è il punto di␣
↪partenza.
        - isdepot: Una variabile booleana che dice se il nodo è il punto di␣
↪arrivo.
        - vehicles: Il numero di veicoli che partono da questo nodo (è 0 se il␣
↪nodo non è la partenza).

                    *** Parametri usati da Mapper ***
        - assigned: Assume valore 1 se il nodo è assegnato ad un nodo sorgente␣
↪e 0 altrimenti.
        - preferences: Utilizzato in caso di nodo di partenza per la ricerca␣
↪greedy dei nodi da collegare.
        - nodes: Utilizzato nel caso del nodo di partenza per conservare i nodi␣
↪ad esso assegnati.

                    *** Parametri usati da PJS (Panadero Juan Savings) ***
        - from_source: La lunghezza del percorso corrente dal nodo di partenza␣
↪a questo nodo.
        - to_depot: La lunghezza del percorso corrente da questo nodo al nodo␣
↪di arrivo.
        - route: Il percorso corrente corrispondente al nodo.
        - link_left: Assume valore 1 se il nodo è collegato al nodo di␣
↪partenza, 0 altrimenti.
        - link_right: Assume valore 1 se il nodo è collegato al nodo di arrivo,␣
↪0 altrimenti.
        """
        self.id = id
        self.x = x
```

```
        self.y = y
        self.score = score
        self.issource = issource
        self.isdepot = isdepot

        # Mapper
        self.assigned = False
        self.preferences = collections.deque()
        self.nodes = collections.deque()

        # PJS
        self.from_source = 0
        self.to_depot = 0
        self.route = None
        self.link_left = False
        self.link_right = False


class Problem:
    """
    Un'istanza di questa classe rappresenta un problema di Orienteering.
    """

    def __init__(self, name, n_nodes, DMAX, sources, nodes, depot):
        """
        - name: Il nome del problema
        - n_nodes: Il numero di nodi.
        - DMAX: La distanza massima percorribile dai giocatori / budget di␣
↪tempo per i percorsi.
        - sources: Il nodo di partenza.
        - nodes: I nodi da visitare.
        - depot: Il nodo di arrivo.

        - dists: La matrice delle distanze tra i nodi.
        - positions: Un dizionario delle posizioni dei nodi.
        - edges: Gli archi che collegano i nodi.
        """
        self.name = name
        self.n_nodes = n_nodes
        self.DMAX = DMAX
        self.sources = sources
        self.nodes = nodes
        self.depot = depot

        # Inizializzo l'elenco degli archi e le posizioni dei nodi
        edges = collections.deque()
        dists = np.zeros((n_nodes, n_nodes))
```

```
        # Calcolo la matrice delle distanze e istanzio gli archi
        # e definisco i colori e le posizioni dei nodi
        source_id = 0
        for node1, node2 in itertools.permutations(itertools.chain(sources,␣
↪nodes, (depot,)), 2):
            # Calcolo il costo dell'arco con la funzione euclidean()
            id1, id2 = node1.id, node2.id
            cost = euclidean(node1, node2)
            # Compilo la matrice orientata delle distanze
            dists[id1, id2] = cost
            # Crea l'arco
            if not node1.isdepot and not node2.issource:
                edges.append(Edge(node1, node2, cost))

        self.dists = dists
        self.edges = edges


    def iternodes (self):
        """
        Un metodo per iterare su tutti i nodi del problema (es. partenza, nodi␣
↪da visitare, arrivo)
        """
        return itertools.chain(self.sources, self.nodes, (self.depot,))
```

[ ]:
```
import random
import math

#-------------------------------ITERATORI-----------------------------------------
def greedy (preferences):
    """
    Questa funzione effettua una ricerca greedy dei nodi da collegare alla␣
↪sorgente
    ordinandoli dal migliore al peggiore, e non appena incontra un nodo che non␣
↪è
    stato ancora assegnato, lo restituisce.

    - preferences: La lista dei nodi da collegare ordinati dal migliore al␣
↪peggiore.
    """
    for _, node in preferences:
        if not node.assigned:
            yield node


def BRA (preferences, beta=0.3):
    """
```

```
    Questo metodo (Biased Randomised Approach) effettua una selezione casuale e
    parziale sull'elenco dei nodi. La selezione si basa su una funzione quasi
    geometrica, di questo tipo:
                          f(x) = (1 - beta) ^ x
    e quindi dà la priorità ai primi elementi dell'elenco.

