Risoluzione del tsp

Filippo di Gravina, Ciocia Gioacchino

Università degli studi di Bari

Sommario

Abstract

Il nostro progetto si concentra sulla risoluzione del Traveling Salesman Problem (TSP), un classico problema di ottimizzazione. Utilizzeremo una varietà di algoritmi, tra cui Two Opt, Genetico, Branch and Bound e Dynamic Programming, per trovare soluzioni efficienti al problema. Inoltre, implementeremo algoritmi di visualizzazione per rappresentare graficamente i risultati su una mappa, offrendo una chiara comprensione delle soluzioni ottenute. Questo approccio integrato permette di esplorare diverse strategie di risoluzione del TSP e di valutarne l'efficacia attraverso una visualizzazione intuitiva e dettagliata dei risultati.

Elenco argomenti di interesse:

Dal capitolo 3: branch and bound, programmazione dinamica.

Dal capitolo 4: iterative best improvement, algoritmo genetico, random step, random restart, valutazione degli algoritmi randomici.

Dal capitolo 5 e dal capitolo 15: basi di conoscenza, inferenza di nuova conoscenza

Dal capitolo 7: apprendimento supervisionato, KNN e case-based-reasoning

Dal capitolo 8: reti neurali

Definizione del problema:

Il problema del commesso viaggiatore, noto anche come TSP (Traveling Salesman Problem), è un problema di ottimizzazione. La definizione è la seguente:

Dato un insieme di città e le distanze tra ciascuna coppia, il problema consiste nel trovare il percorso più breve che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza.

In termini di teoria dei grafi, il problema del commesso viaggiatore può essere definito come la ricerca di un particolare ciclo hamiltoniano (un ciclo che visita ogni nodo una volta) che minimizza un determinato parametro, come la distanza totale percorsa.

Matematicamente, il problema del TSP può essere associato a un grafo G=(V,A), dove V è l'insieme degli n nodi (o città) e A è l'insieme degli archi (o strade). Il costo dell'arco per andare dal nodo i al nodo j è indicato con cij. Il TSP è simmetrico se, per ogni coppia di città i e j, cij=cji, altrimenti si dice asimmetrico.

Risoluzione del tsp

Il TSP (traveling salesman problem), è uno dei problemi più noti, e negli anni si sono susseguite diverse soluzioni che hanno migliorato man mano i risultati ottenuti. La migliore soluzione attualmente individuata è stata fornita da Concorde, il quale è riuscito a risolvere un'istanza del tsp che presentava 85.900 nodi (un totale di $7.37 * 10^9$ archi), in circa 136 cpu years.

Questo problema fa parte di quella classe di problemi denominati NP-hard, ovvero questo problema presenta le seguenti caratteristiche:

- Non è possibile individuare una soluzione in tempo polinomiale utilizzando algoritmi deterministici.
- I problemi facenti parte alla classe di problemi NP-HARD hanno complessità almeno maggiore di quella del problema più complesso in NP

Le soluzioni che esauriscono l'esplorazione dello spazio delle soluzione (e contestualmente sono anche soluzioni deterministiche a questo problema) sono le seguenti:

- Brute force: Esplorazione dell'intero spazio di ricerca, mediante la creazione di tutti i percorsi possibili. Questo algoritmo ovviamente trova una soluzione, ma la sua complessità nel caso pessimo è $O(n!)$. Questo comporta l'impossibilità nell'utilizzo di questo approccio già con 14 nodi.
- DP: Questo approccio permette di ridurre il tempo computazionale necessario per l'eplorazione dello spazio di ricerca, mediante l'utilizzo di bitmask, le quali permettono una rappresentazione più efficiente di tutti i possibili percorsi. Questo algoritmo richiede tempo computazionale minore rispetto alla brute force, dato che nel caso pessimo si ha una complessità $O(n^2 * 2^n)$. Questo algoritmo migliora la capacità di risoluzione della brute force permettendo di ottenere soluzioni anche per un grafo, completamente connesso, con 24 nodi.

