Utilizzo di ACO per il Traveling Salesman Problem

Marco Colavita

March 2020

Contents

Chapter 1

Introduzione

Scopo di questa relazione è quello di spiegare come risolvere il *problema del commesso viaggiatore* (TSP) utilizzando come base di soluzione l' *algoritmo delle colonie di formiche* (ACO), un algoritmo per la ricerca di percorsi ottimali su grafi.

Chapter 2

Ant colony optimization

2.1 Origine

L'idea di proporre questo tipo di algoritmo nasce dallo studio e dall'osservazione delle formiche, specialmente dal loro metodo di procurarsi del cibo da portare nel formicaio.
Collettivamente, queste ultime sono capaci di trovare il percorso più breve per arrivare ad una fonte di cibo.
Il loro mezzo di comunicazione è il loro formicaio. Per inviare dei segnali, o meglio, comunicare con altri individui della propria specie, le formiche utilizzano il feromone. Le informazioni si scambiano in ambito locale, infati una formica ha accesso al feromone solo se si trova nel luogo dove questo è stato depositato.
Lo scopo principale dell'algortimo è quello di trovare il percorso migliore per arrivare alla fonte di cibo.

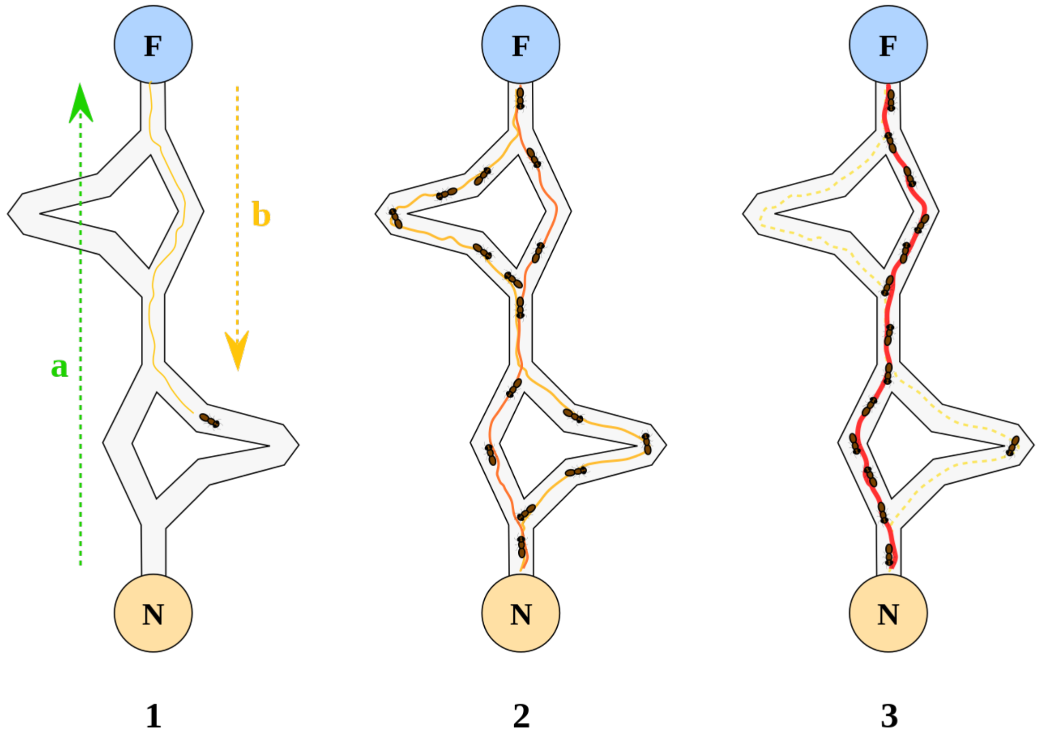

Figure 2.1: 1. La prima formica trova la fonte di cibo (F) tramite una percorso qualsiasi (a) e ritorna al formicaio (N) lasciandosi dietro una scia di feromone. 2. Successivamente le altre formiche percorrono quattro percorsi possibili, ma il rafforzamento della pista rende più attraente il percorso più breve. 3. Le formiche percorrono il percorso più breve, le lunghe porzioni di altri percorsi perdono la loro scia di feromone.

2.2 Descrizione generale: Ant system

Parte degli algortimi sono ora tutti raggruppati sotto il nome di *Ant Colony Optimization*. Il primo algoritmo delle colonie delle formiche proprosto è l'*Ant System*. L'algoritmo generale è basato su un insieme di formiche, ciascuna delle quali attraversa un percorso tra quelli possibili. In ogni fase, la formica sceglie di spostarsi da un nodo all'altro secondo alcune regole:

- più un nodo è distante, meno possibilità ha di essere scelto;

- più l'intensità del percoro del feromone situato sul crinale tra due nodi è maggiore, più possibilità ha di essere scelto;
- una volta completato il suo percorso, la formica deposita, su tutti i bordi attraversati, più feromone se il percorso è breve;
- i percorsi di feromone evaporano ad ogni iterazione.