    - preferences: La lista dei nodi da collegare ordinati dal migliore al␣
 ↪peggiore.
    - beta: Il parametro della distribuzione quasi geometrica.

    Valore di ritorno: L'elemento selezionato ad ogni iterazione.
    """
    L = len(preferences)
    options = list(preferences)
    for _ in range(L):
        idx = int(math.log(random.random(), 1.0 - beta)) % len(options)
        _, node = options.pop(idx)
        if not node.assigned:
            yield node
```

[ ]:
```
import numpy as np
import itertools
import operator


#-------------------------MAPPATURA--------------------------
def _reset_assignment (node):
    """
    Metodo utilizzato per reimpostare l'affiliazione di un nodo a una sorgente.
    """
    node.assigned = False
    return node


def mapper (problem, iterator):
    """
    Questa funzione ricostruisce il percorso dal nodo di partenza al nodo di␣
 ↪arrivo.

    - problem: Istanza dell'Orienteering Problem da risolvere.
    - iterator: Iteratore greedy usato per prelevare i nodi dall'elenco di␣
 ↪visita.

    Valore di ritorno: Un percorso in 2-dimensioni, dove il generico elemento␣
 ↪(i, j) è
    1 se il nodo j è assegnato al nodo sorgente i, 0 altrimenti.
    """
```

```
    # Estrae le caratteristiche del problema
    dists = problem.dists
    sources, nodes, depot = problem.sources, problem.nodes, problem.depot
    n_sources, n_nodes = len(problem.sources), len(problem.nodes)

    # Reimposta l'origine a cui appartengono i nodi
    nodes = tuple(map(_reset_assignment, nodes))

    # Calcola le distanze (euclidee) assolute tra due nodi
    abs_dists = np.array([[dists[s.id, n.id] for n in nodes] for s in sources]).
↪astype("float32")

    # Calcola le distanze marginali
    for i, source in enumerate(sources):
        marginal_dists = abs_dists[i,:] - np.concatenate((abs_dists[:i,:],␣
↪abs_dists[i:,:],), axis=0).min(axis=0)
        source.preferences = iterator(sorted(zip(marginal_dists, nodes),␣
↪key=operator.itemgetter(0)))
        source.nodes = collections.deque()

    # Assegna nodi alle origini
    # Inizializza il numero di nodi già assegnati e la matrice di mappatura
    n_assigned = 0
    mapping = np.zeros((n_sources, n_sources + n_nodes))
    _null_element = object()
    # Finchè ci sono nodi non assegnati
    for source in itertools.islice(itertools.cycle(sources), n_nodes):
        # Considero la lista dei nodi relativi al nodo sorgente attualmente␣
↪considerato
        preferences = source.preferences
        # Prelevo un nodo
        picked_node = next(preferences, _null_element)
        # Se non ci sono più nodi esco
        if picked_node is _null_element:
            break
        # Assegno il nodo alla sorgente
        source.nodes.append(picked_node)
        picked_node.assigned = True
        mapping[source.id, picked_node.id] = 1
        n_assigned += 1