Questi algoritmi, come abbiamo potuto constatare, non sono vie percorribili per la risoluzione di questa tipologia di problemi. A questo punto la domanda potrebbe essere: "Quale tipologia di algoritmi potrebbe essere applicata per la risoluzione di questo problema?"

Gli algoritmi maggiormente utilizzati per la risoluzione di questa classe di problemi sono gli algoritmi euristici. Questi algoritmi appartengono alla classe di algoritmi non deterministici. Questa tipologia di problemi esplora lo spazio di ricerca al fine di individuare delle soluzioni non necessariamente ottime, ma accettabili rispetto ad alcuni criteri pre-impostati. È possibile, comunque, utilizzare questi algoritmi per raggiungere una soluzione ottimale non impostando vincoli sul tempo necessario per la ricerca della soluzione ed implementando delle tecniche per uscire dai minimi locali che non siano anche il minimo globale. Questi algoritmi privilegiano percorsi promettenti e utilizzano tecniche di ottimizzazione locali o globali.

Contesto

Lo scenario in cui abbiamo deciso di operare è il seguente:

"Un tour operator vuole organizzare un tour nelle maggiori città italiane. L'obiettivo è quello di minimizzare il tempo di percorrenza per i viaggi. Per farlo il tour operator deve tenere in considerazione una serie di fattori quali:

- Quanto è dissestata la strada che collega due città?
- Quali sono le condizioni metereologiche?
- Ci sono lavori sulla strada che porta da una città all'altra?"

Questi fattori ovviamente contribuiscono all'aumento del tempo di percorrenza sulle strade extraurbane che collegano le città.

Abbiamo deciso di utilizzare come territorio di riferimento il territorio italiano. Abbiamo quindi effettuato una rappresentazione mediante un grafo delle città italiane, in cui le città sono rappresentate dai nodi del grafo, mentre gli archi rappresentano le strade che collegano due città.

Adesso mostreremo come siamo andati a creare i valori influenzanti per le strade.

Assegnazione dei valori influenzanti.

Abbiamo quindi supposto che le città fossero connesse tra loro mediante una strada extraurbana, con velocità di percorrenza 90km/h. Questo fattore ovviamente viene influenzato dalle condizioni in cui il trasporto avviene. Riportiamo nella tabella sottostante l'influenza che ogni fattore ha sul tempo.

	coefficiente			
Fattori Influenzanti	No	lieve	moderata	forte
Pioggia	1	1.1	1.2	1.3
Strada Dissestata	1	1.1	1.2	1.4

	entità	
Fattore Influenzante	No	Si
Lavori	1	1.5

Questi fattori influenzati vengono assegnati nel seguente modo:

- Per l'80% dei nodi viene assegnata la situazione meteorologica in modo randomico

- Per i restanti nodi abbiamo implementato in Prolog un knn che, attraverso lo studio della situazione metereologica delle città più vicine attribuisce alla città la situazione metereologica più appropriata
- Per gli archi viene assegnata la situazione meteorologica peggiore dei due nodi che lo coinvolgono. Ad esempio, se in una delle due città c'è una pioggia moderata e nell'altra città c'è una pioggia forte, consideriamo per l'arco la pioggia forte.
- A questo punto, il grafo generato viene utilizzato dai nostri algoritmi per la ricerca di una soluzione.

1 **Algoritmi utilizzati**

Gli algoritmi che abbiamo testato sul tsp sono diversi, e ognuno di questi ha ottenuto diverse prestazioni. I test sugli algoritmi sono stati condotti sui file appartenenti alla libreria tsplib95. L'utilizzo di questa libreria per la valutazione degli algoritmi è legata al fatto che questa venga utilizzata come benchmark standard per la valutazione degli algoritmi per la risoluzione del TSP.

Andiamo a vedere singolarmente i risultati ottenuti da questi ed il loro funzionamento

1.1 DP

Come anticipato nell'introduzione di questo documento, l'algoritmo in questione non può essere impiegato per trovare una soluzione ottimale in grafi con più di 25 nodi, poiché richiederebbe un'eccessiva quantità di tempo computazionale.