2.3 Descrizione formale

La regola di spostamento, chiamata *Regola casuale di transizione proporzionale* è matematicamente scritta:

$$p_{ij}^k(t) = \begin{cases} \dfrac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_{l \in J_i^k} \tau_{il}^\alpha \cdot \eta_{il}^\beta}, con j \in J_i^k & (2.1) \\ 0, con j \notin J_i^k & (2.2) \end{cases}$$

dove ${J_i}^k$ è l'elenco dei possibili spostamenti di una formica k quando si trova in un nodo i, η_{ij}, la visibilità e τ_{ij}(t) l'intensità della pista ad una data iterazione t. I due parametri principali che controllano l'algoritmo sono α e β, che controllano l'importanza relativa dell'intensità e della visibilità di un bordo. Una volta fatto il giro di tutti i nodi, una formica k deposita una quantità di feromone $\Delta {t_{ij}}^k$ di feromone su ogni bordo del suo percorso:

$$\Delta t_{ij}^k(t) = \begin{cases} \dfrac{Q}{L^k(t)}, con (i,j) \in T^k(t) & (2.3) \\ 0, con (i,j) \notin T^k(t) & (2.4) \end{cases}$$

dove $T^k(t)$ è il giro fatto dalla formica all'iterazione t, $L^k(t)$ la lunghezza del percorso e Q un parametro di regolazione. Alla fine di ogni iterazione dell'algoritmo, il feromone depositato nelle precedenti iterazioni delle formiche evaporano:

$$\rho \tau_{ij}(t) \tag{2.5}$$

e alla fine dell'iterazione si avrà la quantità di feromone che non è evaporato e di quello che verrà depositato:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta \tau_{ij}^k(t) \tag{2.6}$$

dove m è il numero di formiche utilizzate per l'iterazione t e ρ un parametro di regolazione.

2.4 Applicazioni

Con il passare del tempo questi tipi di algoritmi sono stati applicati a molti problemi di ottimizzazione combinatoria. Nel caso in cui il grafo studiato può cambiare durante l'esecuzione: la colonia di formiche si adatterà in modo relativamente flessibile al cambiamento. L'Ant Colony Optimization è un'euristica il cui punto di forza consiste nell'essere un sistema intelligente distribuito. Ciò significa che le scelte adottate non sono prese da un'unica intelligenza che lavora al problema, ma da una colonia che pensa autonomamente e grazie ai feromoni riesce a condividere le soluzioni appena vengono trovate, adattando dinamicamente il processo di ottimizzazione.

Chapter 3

PLI: ottimizzazione

Si deve definire a questo punto il concetto di base, ovvero quello di ottimizzazione. Si parla di ottimizzazione in *ricerca operativa*, più precisamente nella sezione riguardante la *programmazione lineare intera* (PLI). I problemi di Ricerca Operativa sono dei problemi decisionali di ottimizzazione di un obiettivo in presenza di risorse limitate. Per ogni problema viene costruito un modello matematico che lo definisce e ne risolve il modello. Un modello matematico di ottimizzazione costituisce un problema di massimo o di minimo. La definizione di problema di programmazione lineare intera è la seguente:

Definizione 1. *Un problema di programmazione lineare intera consiste nel trovare il minimo (o il massimo) di una funzione lineare su una regione definita da vincoli lineari e da vincoli di interezza sulle variabili e supponendo che i dati dei problemi siano numeri interi. Un problema di PLI può essere di questo tipo:*

$$\begin{cases} max c^T x & (3.1) \\ Ax \leq b & (3.2) \\ x \in \mathbf{Z}^n & (3.3) \end{cases}$$

dove A è una matrice mxn a componenti intere, $b \in \mathbf{Z}^{\ m}$ e $c \in \mathbf{Z}^n$. I problemi dove le variabili sono vincolate ad avere 0 oppure 1 si chiamano problemi di ottimizzazione combinatorica.

La funzione citata nella definizione è una funzione lineare

$$f : \mathbf{R}^n \to \mathbf{R} \tag{3.4}$$

Chapter 4

TSP

4.1 Definizione

Il problema del commesso viaggiatore, o TSP è uno dei più famosi problemi dell'informatica. Il problema è così definito: dato un insieme definito di città C e siano note le distanze tra ciascuna coppia di esse, il commesso viaggiatore deve trovare il tragitto di minima percorrenza per poter visitare una ed una sola volta ogni città dell'insieme C. Purtroppo questo problema fa parte della classe di algortimi NP-completi, ovvero la sua soluzione richiede un tempo più che polinomiale. Tuttavia, sotto opportune ipotesi, ci sono algoritmi efficienti che forniscono una distanza complessiva che non è tanto differente da quella minima. Il problema del commesso viaggiatore può essere modellato nel seguente modo:

Definizione 2. *Consideriamo un grafo orientato completo G= (N,A), in cui sia definito un costo c_{ij} per ogni arco (i,j) $\in$ A. Un ciclo orientato che passa per tutti i nodi del grafo una ed una sola volta è detto ciclo hamiltoniano e il suo costo è definito come la somma dei costi degli archi da cui è formato. Il problema del commesso viaggiatore coniste nel trovare il ciclo hamiltoniano di costo minimo. Se il grafo non è orientato il problema viene chiamato TSP simmetrico dove per ogni arco (i,j) $\in$ A si ha che $c_{ij}= c_{ji}$; se invece il grafo è orientato si parla di TSP asimmetrico.*

Per il problema del commesso viaggiatore verrà definito solo il caso asimmetrico, dove il grafo è orientato, poichè il viaggiatore potrà visitare una data città dell'insieme C una ed una sola volta. Un ciclo hamiltoniano C è rappresentato mediante le variabile binarie x_{ij} dove

$$x_{ij} = \begin{cases} 1, se(i,j) \in C & (4.1) \\ 0, se(i,j) \notin C & (4.2) \end{cases}$$