        # Se tutti i nodi sono stati assegnati esco dal ciclo
        if n_assigned == n_nodes:
            break

    # Ritorno il percorso
    return mapping
```

1.1.2 Algoritmo di Biased Randomisation con il calcolo dei Panadero-Juan Savings

Gli algoritmi di **Biased Randomisation** si basano sull'introduzione della *casualità* in un algoritmo greedy. Gli algoritmi costruttivi si basano sulla generazione di soluzioni con un approccio greedy e selezionando in sequenza gli elementi da includere in una soluzione. In questo caso, le soluzioni candidate sono ordinate in un elenco discendente del valore della funzione obiettivo relativa a ciascuna mossa intraprendibile. *Probabilità decrescente* di selezione sono assegnati agli elementi in questo elenco, questa *distribuzione di probabilità* asimmetrica per la selezione degli elementi nella soluzione introduce in un certo senso la casualità in un algoritmo greedy che preserva la logica alla sua base. Questo approccio di generazione di una vasta gamma di soluzioni alternative consentono di migliorare la soluzione generata dalla sola procedura greedy di base. La base costruttiva dell'euristica utilizzata in questo elaborato è composta dalle seguenti fasi:

1. In primo luogo, viene generata una **soluzione fittizia iniziale**. Questa soluzione fittizia iniziale è composta da tutti i percorsi che partono dal nodo di partenza passano per un nodo intermedio per poi andare ad un certo nodo di arrivo (non quello prefissato).

2. Viene selezionata una coppia di percorsi per l'unione. Un'operazione di unione si basa sul tentativo di aggiungere un percorso alla fine di un altro, ovvero si cerca di unire il nodo di arrivo di un percorso con il nodo di partenza dell'altro percorso. Le possibili operazioni di unione sono ordinate in base a una somma ponderata del saving sul costo del viaggio, ovvero la distanza del viaggio ed il punteggio associato ai nodi alle due estremità dell'arco di unione. Il **saving, calcolato secondo il metodo di Panadero-Juan** associato all'arco (i,j) è il seguente:

$$saving_{ij} = \alpha(d_{in} + d_{0j} - d_{ij}) + (1-\alpha)(score_i + score_j) \tag{1}$$

   dove d_{ij} rappresenta la distanza del nodo i dal nodo j. L'equazione per il calcolo del saving fornisce un peso pari ad α al saving sul costo relativo all'inserimento dell'arco (i,j) e un peso di $(1-\alpha)$ per gli score relativi ai nodi appena uniti per formare il percorso totale. Ciò riflette gli obiettivi principali nella risoluzione di un OP, ovvero quelli di generare efficienza (rispettare la distanza massima DMAX) ed allo stesso tempo massimizzare il punteggio totale ottenuto. Una distribuzione di probabilità asimmetrica viene applicata all'elenco ordinato delle operazioni di unione, introducendo così la **"biased randomisation"** nell'algoritmo greedy di base. In particolare, si fa uso di una distribuzione geometrica, la cui equazione è data da:

$$randI = Mod(\lfloor \frac{log(uniRand(0,1))}{log(1-\beta)} \rfloor, |C|) \tag{2}$$

   il valore restituito da questa distribuzione è utilizzato per selezionare casualmente un indice (randI) dall'elenco delle soluzioni candidate, dove $|C|$ è la lunghezza della lista delle soluzioni candidate, uniRand è un input casuale uniforme tra 0 e 1 e β è il parametro che controlla il livello di greedness e casualità in un algoritmo randomizzato. Ogni volta che $\beta = 1$, abbiamo che la scelta è completamente greedy. Al contrario, ogni volta che $\beta = 0$, otteniamo un comportamento totalmente casuale. Quando $0 < \beta < 1$, otteniamo un comportamento misto.

3. Si ripete il passaggio 2., finché non sono più possibili unioni tra due percorsi.

4. Tra i percorsi generati seguendo i passaggi 1. – 3., si seleziona il percorso che massimizza il punteggio totale.

Selezionando $0 < \beta < 1$ e ripetendo i passaggi 1. – 4. per un numero specificato di iterazioni o per un periodo di tempo, l'algoritmo di biased randomisation può generare **soluzioni di buona qualità**, che raggiungono un punteggio totale che si avvicina a quello ottimo.

[ ]:
```
#----------------------------EURISTICA--------------------------------
def _bra (edges, beta):
    """
    Funzione che seleziona gli archi secondo un approccio di tipo Biased␣
 ↪randomised.

    - edges: Lista degli archi.
    - beta: Il parametro della distribuzione quasi geometrica.

    Valore di ritorno: L'arco selezionato.
    """
    L = len(edges)
    options = list(edges)
    for _ in range(L):
        idx = int(math.log(random.random(), 1.0 - beta)) % len(options)
        yield options.pop(idx)


def PJS (problem, source, nodes, depot, beta):
    """
    In questa funzione è implementato l'algoritmo euristico di Panadero Juan␣
 ↪Savings.
    Viene generalmente utilizzato per risolvere un problema di orienteering.