Implementazione dell'algoritmo:

Questo algoritmo, per la sua implementazione, richiede una fase di preparazione. In questa fase si vanno a generare le seguenti componenti:

- **Matrice di adiacenza:** per ogni coppia di nodi i e j appartenenti all'insieme dei nodi, si va ad inserire all'interno della matrice in posizione M<i,j> il costo dell'arco che porta da i a j.
- **Matrice dp:** la matrice dp viene inizializzata a -1. Questa matrice è di dimensione $n * 2^n$ e viene utilizzata per memorizzare le soluzioni ottimali ai sotto-problemi. Ogni cella della matrice in posizione i, j dove i è il nodo che stiamo considerando in questo momento mentre j è l'indice che rappresenta i nodi visitati nelle precedenti iterazioni attraverso la bitmask.
- **Matrice parent:** la matrice parent viene inizializzata a -1. Questa matrice è di dimensione $n * 2^n$ e viene utilizzata per memorizzare qual è il nodo successivo da visitare. La memorizzazione è basata sul risultato delle chiamate ricorsive successive della funzione.

Funzionamento della DP.

```
def tsp (self, pos, bitmask):

    if bitmask == 2 ** self.n - 1:
        return self.adj[pos][0]

    if self.dp[pos][bitmask] != -1:
        return self.dp[pos][bitmask]

    ans = self.inf

    for nxt in range(self.n):
        if nxt != pos and (bitmask & (2 ** nxt)) == 0:
            curr = self.adj[pos][nxt] + self.tsp(nxt, bitmask | (2 ** nxt))
            if curr < ans:
                ans = curr
                self.parent[pos][bitmask] = nxt

    self.dp[pos][bitmask] = ans
    return int(self.dp[pos][bitmask])
```

Com'è possibile osservare andiamo a verificare che il nodo che stiamo attualmente considerando non sia stato ancora visitato. Questa operazione viene svolta mediante l'utilizzo della bitmask.

Attraverso la matrice parent è possibile anche ricostruire il percorso migliore generato. Questa operazione viene svolta all'interno della funzione reconstruct_path di cui lasciamo l'implementazione al di sotto di questo paragrafo.

```
def reconstruct_path (self):

    path = []
    curr = 0
    mask = 1

    while curr != -1:
        path.append (curr)
        curr = self.parent[curr][mask]
        mask = mask | int(2 ** curr)

    path.append (path[0])
    return path
```

1.2 Branch and bound

Il primo approccio implementativo di questo algoritmo è stato provare quello di implementare la branch and bound in Prolog. Abbiamo notato però che i tempi di esecuzione erano troppo lunghi e, per questo motivo, abbiamo deciso di implementare lo stesso algoritmo in Python per verificarne le differenze in termini di tempo computazionale.

La variante che abbiamo deciso di implementare è un'euristica informata in quanto l'upper bound iniziale è fornito da altri algoritmi. Infatti, nelle fasi di testing di questo algoritmo siamo andati ad utilizzare le soluzioni sub ottimali trovate dall'algoritmo two-opt o dal genetico.

Questo algoritmo, come visto anche nella teoria del capitolo 3 è di semplice implementazione. Inoltre, riesce ad essere migliore in termini di tempo computazionale rispetto alla dp nel caso in cui l'upper bound iniziale rappresenta una stima sufficientemente buona della soluzione.

Implementazione della branch and bound in python:

```python
def bb (path, cost, remaining):

    global sol
    global bound
    global cont

    m = len(path)
    cont += 1

    if m == n + 1:
        if cost < bound and path[0] == path[-1]:
            sol, bound = path, cost
        return

    if cost + lower * (n+1 - m) > bound:
        return

    for x in remaining:
        if x == path[-1]:
            continue
        print(path)
        r = remaining.copy()
        r.remove(x)

        p = path.copy()
        p.append(x)

        peso = cost

        if len(p) > 1:
            peso += G[path[-1]][x]['weight']

        bb (p, peso, r)
```

Essendo un algoritmo ricorsivo, abbiamo utilizzato delle variabili globali per evitare di doverle passare ogni volta come parametri alla funzione.

Attraverso l'utilizzo del bound è possibile effettuare il pruning di parte dello spazio di ricerca ed effettuare eventualmente il backtracking per esplorare i percorsi promettenti.