4.2 Modello matematico

Il modello matematico sul quale si baserà la risoluzione del TSP asimmetrico potrebbe essere il seguente:

$$\begin{cases} min \sum_{ij \in A} c_{ij} x_{ij} & (4.3) \\ \sum_{i \in N, i \neq j} x_{ij} = 1, \forall j \in N & (4.4) \\ \sum_{j \in N, j \neq i} x_{ij} = 1, \forall i \in N & (4.5) \\ \sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1, \forall S \subset N, S \neq \emptyset & (4.6) \end{cases}$$

La funzione obiettivo è la somma dei costi degli archi selezionati. Il primo insieme di vincoli stabilisce che il ciclo hamiltoniano debba avere un solo arco entrante per ogni nodo $j \in N$; mentre, in modo analogo, il secondo insieme di vincoli stabilisce che il ciclo debba avere un solo arco uscente per ogni nodo $i \in N$. Il terzo insieme di vincoli serve per garantire che nel ciclo hamiltoniano non ci siano dei cicli orientati che non passano in tutti i nodi, quindi che ci sia almeno un arco uscente da ogni sottoinsieme non vuoto di S di nodi. Tale modello però non può essere utilizzato direttamente, a meno che il numero dei nodi, ovvero di città, da visitare sia estremamente piccolo. Sono state definite diverse soluzioni per risolvere il problema del commesso viaggiatore, tra queste vi è presente anche quella utilizzando l'*Ant System*.

4.3 ACO per TSP

L'Ant System o generalizzato ACO con il corso del tempo è stato impiegato per la risoluzione di diversi problemi di ottimizzazione combinatoria, tra cui il problema del commesso viaggiatore. Tale algoritmo si è dimostrato efficiente ed ha le seguenti caratteristiche:

- E' *versatile*, infatti esso può essere applicato a versioni simili dello stesso problema, come appunto il TSP simmetrico e TSP asimmetrico.
- E' *robusto*, può essere applicato con modifiche minime ad altri problemi di ottimizzazione combinatoria come ad esempio il *job scheduling problem* (*JSP*)
- E' un approccio *population based.*

Il TSP può essere risolto con l'ausilio del Ant System, facendo uso delle informazioni che ci dà quest'ultimo, come ad esempio il feromone che deposita ogni formica nel proprio tragitto ed un valore euristo η sopra citato.
Prendendo come banco di lavoro la definizione di TSP:

- Le formiche costruiscono una soluzione seguendo un cammino nel grafo delle città

- Una regola di transizione viene utilizzata per scegliere la prossima città da visitare

- Vengono aggiornati sia l'eurista che il feromone, dove i valori di quest'ultimi vengono aggiornati in base alla qualità della soluzione trovata dalle formiche

il problema parte da una qualsiasi città i del grafo delle città e la scelta probabilista della prossima città da visitare viene calcolata con la formula (2.1) sopra citata. Il feromone invece viene aggiornato con le formule (2.6) e (2.3) del capitolo 2.
L'algoritmo richiede un accurato tuning dei parametri, che influiscono pesantemente sulle efficienza ed efficacia di quest'ultimo.

Chapter 5

Descrizione del problema a livello algoritmico

5.1 Introduzione a Python

In questo capitolo tratteremo le varie scelte implementative utilizzate per risolvere il TSP con ACO.
Python è un linguaggio di programmazione di scripting ad oggetti. Come caratteristica principale permette la "*tipizzazione dinamica*" (dynamic typing), cioè:

- riconosce automaticamente oggetti quali numeri stringhe, liste, etc..., e quindi non richiede di dichiararne il tipo e la dimensione prima dell'utlizzo;

- effettua in modo automatico l'allocazione e la gestione della memoria.

Queste caratteristiche contribuiscono in modo davvere sostanziale a velocizzare la prototipazione di diversi algoritmi.
Incorpora al proprio interno dei moduli. I moduli sono dei file dove è possibile porre definizione e usarle in uno script o in una sessione interattiva. Le definizioni presenti in un modulo possono essere *importate* in altri moduli o entro il modulo main. Nello sviluppo dell'algoritmo sono stati importati diversi moduli di supporto che sono:

- **NumPy**: E' uno dei package fondamentali per il calcolo numerico, permette la creazione di array N-dimensionali utilizzati come contenitori di dati generici per la risoluzione di sistemi lineari.

- **MathPlotlib**: E' una libreria di plotting 2D che produce figure di qualità e un ambiente interattivo per una valutazione accurata di ciò che viene plottato.

- **Scipy**: Fornisce molte routine numeriche amichevoli ed efficienti, come la routine per l'integrazione numeri e l'ottimizzazione.

La scelta di Python come linguaggio di programmazione è dovuta dal fatto che quest'ultimo è un linguaggio *completo*.