    - problem: L'istanza del problema da risolvere.
    - source: Il nodo di partenza per il quale verrà utilizzato il PJS.
    - nodes: I nodi da visitare.
    - depot: Il nodo di arrivo.
    - beta: Il parametro del biased randomisation (N.B. questo valore è vicino␣
 ↪ad 1 per avere un comportamento greedy)

    Valore di ritorno: Il percorso che farà il giocatore a partire dal nodo di␣
 ↪partenza fino a quello di arrivo.
    """
    # Sposto i valori utili nelle variabili locali
    dists, DMAX = problem.dists, problem.DMAX
    source_id, depot_id = source.id, depot.id

    # Filtro gli archi mantenendo solo quelli che interessano a questo␣
 ↪sottoinsieme di nodi e li ordino
```

```
    sorted_edges = sorted([e for e in problem.edges if e.inode in nodes and e.
↪jnode in nodes], key=lambda edge: edge.savings[source_id], reverse=True)
    # Costruisco una soluzione fittizia in cui un giocatore parte dal nodo di␣
↪partenza e visita
    # un singolo nodo, per poi andare al nodo di arrivo.
    routes = collections.deque()
    for node in nodes:
        # Calcola le distanze dal nodo_intermedio al nodo di arrivo
        # e dal nodo di partenza al nodo_intermedio e le salvo.
        node.from_source = dists[source_id, node.id]
        node.to_depot = dists[node.id, depot_id]
        # Verifico se il nodo può essere visitato secondo il DMAX.
        if node.from_source + node.to_depot > DMAX:
            node.route = None
            continue
        # Alla fine costruisco un nuovo percorso che va dal
        # nodo di partenza ad un nodo_intermedio e dal nodo_intermedio al nodo␣
↪di arrivo.
        route = Route(source, depot, node)
        node.route = route
        node.link_left = True
        node.link_right = True
        routes.append(route)

    # Unisco i percorsi dando priorità agli archi con il saving maggiore
    for edge in _bra(sorted_edges, beta):
        inode, jnode = edge.inode, edge.jnode
        iroute, jroute = inode.route, jnode.route
        # Se l'arco già connette i nodi nel percorso attuale
        # considero l'arco successivo
        if iroute is None or jroute is None or iroute == jroute:
            continue
        # Se inode è l'ultimo nodo del suo percorso e jnode è il primo del suo
        # percorso, l'unione dei due percorsi è possibile.
        if inode.link_right and jnode.link_left:
            # Confronta la lunghezza della nuova rotta con DMAX
            if iroute.cost - inode.to_depot + jroute.cost - jnode.from_source +␣
↪edge.cost <= DMAX:
                # Unisce i due percorsi
                iroute.merge(jroute, edge, dists)
                # Rimuovere il percorso presente in iroute
                routes.remove(jroute)
        # Se ho trovato un percorso esco dal ciclo
        if len(routes) == 1:
            break

    # Restituisco il percorso trovato
```

```
    return sorted(routes, key=operator.attrgetter("score"), reverse=True)[:1]


@functools.lru_cache(maxsize=None)
def PJS_cache (problem, source, nodes, depot, alpha):
    """
    Implementazione memorizzata nella cache del PJS.
    Usata per analizzare il comportamento euristico e deterministico␣
↪dell'algoritmo
    senza il bisogno di esplorare diverse soluzioni.

    - alpha: Il valore alfa utilizzato per calcolare i savings sugli archi
    (utilizzato solo per la memorizzazione nella cache)

    Valore di ritorno: Il percorso che farà il giocatore a partire dal nodo di␣
↪partenza fino a quello di arrivo.
    """
    return PJS(problem, source, nodes, depot, beta=0.999)
    #N.B se beta tende ad 1 vuol dire che la soluzione è data dall'algoritmo␣
↪greedy
    #se invece beta è un valore compreso tra 0 e 1 allora forse potrebbero␣
↪cambiare le cose #OCCCHIOOOOOOOOOOOOOOOOOOOOOO


def multistartPJS (problem, source, nodes, depot, alpha, maxiter, betarange):
    """
    Questo metodo è un'esecuzione multi-start del PJS.
    Ad ogni iterazione, viene generata una nuova soluzione utilizzando un␣
↪parametro beta
    diverso nella selezione degli archi per il processo di unione.

    - maxiter: Il numero massimo di iterazioni.
    - betarange: L'intervallo in cui il beta viene generato casualmente ad ogni␣
↪iterazione.