L'esplorazione di un percorso termina se:

- Il percorso che stiamo considerando è completo ed il suo costo è minore del bound.
- Se il costo del percorso parziale che stiamo considerando supera il bound.

Questo algoritmo garantisce l'ottimalità, ma con grafi che presentano un numero di nodi troppo elevati non riesce a trovare una soluzione in tempi computazionali sufficientemente buoni, specialmente se il bound non è un'approssimazione sufficientemente buona della soluzione ottimale.

Implementazione della branch and bound in Prolog:

In questo caso siamo andati a definire le seguenti regole:

tsp\5 caso base

```
% Caso base: tutti i nodi visitati, ritorna al nodo iniziale e calcola il costo
tsp(Current, [], Path, Cost, Start) :-
    edge(Current, Start, CostToStart),
    append(Path, [Start], PathToStart),
    CostTot is Cost + CostToStart,
    update_best_path(PathToStart, CostTot).
```

Nel caso base, ovvero il caso in cui la lista dei nodi da visitare è vuota, siamo andati ad effettuare le seguenti operazioni:

- edge(current, start, costToStart): Questa istruzione trova il costo per tornare dal nodo corrente ‘Current’ al nodo di partenza ‘Start’.
- append(Path, [Start], PathToStart): Mediante l’utilizzo di questo predicato, siamo andati ad aggiungere il nodo Start in coda al path che abbiamo generato nei passi ricorsivi.
- CostTot is Cost + CostToStart: Siamo andati a modificare il costo totale.
- update_best_path(PathToStart, CostTot): permette di verificare se il percorso corrente è quello migliore.

tsp\5 chiamata ricorsiva

```
% Caso ricorsivo: trova il percorso migliore tra i nodi rimanenti
tsp(Current, Nodes, Path, CostP, Start) :-
    %write('Current: '), write(Current), nl,
    %write('Nodes: '), write(Nodes), nl,
    %write('Path: '), write(Path), nl,
    %write('Cost: '), write(CostP), nl, nl, nl,
    select(Next, Nodes, RestNodes),
    edge(Current, Next, Cost),
    bound(Bound),
    CostTot is Cost + CostP,
    CostTot =< Bound,
    append(Path, [Next], NewPath),
    tsp(Next, RestNodes, NewPath, CostTot, Start),
    fail. % Forza il backtracking per esplorare tutte le possibilità
```

La funzione tsp/5 ha i seguenti parametri:

- Current: la città che stiamo attualmente visitando.
- Nodes: le restanti città da visitare.
- Path: il percorso attualmente costruito
- Costp: Costo del percorso attuale.
- Start: città di partenza.

Il funzionamento della funzione è il seguente:

- select(Next, Nodes, RestNodes): è un predicato standard definito in prolog, che restituisce True quando RestNodes è la lista che corrisponde a Nodes da cui è stato eliminato l'elemento Next. In questo modo andiamo a prelevare uno alla volta l'elemento successivo da considerare da quelli contenuti nella lista dei nodi restanti.
    - edge(Current, Next, Cost): questo predicato ci permette di prelevare il costo dell'arco da Current a Next che abbiamo definito nel file dei fatti che abbiamo preventivamente costruito.

- bound(Bound): Questo predicato viene richiamato per controllare il valore del fatto contenuto all'interno del file che abbiamo fornito in input.
- CostTot is Cost + CostP e CostTot =< Bound: Queste due istruzioni vengono utilizzate per valutare il costo del percorso attuale + costo dell'arco che porta dal nodo Current al nodo Next, e ci permette di effettuare il pruning dell'albero di ricerca se questa quantità supera Bound, in quanto questa quantità risulta già subottimale rispetto alla soluzione trovata.
- append(Path, [Next], NewPath): Aggiunge la città corrente Next al percorso Path creando un nuovo percorso.
- tsp(Next, RestNodes, NewPath, CostTot, Start): Effettua una chiamata ricorsiva con i valori aggiornati dei parametri.
- Fail: viene utilzzato per forzare il backtracking. Nello specifico permette a Prolog di esplorare tutte le soluzioni possibili, dato che normalmente dopo l'individuazione di un valore al di sotto del bound la computazione terminerebbe in quanto verrebbe restituito True.