5.2 Scelta dei principali algoritmi

In questa sezione ci soffermeremo sull'analisi e l'implementazione degli algoritmi per la risoluzione del problema del commesso viaggiatore, utilizzando come supporto l'Ant System

5.2.1 File di testo cities.txt

Questo file di testo viene usato per andare a creare il grafo delle città che verrà visitato dalle formiche.

```
1 1304 2312
2 3639 1315
3 4177 2244
4 3712 1399
5 3488 1535
6 3326 1556
7 3238 1229
8 4196 1004
9 4312 790
10 4386 570
11 3007 1970
12 2562 1756
13 2788 1491
14 2381 1676
15 1332 695
16 3715 1678
17 3918 2179
18 4061 2370
19 3780 2212
20 3676 2578
21 4029 2838
22 4263 2931
23 3429 1908
24 3507 2367
```

```
25 3394 2643
26 3439 3201
27 2935 3240
28 3140 3550
29 2545 2357
30 2778 2826
31 2370 2975
```

Dove i numeri da 1 a 31(la prima colonna) indicano i nodi (città) del grafo e le colonne 2 e 3 del file indicano le coordinate di ogni nodo.
La rappresentazione di questo grafo è la seguente:

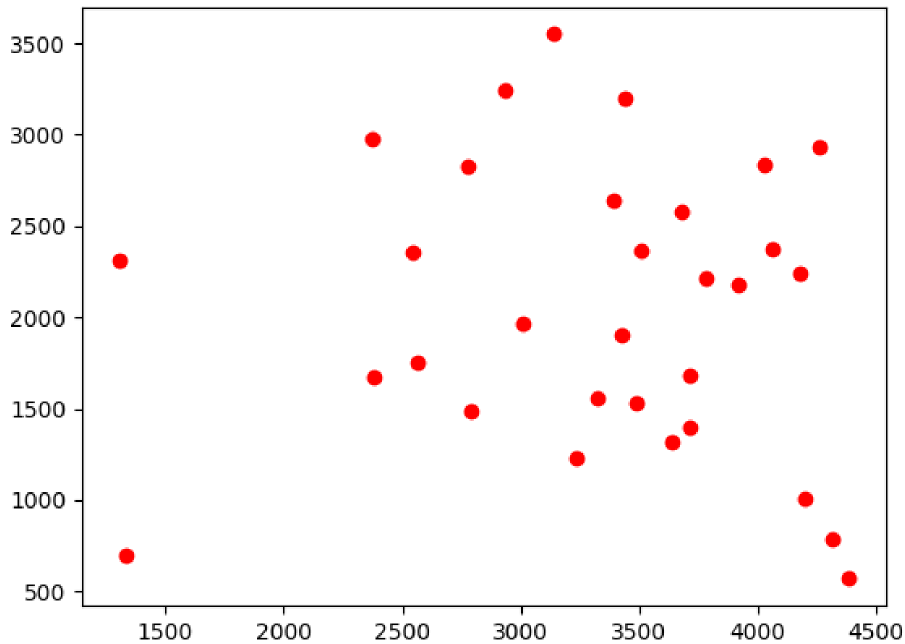

Figure 5.1: Plot del grafo delle città

5.2.2 matrix.py

Questo file .py è stato utilizzato per inizializzare tutti i parametri con cui andare a lavorare.
Contiene al suo interno quattro funzioni.

```
import math

def _set_points():
```

```
    '''
    Open the file in 'data' direcotry for set the cities
    and their coordinates
    :return:
    '''
    cities=[]
    points=[]
    with open('./data/cities.txt') as f:
        for line in f.readlines():
            city= line.split(' ')
            cities.append (dict (index=int (city[0]), x=int (city[1]), y=int (city[2])))
            points.append ((int (city[1]), int (city[2])))
    return cities,points
```

Utilizzata per creare le liste di coordinate e di città per la costruzione del grafo.

```
def _create_distance_matrix(const_matrix:list ,cities:list):
    '''
    :param cities: cities to visit
    :param const_matrix: Distance matrix initially empty
    :return: Complete distance matrix
    '''

    rank = len(cities)
    for i in range(rank):
        row= []
        for j in range(rank):
            row.append(distance (cities[i],cities[j]))
        const_matrix.append(row)
    return const_matrix,rank
```

Presa una lista vuota (const-matrix) e la lista delle città, va a riempire la matrice distanza facendo ausilio della funzione *distance* che ritorna la distanza euclidea tra due città.

```
def distance(city1: dict, city2: dict):
    '''


    :param city1: City-i
    :param city2: City-j
    :return: the Euclidian distance between 2 cities
    '''
    return math.sqrt(((city1['x'] - city2['x']) ** 2) + ((city1['y'] - city2['y']) ** 2))
```

La matrice distanza viene poi salvata all'interno di un nuovo file .txt mediante la funzione $create - file$

```python
def _create_file(const_matrix:list , rank:int):
    '''
    :param const_matrix: Distance matrix
    :param rank: Size of const_matrix
    :return: void
    '''
    f = open ('./data/distance.txt', 'w')
    for i in range (rank):
        for j in range (rank):
            f.write ("%.2f " % const_matrix[i][j])
        f.write ('\n')
    f.close ()
```

Il contenuto del file di test *distance.txt* conterrà la matrice distanza delle città contenute nel file cities.txt.
Per facilitare la lettura verrà mostrato solo parte del contenuto del file, siccome è una matrice 31x31.

```
0 2539 2874 2575 2318 2159 2217 3174 3371 3540 1737 1375 1696 1251 1617 2493 2617 2758 2478 2387 2775 3023 2163 2204 2116 2313 1877 2214 1242 1561 1255
2539 0 1074 111 267 395 410 638 854 1055 910 1164 869 1309 2389 371 908 1136 908 1264 1572 1732 629 1060 1350 1897 2050 2290 1511 1739 2089
2874 1074 0 964 989 1094 1383 1240 1460 1687 1202 1687 1580 1884 3239 731 267 171 398 602 612 692 820 681 879 1209 1592 1668 1636 1515 1949
2575 111 964 0 262 417 504 625 855 1068 907 1204 929 1360 2482 279 807 1032 816 1180 1474 1628 582 989 1284 1823 1998 2226 1510 1705 2070
2318 267 989 262 0 163 395 885 1111 1318 649 952 701 1116 2314 268 774 1013 737 1060 1411 1597 378 832 1112 1667 1792 2045 1251 1473 1823
2159 395 1094 417 163 0 339 1030 1249 1448 523 790 542 953 2172 408 859 1097 798 1080 1462 1664 367 831 1089 1649 1729 2003 1119 1383 1711
2217 410 1383 504 395 339 0 984 1160 1324 776 857 521 967 1979 655 1168 1407 1123 1418 1793 1987 705 1169 1423 1982 2034 2323 1324 1662 1950
.
.
.
1255 2089 1949 2070 1823 1711 1950 2687 2923 3138 1190 1234 1542 1299 2505 1868 1741 1796 1603 1365 1665 1894 1503 1289 1076 1093 624 961 642 434 0
```