    Valore di ritorno: La migliore soluzione trovata come insieme di percorsi,␣
↪e il rispettivo punteggio.
    """
    # Genera la soluzione greedy di partenza
    bestroutes = PJS_cache(problem, source, nodes, depot, alpha)
    bestscore = sum(r.score for r in bestroutes)

    # Salva il range in cui varia il beta
    betamin, betamax = betarange

    for _ in range(maxiter):
```

```
        # Genera una nuova soluzione
        routes = PJS(problem, source, nodes, depot, beta=random.
↪uniform(betamin, betamax))
        score = sum(r.score for r in routes)

        # Eventualmente aggiorno il percorso migliore
        if score > bestscore:
            bestroutes, bestscore = routes, score

    # Restituisco la migliore soluzione trovata finora
    return bestroutes, bestscore
```

1.1.3 Diagramma di flusso dell'euristica

Le operazioni **greedy** di base per la generazione delle soluzioni fittizie iniziali, e dunque, della migliore soluzione iniziale sono le seguenti:

Nel seguente diagramma viene mostrato il flowchart delle operazioni effettuate dall'**euristica multistart** per calcolare il percorso con punteggio totale migliore:

[ ]:
```
#------------------------------RISOLUTORE--------------------------------
def set_savings (problem, alpha=0.3):
    """
    Questo metodo calcola il saving degli archi in base all'alfa dato.

    NOTA: gli archi vengono modificati sul posto.

    - problem: L'istanza del problema da risolvere.
    - alpha: Il parametro alfa del PJS.

    Valore di ritorno: L'istanza del problema modificata sul posto.
    """
    dists, depot = problem.dists, problem.depot
    for edge in problem.edges:
        cost, inode, jnode = edge.cost, edge.inode, edge.jnode
        score = inode.score + jnode.score
        edge.savings = {
            source.id : alpha*(dists[inode.id, depot.id] + dists[source.id,␣
↪jnode.id] - cost) + (1.0-alpha)*score
        for source in problem.sources}
    return problem


def alpha_optimisation (problem, alpha_range=np.arange(0.0, 1.1, 0.1)):
    """
    Questo metodo viene utilizzato per ottimizzare il parametro alfa.
```

```
    Il parametro alfa viene utilizzato nel calcolo dei savings sugli archi:

        saving = alfa*distanza_saving + (1-alfa)*punteggio

    Più basso è alfa, maggiore è l'importanza del punteggio,
    più alto è alfa, maggiore è l'importanza della distanza di saving.

    In pratica eseguiamo 10 esecuzioni deterministiche dell'algoritmo
    (cioè, Mapper e poi PJS) per 10 diversi livelli di alfa.
    Il valore dell'alfa che fornisce la migliore soluzione deterministica
    viene salvato come il migliore alfa.

    NOTA: Questo metodo modifica anche il saving degli archi.

    - problem: L'istanza del problema da risolvere.
    - alpha_range: Il range di alfa da testare.

    Valore di ritorno: Il miglior valore ottenuto per l'alfa.
    """
    # Sposto i valori utili nelle variabili locali
    dists, depot, sources, nodes = problem.dists, problem.depot, problem.
↪sources, problem.nodes

    # Eseguo una volta il mapper deterministico
    mapping = mapper(problem, iterator=greedy)

    # Inizializzo il migliore alfa a zero
    best_alpha, best_score = 0.0, float("-inf")

    # Proviamo diversi valori del parametro alfa e manteniamo il migliore
    for alphatest in alpha_range:
        # Provo un nuovo valore di alfa
        alphatest = round(alphatest, 1)
        # Calcolo i risparmi sugli archi in base al nuovo valore di alfa
        set_savings(problem, alphatest)
        # Eseguo una versione deterministica dell'algoritmo PJS per ciascuna␣
↪sorgente.
        routes = []
        for source in problem.sources:
            partial_routes = PJS_cache(problem, source, tuple(source.nodes),␣
↪depot, alphatest)
            routes.extend(partial_routes)
        # Punteggio totale ottenuto (vale a dire, qualità della soluzione)
        total_score = sum(r.score for r in routes)
        # Eventualmente aggiorno il valore dell'alfa
        if total_score > best_score:
            best_alpha, best_score = alphatest, total_score
```

```
    # Imposta il saving degli archi utilizzando l'alfa migliore trovato
    set_savings(problem, best_alpha)
    # Restituisce il miglior alfa ottenuto
    return best_alpha


def heuristic (problem, iterator, alpha):
    """
    Questa è l'esecuzione principale del risolutore.
    Può essere deterministico o stocastico a seconda dell'iteratore
    passato come argomento. (ad esempio, greedy e Biased Randomised Approach)