Update_best_path:

```
update_best_path(NewPath, NewCost) :-

    bound(CurrentBound),

    %write('Path: '), write(NewPath), nl,
    %write('Cost: '), write(NewCost), nl,
    %write('Bound: '), write(CurrentBound), nl,

    NewCost < CurrentBound,

    retractall(best_path(_, _)),
    write('Path: '), write(NewPath), nl,
    write('Cost: '), write(NewCost), nl,
    assertz(best_path(NewCost, NewPath)),

    retractall(bound(_)),
    assertz(bound(NewCost)),  % Aggiornare il bound

    best_path(Cost, Path),
    bound(Bound),

    write('Percorso: '), write(Path), nl,
    write('Costo: '), write(Cost), nl,
    write('Nuovo bound: '), write(Bound), nl.
```

Questo predicato viene utilizzato per aggiornare, mediante il predicato assertz di Prolog, il percorso migliore e il bound nel caso in cui il costo del percorso attualmente considerato sia migliore del bound precedente. Nel caso in cui il costo sia minore andiamo a:

- Rendere il bound uguale al nuovo costo.
- Prendere il percorso attuale e a memorizzarlo nella variabile best_path

Valutazione delle due alternative

Abbiamo riscontrato delle differenze in termini di tempo computazionale utilizzato dai due algoritmi. Nello specifico, abbiamo notato che la variante Prolog dell'algoritmo risulta essere la più veloce. Abbiamo quindi deciso di includere nel progetto la possibilità di utilizzare questo algoritmo.

1.3 Algoritmo genetico

Il primo degli algoritmi euristici che abbiamo testato su questo scenario è stato l'algoritmo genetico. Questo algoritmo si adatta perfettamente alla tipologia di problema che stiamo andando a risolvere; infatti, come visto nel capitolo 4, l'obiettivo è quello di esplorare uno spazio di ricerca molto ampio al fine di individuare le soluzioni ottime attraverso una ricerca locale.

La nostra implementazione di questo algoritmo è stata la seguente:

1.3.1 Implementazione dell'algoritmo genetico:

```
def genetic_algorithm(self, file, population_size, elite_size, mutation_rate, generations, tempo, migliore):
    population = [self.nearest_neighbor() for _ in range(population_size)]
    self.inizio = time.time()
    best_distance = 1e15
    for generation in range(generations):
        population = sorted(
            population, key=lambda tour: self.total_distance(tour))
        next_generation = population[:elite_size]
        while len(next_generation) < population_size:
            if random.random() < mutation_rate:
                parent1, parent2 = random.sample(
                    population[:elite_size], 2)
                offspring = self.mutate(self.crossover(parent1, parent2))
            else:
                offspring = self.nearest_neighbor()
            next_generation.append(offspring)
        population = next_generation
    best_tour = min(population, key=lambda tour: self.total_distance(tour))
    best_distance = self.total_distance(best_tour)
    return best_tour, best_distance, time.time() - self.inizio
```

Innanzitutto, abbiamo deciso di implementare questo algoritmo mediante l'utilizzo di una approssimazione iniziale. Per farlo ci siamo costruiti una funzione che calcolasse il nearest neighbour nel seguente modo:

```
def nearest_neighbor(self):
    unvisited_cities = set(self.node_list)
    current_city = random.choice(self.node_list)
    unvisited_cities.remove(current_city)
    tour = [current_city]
    while unvisited_cities:
        nearest_city = min(unvisited_cities, key=lambda city: self.graph[current_city][city]['weight'])
        unvisited_cities.remove(nearest_city)
        tour.append(nearest_city)
        current_city = nearest_city
    return tour
```

Abbiamo creato un set di città non visitate. A questo punto la prima città viene scelta mediante una random choice. Questa città viene rimossa dalla lista delle città non visitate, e viene inserita all'interno del vettore tour.

A questo punto andiamo ad iterare fino a quando il vettore unvisited non risulta essere vuoto.