Gli elementi sulla diagonale sono tutti 0 siccome la distanza tra una città e la stessa è nulla.

5.2.3 La classe Graph

Utilizzando le formule e le definizioni viste nel capitolo 2, andremo ad analizzare i principali algoritmi per la risoluzione del problema del commesso viaggiatore. In questa classe verrà inizializzato il grafo dove si muoveranno le formiche.

```
class Graph(object):
    def __init__(self, cost_matrix: list, rank: int):
        """
        :param cost_matrix:
        :param rank: rank of the cost matrix
        """
        self.matrix = cost_matrix
        self.rank = rank
        self.pheromone = [[1 / (rank * rank) for j in range(rank)] for i in range(rank)]
```

La funzione prende come parametri una lista chiamata const-matrix e la sua relativa dimensione *rank*. La quantità di feromone di ogni nodo del grafo viene inizializzata come

$$\frac{1}{rank^2} \tag{5.1}$$

5.2.4 La classe ACO

E' la classe dove viene creata la colonia di formiche, e dove vengoni inizializzate tutte le varibili dell'algoritmo.

```
class ACO(object):
    def __init__(self):
        """
        :param ant_count:
        :param generations:
        :param alpha: relative importance of pheromone
        :param beta: relative importance of heuristic information
        :param rho: pheromone residual coefficient
        :param q: pheromone intensity
        """
        self.Q = 10
        self.rho = 0.5
        self.beta = 10.0
        self.alpha = 1.0
        self.ant_count = 10
        self.generations = 100
```

La funzione di inizializzazione attribuisce alla variabili di regolazione dell'algoritmo dei valori. $ant - count$ corrisponde al numero di formiche che percorrono

il grafo ad ogni iterazione; quest'ultime vengono assegnate alla variabile *generations*.
alpha e *beta* sono i due parametri principali che controllano l'algoritmo; *rho* il feromone depositato nelle scorse iterazioni e Q un parametro di regolazione.

```
    def update_pheromone(self, graph:Graph, ants:list):
        '''

        :param graph: city graph
        :param ants: number of ants
        :return: new pheromene's intensity
        '''

        for i, row in enumerate(graph.pheromone):
            for j, col in enumerate(row):
                graph.pheromone[i][j] *= self.rho
                for ant in ants:
                    graph.pheromone[i][j] += ant.pheromone_delta
[i][j]
```

All'interno della classe è definita la funzione $update-pheromone$ che prende come parametri il grafo e una lista della formiche di quella iterazione. Serve per aggiornare il feromone sui bordi del grafo, utilizzando $pheromone-delta$ di ogni formica (classe che verrà definita a breve).

```
    def get_solution(self,graph: Graph):
        '''

        :param graph: city graph
        :return: The best solution and the best cost
        '''
        '''
        We need to define two variables: 'best_cost' and '
best_solution'
        tha represents the best path and cost associated with it
        '''
        best_cost = 100000000000.00
        best_solution = []
        for gen in range (self.generations):
            ants = [Ant (self, graph) for i in range (self.
ant_count)]
            for ant in ants:
                for i in range (graph.rank - 1):
                    ant.select_node()
                ant.total_cost += graph.matrix[ant.tabu[-1]][ant
.tabu[0]]
                if ant.total_cost < best_cost:
```

```
                    best_cost = ant.total_cost
                    best_solution = [] + ant.tabu
                ant.update_pheromone_delta()
            self.update_pheromone(graph, ants)
        return best_solution, best_cost
```

E' la funzione cardine dell'algoritmo; ossia quella di trovare il miglior percorso all'interno del grafo. Ritorna un numero reale per il costo del cammino e una lista di nodi relativa al path del miglior cammino nel grafo. Vengono inizializzate due variabili relative al costo($best - cost$) e al cammino($best - solution$). Alla prima variabile viene assegnato un numero pari a 10^{11}, mentre al cammino viene assegnata una lista vuota.
Per ogni generazione (100 da me assegnata) verranno create 10 formiche che andranno a percorrere il grafo. Queste 10 formiche utilizzeranno la funzione definita nella classe Ant, $select - node$, per decidere quale nodo andare a visitare nel grafo. Viene successivamente calcolato il costo del loro spostamento, se minore di best-cost, questo diventerà uguale al nuovo costo minore e verrà aggiunto al cammino soluzione tale spostamento.
Verrà aggiornanto il pheromone-delta di ogni formica e dell'intera colonia successivamente.