    - problem: L'istanza del problema da risolvere.
    - iterator: L'iteratore da passare al mapper.
    - alpha: Il valore alfa utilizzato per calcolare i savings sugli archi␣
 ↪(utilizzato solo per la memorizzazione nella cache)

    Valore di ritorno: La soluzione come insieme di percorsi, il loro punteggio␣
 ↪totale, e la matrice di mappatura.
    """
    # Mapper
    mapping = mapper(problem, iterator)
    # PJS per calcolare i percorsi
    routes = []
    for source in problem.sources:
        r = PJS_cache(problem, source, tuple(source.nodes), problem.depot,␣
 ↪alpha)
        routes.extend(r)
    # Calcolo il punteggio totale
    score = sum(r.score for r in routes)
    # Restituisco la mappatura, i percorsi e il punteggio
    return score, mapping, tuple(routes)


#OCCHIOO A MAXITER POTREBBE MIGLIORARE LA SOLUZIONE?????
def multistart (problem, alpha, maxiter=1000, betarange=(0.1, 0.3)):
    """
    Questa è l'esecuzione multistart dell'algoritmo PJS.
    Ad ogni iterazione viene generata una nuova soluzione introducendo
    lievi modifiche attraverso un approccio biased randomised.
    La nuova soluzione generata viene confrontata con la migliore e
    eventualmente sostituita (se il punteggio ottenuto è maggiore).

    - problem: L'istanza del problema da risolvere.
    - alpha: Il valore alfa utilizzato per calcolare i savings sugli archi␣
 ↪(utilizzato solo per la memorizzazione nella cache)
    - maxiter: Il numero massimo di iterazioni e diverse mappature testate.
```

```
    - betarange: L'intervallo del parametro beta da utilizzare nel biased␣
↪randomisation.

    Valore di ritorno: La migliore soluzione trovata finora con la rispettiva␣
↪mappatura e punteggio.
    """
    # Verifico i valori forniti per il parametro beta
    if betarange[0] > betarange[1]:
        raise Exception("Min beta should be higher than max beta.")

    # Salva l'intervallo di beta
    minbeta, maxbeta = betarange

    # Inizializzo la soluzione di partenza come quella greedy
    bscore, bmapping, broutes = heuristic(problem, iterator=greedy, alpha=alpha)

    # Ricerca locale ripetuta
    for i in range(maxiter):

        # Inizializzo l'iteratore biased randomised
        _bra = functools.partial(BRA, beta=random.uniform(minbeta, maxbeta))

        # Genero una nuova soluzione
        score, mapping, routes = heuristic(problem, iterator=_bra, alpha=alpha)

        # Eventualmente aggiorno il migliore
        if score > bscore:
            bscore, bmapping, broutes = score, mapping, routes

    # Restituisco la migliore soluzione trovata finora
    return bscore, bmapping, broutes
```

[ ]:
```
import os
import networkx as nx


#----------------------FUNZIONI DI UTILITA'---------------------------
def euclidean (inode, jnode):
    """
    Questo metodo calcola la distanza euclidea tra due nodi.

    - inode: Primo nodo.
    - jnode: Secondo nodo.
    """
    return math.sqrt((inode.x - jnode.x)**2 + (inode.y - jnode.y)**2)
```

```
def read_problem (path,filename, n_nodes, DMAX):
    """
    Questo metodo viene utilizzato per leggere un problema di Orienteering da un file .txt
    e restituisce un'istanza di problema standard.

    - path: Il percorso in cui si trova il file.
    - filename: Il nome del file da leggere.
    - n_nodes: Il numero di nodi del problema
    - DMAX: Il valore di DMAX del problema