Ad ogni iterazione andiamo a cercare la città all'interno del vettore unvisited con distanza minima dall'ultima città inserita all'interno del vettore tour.

Questo meccanismo ci permette di ottenere un vettore di tour validi, in quanto ogni elemento è presente nel vettore una sola volta e il vettore risultante è un'approssimazione iniziale della soluzione.

Funzionamento dell'algoritmo genetico

Iteriamo per il numero di generazioni fornito in input alla funzione, e per ogni iterazione, andiamo ad eseguire le seguenti operazioni:

- Utilizziamo una lambda function per individuare la distanza di ogni percorso.
- Inizializziamo il vettore di tour new_generation con gli elementi migliori della precendente generazione. Nello specifico, il numero di elementi che conserviamo è dato da un fattore chiamato elite.
- Iteriamo fino a quando la dimensione di new_generation non è uguale a quella della popolazione precedente.

- A questo punto, con probabilità definita dal mutation rate, abbiamo due possibilità:
    - Il nuovo elemento generato, appartenente alla popolazione, è generato mediante il nearest neighbour.
    - Viene generato un nuovo elemento mediante l'utilizzo del metodo crossover, il quale effettua l'operazione di crossover a due punti su due genitori scelti randomicamente dagli elementi appartenti all'elite preservata. Al termine dell'operazione di crossover, viene inoltre utilizzato l'algoritmo di mutation che permette di scambiare due geni della nuova progenie.

  In questo modo andiamo quindi a generate tutti gli n - elite_size membri della nuova popolazione.

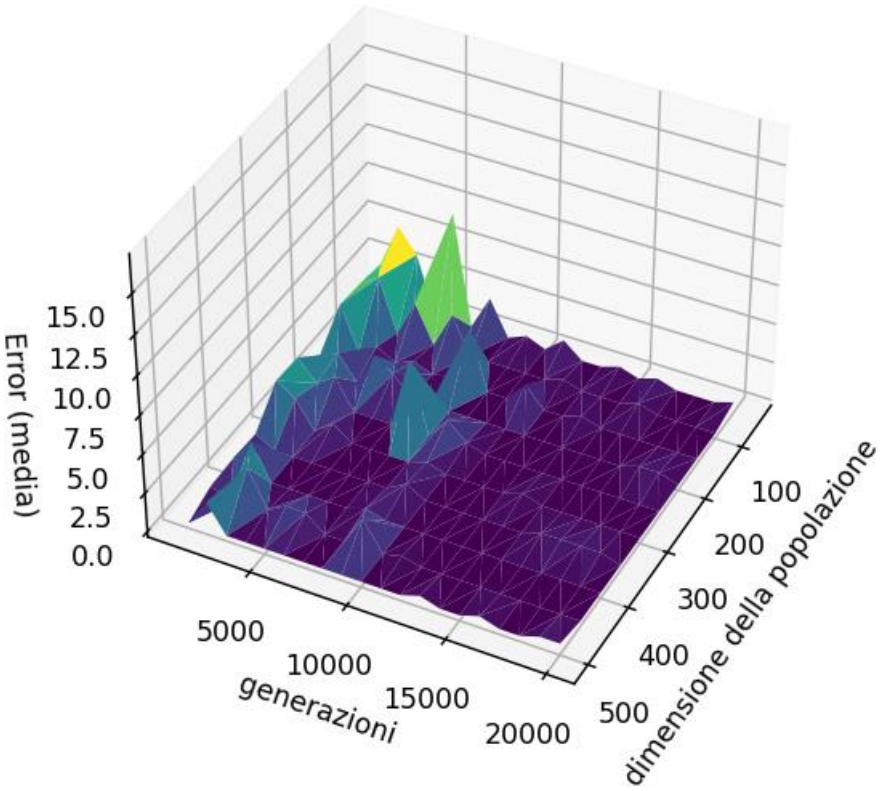
Al termine dell'algoritmo, andiamo ad individuare l'elemento migliore all'interno della popolazione, e ne calcoliamo la lunghezza del percorso.