5.2.5 La classe ant

Tale classe è dedicata alla singola formica della colonia. Alla creazione, ogni formica inzializza le seguenti variabili:

1. **total-cost**: Numero che viene assegnato allo spostamento dal nodo i al nod j del grafo
2. **tabu**: Lista dei nodi visitati
3. **pheromone-delta**: Incremento del feromone locale
4. **allowed**: Lista dei nodi che non sono stati ancora visitati
5. **eta**: Variabile per la regolazione dell'algoritmo

```
class Ant(object):
    def __init__(self, aco: ACO, graph: Graph):
        self.colony = aco
        self.graph = graph
        self.total_cost = 0.0
        self.tabu = []  # tabu list
```

```python
        self.pheromone_delta= []  # the local increase of pheromone
        self.allowed = [i for i in range(graph.rank)]  # nodes which are allowed for the next selection
        '''
        Eta is equal to 0 if i==j
        because graph.matrix[i][j] is the element on the diagonal
        of the distance matrix
        '''
        self.eta = [[0 if i ==j else 1/graph.matrix[i][j] for j in range(graph.rank)] for i in range(graph.rank)]
        start = random.randint (0, graph.rank - 1)  # Start of problem
        self.tabu.append(start) # append in the list the starting node
        self.current = start
        '''
        Once ispected the node is removed from the list
        of accessible nodes
        '''
        self.allowed.remove(start)
```

Ogni formica lavorerà con il grafo delle città creato in precedenza.
Viene scelto come nodo di partenza un nodo scelto in maniera randomica tra la lista di quelli non ancora visitati e viene inserito all'interno della tabu list. Lo starting node viene assegnato poi alla variabile *current* così da porterlo eliminare dalla lista *allowed*.

```python
    def select_node(self):
        '''

        :param graph:All the nodes(cities)
        :return: next node that will be chosen by the ant
        '''

        '''
        We need to initialize the denominator.
        denominator = Summation of all nodes that belong in allowed list multiplied by
        the intensity of the pheromone multiplied by the distance between city-i and city-j
        '''
        denominator = 0
        for i in  (self.allowed):
            denominator += self.graph.pheromone[self.current][i] ** self.colony.alpha * self.eta[self.current][i]**self.colony.beta
```

```python
        '''
        Now we need to define the transition probability
        from city-i to city-j for the k-th ant.
        The coefficient must be set to a value <1
        '''
        probability = [0 for i in range(self.graph.rank)] #all the coefficients are equals to 0 for every node

        for i in range(self.graph.rank) :
            #I need to check if the node is allowed to visit
            try:
                self.allowed.index(i)
                probability[i] = self.graph.pheromone[self.current][i] ** self.colony.alpha * \
                    self.eta[self.current][i] ** self.colony.beta / denominator
            except ValueError:
                pass

        '''
        The ant must choose a node to move to.
        It chooses it randomly
        '''
        selected=0
        choosen = random.random()
        for i, probabilities in enumerate(probability):
            choosen -= probabilities
            if choosen <= 0:
                selected = i
                break
        self.allowed.remove(selected)
        self.tabu.append(selected)
        self.total_cost += self.graph.matrix[self.current][selected]
        self.current = selected
```

E' la funzione che viene chiamata per ogni formica all'interno della funzione $get - solution$ della classe ACO. Vengono applicate le formule viste nel capitolo 2 per la risoluzione del problema.
Vengono inizializzate le variabili come *denominator* (=0) e l'array di probabilità di visita di nodi. Il denominatore viene poi calcolato come la somma di tutti i nodi che compaiono nella allowed list moltiplicata per l'intensità del feromone, moltiplicato per la distanza tra una città i e una citta j.

$$denominator = \sum_{l \in J_i^k} \tau_{il}^{\alpha} \cdot \eta_{il}^{\beta} \tag{5.2}$$

Successivamente per ogni nodo verrà calcolata la *Regola casuale di transizione*

proporzionale citata nel capitolo 2, ovvero la regola di spostamento da un nodo i ad un nodo j.
Una volta assegnata questa probabilità ad ogni nodo del grafo, la formica deve scegliere quale andare a visitare e lo fa in maniera del tutto randomica. Verrà selezionato un nodo nella lista dei visitabili per poi essere rimosso e inserito nella *tabu list*. Viene infine calcolato il costo della distanza dal nodo corrente e il nodo appena selezionato.

L'ultima funzione della classe Ant è la funzione $update - pheromone - delta$ che serve per l'aggiornamento della quantità di feromone sul bordo del grafo.

```
    def update_pheromone_delta(self):
        '''
        :param graph: City graph
        :return: update pheromone of nodes
        '''
        self.pheromone_delta = [[0 for j in range (self.graph.rank)] for i in range (self.graph.rank)]
        for _ in range (1, len (self.tabu)):
            i = self.tabu[_ - 1]
            j = self.tabu[_]
            self.pheromone_delta[i][j] = self.colony.Q / self.graph.matrix[i][j]
```

L'aggiornamento del feromone, citato sempre nel capitolo 2, è il seguente:

$$\Delta t_{ij}^{k}(t) = \frac{Q}{L^{k}(t)} \tag{5.3}$$

Dove Q è il parametro di regolazione fornito appena creata la colonia, mentre $L^k(t)$ è la lunghezza del percorso in quell'iterazione.