    Valore di ritorno: L'istanza del problema.
    """
    with open(path + filename + '.txt', 'r') as file:
        # inizializzo la lista dei nodi
        sources, nodes, depot = [], [], None
        # leggo l'istanza
        if (filename == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
            filename == "Scavi_Archeologici_Pompei_Marina_Misteri" or
            filename == "Scavi_Archeologici_Pompei_Marina_Anfiteatro"):
            for i, line in enumerate(file):
                node_info = line.split(',')
                if i == 0:
                    # la prima riga indicherà il nodo sorgente (start_point)
                    sources.append(Node(i, float(node_info[1]), float(node_info[2]), int(node_info[3]),
                                        issource=True))
                elif i == n_nodes - 1:
                    # l'ultima riga indicherà il nodo destinazione (end_point)
                    depot = Node(i, float(node_info[1]), float(node_info[2]), int(node_info[3]), isdepot=True)
                else:
                    # aggiunta dei nodi
                    nodes.append(Node(i, float(node_info[1]), float(node_info[2]), int(node_info[3])))
        else:
            for i, line in enumerate(file):
                node_info = line.split(',')
                if i == 0:
                    # la prima riga indicherà il nodo sorgente (start_point)
                    sources.append(Node(i, float(node_info[0]), float(node_info[1]), int(node_info[2]),
                                        issource=True))
                elif i == n_nodes - 1:
                    # l'ultima riga indicherà il nodo destinazione (end_point)
                    depot = Node(i, float(node_info[0]), float(node_info[1]), int(node_info[2]), isdepot=True)
```

```
                else:
                    #  aggiunta dei nodi
                    nodes.append(Node(i, float(node_info[0]),␣
↪float(node_info[1]), int(node_info[2])))


        return Problem(filename, n_nodes, DMAX, tuple(sources), tuple(nodes),␣
↪depot)
```

Lettura dati dal .txt Per il test del modello è possibile utilizzare le seguenti istanze Benchmark: - Bench32 - Bench52 - Bench102

In particolare, ogni riga dell'istanza Benchmark (Bench32, Bench52, Bench102) ha questo formato: - x y S

dove: x = x coordinate y = y coordinate S = score

Osservazioni: - La prima riga indica lo *start point* - L'ultima riga indica l' *end point*

Per il test del modello dell'applicazione reale è possibile utilizzare le seguenti istanze create dagli AUTORI: - Scavi_Archeologici_Pompei_Anfiteatro_Misteri - Scavi_Archeologici_Pompei_Marina_Misteri - Scavi_Archeologici_Pompei_Marina_Anfiteatro

N.B. Occorre inserire nella riga "namefile" il nome della relativa istanza *N.B.* Le istanze dell'applicazione reale sono state scelte sulla base di differenti start_point e end_point

[ ]:
```
import time


#----------------------------MAIN-------------------------------
namefile = 'Bench102'
path = './IstanzeBenchmark/'

with open(path + namefile + '.txt', 'r') as file:
        # lettura dei parametri
        n_nodes = len(file.readlines())
        DMAX = 70

if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro"):
    zoom = 16
else:
    zoom= 4

problem = read_problem(path,namefile, n_nodes, DMAX)
alpha = alpha_optimisation(problem)
set_savings(problem, alpha=alpha)
```

```
#Stampa dell'alpha ottimizzato
print("alpha = " + str(alpha))

_start_h = time.time()
score_h, mapping_h, routes_h = heuristic(problem, greedy, alpha)
time_exec_h = round(time.time() - _start_h,3)
#Stampa dei risultati euristici
print("Heuristic --> Tempo: " + str(time_exec_h) + " Punteggio: " +␣
 ↪str(score_h))


_start_m = time.time()
score_m, mapping_m, routes_m = multistart(problem, alpha, maxiter=1000,␣
 ↪betarange=(0.1, 0.3))
time_exec_m = round(time.time() - _start_m,3)
print("Multistart --> Tempo: " + str(time_exec_m) + " Punteggio: " +␣
 ↪str(score_m))
```

[ ]:
```
#----------------------------Salvataggio del percorso e calcolo della distanza␣
 ↪totale-----------------
pos={}
for node in problem.iternodes():
        # Prelevo le coordinate dei nodi
        pos[node.id] = (node.x, node.y)

# Salvo il percorso
edges = []
for r in routes_h:
    # NOTA: i nodi del percorso dovrebbero essere sempre nell'ordine in cui
    # sono memorizzati all'interno del deque.
    nodes = tuple(r.nodes)
    edges.append((r.source.id, nodes[0].id))
    for n1, n2 in zip(nodes[:-1], nodes[1:]):
        edges.append((n1.id, n2.id))
    edges.append((nodes[-1].id, r.depot.id))

sortedpos=dict(edges)
mytour = []
distanza = 0
for i in sortedpos:
    distanza += problem.dists[i][sortedpos[i]]
    mytour.append([pos[i][0],pos[i][1]])