1.3.2 Scelta dei parametri

Per la scelta dei parametri, per questo algoritmo, abbiamo condotto una serie di test provando diversi parametri, per individuare quali tra questi fossero i migliori. Nello specifico, questo algoritmo è stato testato su vari grafi della libreria tsplib95, e i grafi andavano da 14 nodi (ovvero "burma14") a 127 nodi (ovvero "bier127"). Purtroppo, questo algoritmo garantisce soluzioni approssimativamente ottime in tempo accettabile solo per grafi sotto i 150 nodi. Infatti, al crescere del numero di nodi del grafo, la dimensione della popolazione deve crescere di conseguenza. In questo modo, il vettore di tour diventa troppo grande.
I risultati individuati al termine di questi test sono i seguenti:

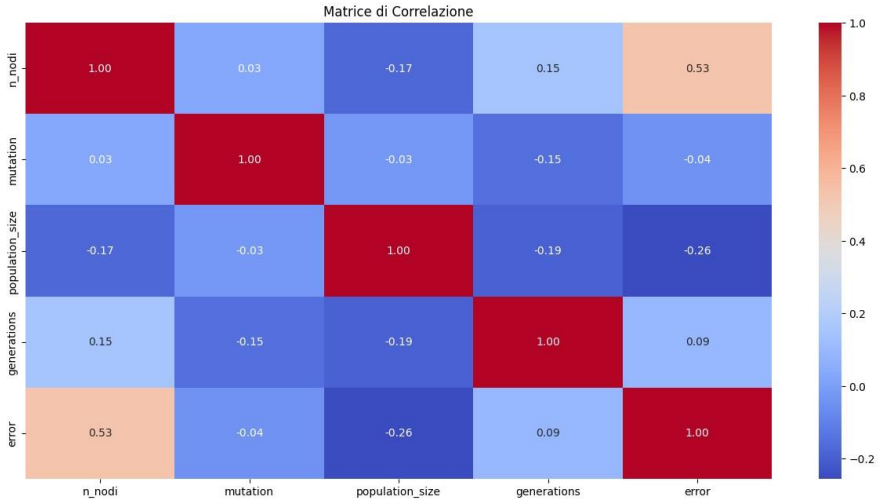
Siamo andati a questo punto ad analizzare come adattare i vari parametri dell'algoritmo per ottimizzare i risultati ottenuti.

Abbiamo quindi utilizzato delle componenti grafiche come quelle che alleghiamo nell'immagine sottostante per valutare quali parametri utilizzare.



Matrice di correlazione:

Con i dati raccolti, siamo andati a creare la matrice di correlazione per visualizzare come la variazione dei parametri condizionanti va a modificare il risultato della predizione, ottenendo il seguente risultato.



Come si può osservare, il parametro che influenza maggiormente il risultato è la dimensione della popolazione. Infatti, si può notare che, al crescere della dimensione della popolazione, decresce l'errore (dato che il fattore di correlazione è negativo).

Al termine di questo studio, abbiamo individuato i seguenti parametri, come parametri ideali da utilizzare:

- Mutation_rate = 0.001
- Population_size = 350
- Generazioni = 8000

1.4 k-opt

L'algoritmo k-opt, come visto nella letteratura scientifica, risulta essere una delle migliori alternative per la ricerca di soluzioni per questo problema sia per l'efficienza in termini computazionali, sia in termini di spazio occupato.

Durante la realizzazione di questo progetto abbiamo testato diverse varianti del k-opt, individuando quella di cui parleremo tra poco come la migliore tra quelle testate.

Si fa presente che gli algoritmi k-opt sono stati studiati nella teoria nel capitolo 4.6.1 ovvero il capitolo relativo all'algoritmo di iterative best improvement. Questo algoritmo, infatti, come vedremo, ha un comportamento estremamente simile.

1.4.1 Implementazione del k-opt:

Per l'implementazione del k-opt eseguiamo le seguenti operazioni:

- Eseguire una ripetizione dell'algoritmo fornendo in input, come approssimazione iniziale della soluzione, la soluzione fornita dall'algoritmo di Christofides.
- Eseguire più iterazioni dell'algoritmo attraverso l'utilizzo della tecnica del random restart. Questo approccio viene utilizzato dato che, potrebbe capitare che l'algoritmo, durante una sua esecuzione, si blocchi in un minimo locale da cui non è più capace di uscire. Questo approccio è stato studiato nel capitolo 4 del libro.