5.2.6 main.py

E' il file .py con cui viene lanciato l'algoritmo.
Una volta definito l'algortimo, possiamo procedere con la creazione della matrice e la risoluzione del problema.

```
    from graph import graph_D
from aco import ACO, Graph
from matrix import _set_points, _create_distance_matrix, _create_file



```

```
def main():
    cities,points = _set_points()
    const_matrix = []
    const_matrix,rank = _create_distance_matrix(const_matrix,
   cities)
    _create_file(const_matrix,rank)
    '''
    After initialize some variables
    the algorithm can start
    '''
    ants = ACO()
    graph = Graph (const_matrix, rank)
    path,cost = ants.get_solution(graph)
    print ('cost: {}, path: {}'.format (cost, path))

    graph_D (points,path)
if __name__ == '__main__':
    main()
```

L'esecuzione del file main.py stamperà sul terminale il costo e il cammino soluzione per l'insieme di città presentato poco fa.

```
    marco@marco-Lenovo-ideapad-330S-15IKB:~$ cd PycharmProjects/
   ESP/
marco@marco-Lenovo-ideapad-330S-15IKB:~/PycharmProjects/ESP$
   python3 main.py
cost: 15892.800061811617, path: [0, 14, 13, 11, 12, 10, 22, 15,
   4, 5, 6, 1, 3, 7, 8, 9, 16, 18, 17, 2, 20, 21, 19, 23, 24,
   25, 27, 26, 29, 30, 28]
```

La partenza, come scritto nel codice, avviene da un nodo casuale dei 31 previsti. La rappresentazione grafica del percorso è affidata al file graph.py.

5.2.7 graph.py

Viene utilizzato per rappresentare il percorso effettuato dalle formiche per trovare il cammino minimo per la risoluzione del problema del commesso viaggiatore.
C'è l'utilizzo del modulo quale *matplotlib* per il plot del grafo.
Il file è il seuguente:

```
import matplotlib
import operator
import matplotlib.pyplot as plt
import pylab
from scipy.optimize import curve_fit
def graph_D (points,path):

```

```
8     x= []
9     y= []
10    for point in points:
11        x.append(point[0])
12        y.append(point[1])
13    plt.plot(x,y,'co',color='red')
14    for _ in range(1, len((path))):
15        i=path[_ -1]
16        j=path[_]
17        plt.arrow (x[i], y[i], x[j] - x[i], y[j] - y[i], color='lightgreen', length_includes_head=True)
18    plt.show()
```

La funzione prende come parametri i punti (coordinate delle città) e il path (cammino soluzione) che sono definiti nel main. Le righe di codice fino alla tredicesima inclusa servono per eseguire i plot delle sole città (vedi figura n. 5.1); il resto del codice, invece, serve per disegnare il cammino soluzione.
Eseguendo gli stessi comandi all'interno del terminale, i risultati saranno i seguenti:

```
marco@marco-Lenovo-ideapad-330S-15IKB:~$ cd PycharmProjects/ESP/
marco@marco-Lenovo-ideapad-330S-15IKB:~/PycharmProjects/ESP$ python3 main.py
ccost: 15799.869920620939, path: [13, 11, 12, 10, 22, 15, 4, 5, 6, 1, 3, 7, 8, 9, 2, 17, 16, 18, 23, 19, 24, 20, 21, 25, 27, 26, 29, 30, 28, 0, 14]
```

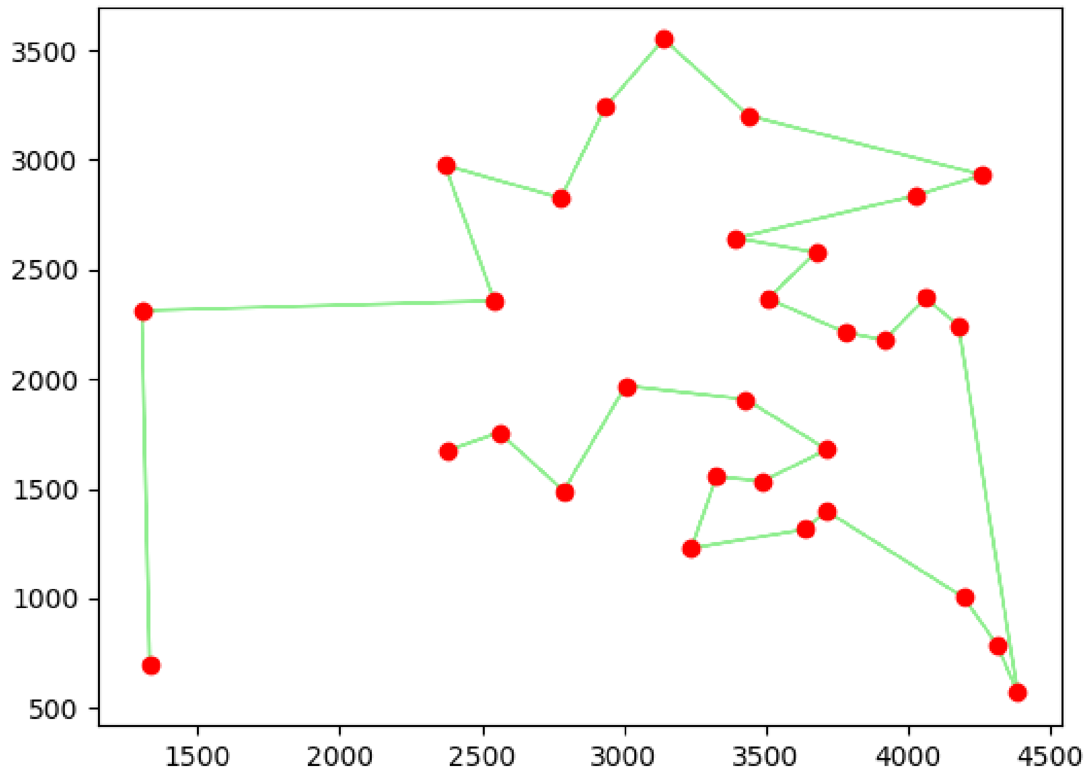

Figure 5.2: Plot del cammino soluzione

L'immagine 5.2 rappresenta il cammino soluzione restituito dal terminale.