mytour.append([pos[problem.n_nodes-1][0],pos[problem.n_nodes-1][1]])

print('Distanza percorsa in km: ' + str(distanza))
```

Calcolo del punto medio tra quelli dati per rappresentarli sulla mappa Utile per la visualizzazione ottimale della mappa

[ ]:
```
media_lat=0
media_long=0

for node in range(len(mytour)):
    media_lat = media_lat + float(mytour[node][0])
    media_long = media_long + float(mytour[node][1])

lat=media_lat/len(mytour)
long=media_long/len(mytour)
```

[ ]:
```
# Stampa del tour in linea d'aria
import folium
map = folium.Map(location=[long,lat], zoom_start = zoom)
for i in range(len(mytour)):
    folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip = i,
                  icon=folium.Icon(color='darkred')).add_to(map)

points = []
for i in range(len(mytour)):
    points.append((mytour[i][1],mytour[i][0]))

folium.PolyLine(points, color='darkred').add_to(map)

map
```

[ ]:
```
#Stampa del percorso per vie REALI
#(ovviamente la distanza in km sarà leggermente diversa da quella in linea␣
 ↪d'aria)
if (namefile == "Scavi_Archeologici_Pompei_Anfiteatro_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Misteri" or
    namefile == "Scavi_Archeologici_Pompei_Marina_Anfiteatro"):
    import openrouteservice as ors
    import folium
    import folium.plugins as plugins

    # API Key di Open Route Service
    ors_key = '5b3ce3597851110001cf6248435cfcfbcf0c42858fde19dccf6f9c0f'

    # Richiesta dei servizi tramite API Key di ORS
    # Apro un Client per effettuare le richieste al Server di ORS
    client = ors.Client(key=ors_key)

    # Traccio il percorso
    route = client.directions(coordinates=mytour,
```

```
                                profile='foot-walking',
                                format='geojson')

    #Istanzio la mappa del problema
    map = folium.Map(location=[long,lat], zoom_start = 16)

    for i in range(len(sortedpos)+1):
        if i == 0: #start point
            folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip = i,
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    number=i,
                                                    border_color= '#b22222',
                                                    ␣
↪background_color='#b22222')).add_to(map)
        elif i == len(sortedpos): #end point
            folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip = i,
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    number=i,
                                                    border_color= '#ffd700',
                                                    ␣
↪background_color='#ffd700')).add_to(map)
        else:
            folium.Marker(location = (mytour[i][1],mytour[i][0]), tooltip = i,
                          icon=plugins.BeautifyIcon(icon="arrow-down",␣
↪icon_shape="marker",
                                                    number=i,
                                                    border_color= '#b22222',
                                                    ␣
↪background_color='#ffffff')).add_to(map)

    # Aggiungo il GeoJson alla mappa
    folium.GeoJson(route, name=('Itinerario Scavi di Pompei con ' + str(DMAX) +␣
↪' ore'),
                   style_function=lambda feature: {'color': 'darkred'}).
↪add_to(map)

    # Aggiungo il livello del percorso alla mappa
    folium.LayerControl().add_to(map)

    print('Distanza percorsa in km: ' +␣
↪str((route['features'][0]['properties']['summary']['distance'])/1000))

map
```

[ ]:
```
import csv

colonne = ['Istanza','DMAX','Metodo','N.POI','Lunghezza percorso (km)','Valore␣
 ↪di soddisfazione (TOTALE)','Tempo di esecuzione (s)']
with open('./Confronti.csv', mode='a', newline='') as csv_file:

    writer = csv.DictWriter(csv_file, fieldnames=colonne)

    writer.writerow({'Istanza': namefile,'DMAX': problem.DMAX,'Metodo':␣
 ↪'Heuristic','N.POI': len(sortedpos)+1,'Lunghezza percorso (km)':␣
 ↪round(distanza,5),'Valore di soddisfazione (TOTALE)': score_h, 'Tempo di␣
 ↪esecuzione (s)': time_exec_h})
    writer.writerow({'Istanza': namefile,'DMAX': problem.DMAX,'Metodo':␣
 ↪'Multistart','N.POI': len(sortedpos)+1,'Lunghezza percorso (km)':␣
 ↪round(distanza,5),'Valore di soddisfazione (TOTALE)': score_m, 'Tempo di␣
 ↪esecuzione (s)': time_exec_m})
```