5.2.8 Ulteriori test

Il test per verificare il funzionamento dell'algoritmo è stato effettuato con 31 città.
Per un ulteriore prova, verrà aggiornato il file cities.txt con 50 città. Mandando in esecuzione il main il risultato sarà il seguente:

```
marco@marco-Lenovo-ideapad-330S-15IKB:~$ cd PycharmProjects/ESP/
marco@marco-Lenovo-ideapad-330S-15IKB:~/PycharmProjects/ESP$
    python3 main.py
cost: 40986.37249826776, path: [26, 18, 16, 29, 5, 6, 17, 43,
    30, 37, 8, 0, 7, 39, 14, 11, 10, 22, 13, 24, 12, 20, 35, 27,
    42, 48, 36, 49, 44, 34, 9, 23, 41, 25, 3, 1, 28, 4, 47, 38,
    31, 33, 40, 15, 21, 2, 46, 19, 32, 45]
```

mentre la rappresentazione grafica è la seguente:

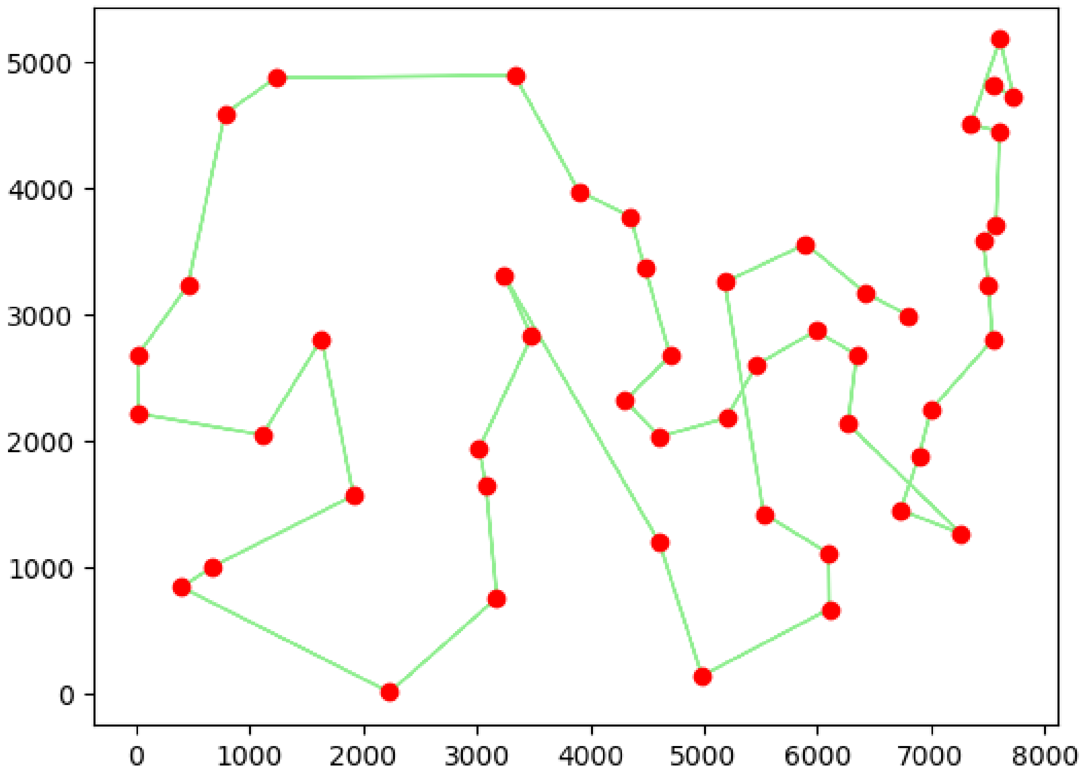

Figure 5.3: Plot del cammino soluzione per 50 città

Chapter 6

Conclusioni

Questa relazione descrive il procedimento seguito per la risoluzione del problema del commesso viaggiatore (TSP) mediante l'utilizzo dell'algoritmo della colonia delle formiche (ACO).
In particolare la relazione, tramite l'utilizzo di ACO, risolve il percorso minimo necessario per visitare tutti i nodi dell'insieme delle città (C). Il numero e le coordinate delle città che vengono utilizzate sono state scelte direttamente dall'autore.
Per verificare la veridicità dei risultati ottenuti dall'esecuzione del programma,per una serie di dati differenti, manualmente questi sono stati confrontati con i valori riportati nel file *distance.txt*. Dato che i dati emersi risultano compatibili è possibile affermare che il codice risolve correttamente il problema e che l'algoritmo ACO scelto viene utilizzato in modo opportuno.

Bibliography

[1] M. Pappalardo e M. Passacantando. *Ricerca Operativa seconda edizione* (2012)

[2] M. Dorigo e A. Colorni. *The Ant System* : *Optimization by a colony of cooperating agents*

[3] M. Dorigo, M. Birattari e T. Stützle *Ant Colony Optimization* : *Artificial Ants as a Computational Intelligence Technique* (2006)