ALGORITMO DI CHRISTOFIDES:

Prima di proseguire, spieghiamo come funziona l'algoritmo di Christofides. Tale algoritmo è stato utilizzato come approssimazione iniziale poiché la soluzione ottenuta dall'algoritmo di Christofides non è sempre ottimale, ma è dimostrato che produce soluzioni con un rapporto di approssimazione di al massimo 3/2 rispetto alla soluzione ottimale del TSP. Questo significa che la lunghezza del percorso trovato dall'algoritmo di Christofides è al massimo la metà più lungo del percorso ottimo.

La soluzione fornita dall'algoritmo di Christofides si articola in diversi passaggi:

1. **Creazione di un albero di copertura minimo**: Si parte dall'assunzione che il TSP sia un problema su un grafo completo, dove ogni città è collegata a tutte le altre. Si costruisce un albero di copertura minimo (Minimum Spanning Tree, MST)

utilizzando l’algoritmo di Prim. Questo è preferito all’algoritmo di Kruskal poiché il grafo, essendo completo, è denso. L'albero di copertura minimo è un sottoinsieme degli archi del grafo che collega tutte le città senza formare cicli, con il minimo costo totale.

2. **Individuazione dei nodi di grado dispari**: Una volta ottenuto l'albero di copertura minimo, si individuano i nodi di grado dispari, cioè i nodi che hanno un numero dispari di archi adiacenti.
3. **Calcolo di un matching perfetto minimo**: Si cerca di trovare un matching perfetto minimo tra i nodi di grado dispari.
4. **Creazione di un grafo euleriano**: Si crea un grafo euleriano unendo l'albero di copertura minimo con il matching perfetto minimo trovato al passo precedente. Questo viene fatto aggiungendo gli archi del matching perfetto all’MST e duplicando gli archi del MST dove necessario per formare un grafo in cui ogni nodo ha un grado pari.
5. **Calcolo di un circuito euleriano**: Si trova un circuito euleriano nel grafo euleriano creato. Un circuito euleriano è un percorso che attraversa ogni arco esattamente una volta e ritorna al nodo di partenza.
6. **Riduzione del circuito euleriano a un ciclo hamiltoniano**: Infine, si riduce il circuito euleriano ottenuto a un ciclo hamiltoniano, che è un ciclo che attraversa ogni città esattamente una volta. Questo viene fatto eliminando le ripetizioni delle città visitate, mantenendo solo una visita per città.

Torniamo ora a parlare della nostra implementazione dell’algoritmo.

```
    # nearest neighbour

    for _ in range (1, rep):

        if ans == y:
            break

        if (self.debug):
            print ('\n-----------------')

        r += 1
        perm, initial_cost = self.nearest_neighbor()
        if (self.debug):
            print ('\nTWO OPT:\n')

        path, cost = self.two_opt (perm)

        if (self.debug):
            print ('\nTHREE OPT:\n')

        if ans == y:
            break

        _, cost = self.three_opt (path)

        if cost < ans:
            ans = cost
            print (f'rep: {r+1}, cost: {cost}, perc: {self.env.perc_error(cost, self.problemName)}% --- nuovo minimo trovato')
        else:
            print (f'rep: {r+1}, cost: {cost}, perc: {self.env.perc_error(cost, self.problemName)}%')

        self.migliore = self.env.perc_error (ans, self.problemName)
```

L'algoritmo sopra visualizzato rappresenta la funzione principale che gestisce l'algoritmo di ricerca della soluzione ottimale per problema di ottimizzazione combinatoria mediante il two-opt. L'algoritmo si basa su un approccio iterativo che sfrutta tre euristiche: il nearest neighbour, il two-opt ed il three-opt.

Funzionamento:

- **Iterazioni**: L'algoritmo viene eseguito per un numero predefinito di iterazioni.
- **Stima iniziale**: In ogni iterazione, viene generata una stima iniziale della soluzione utilizzando l'algoritmo del nearest neighbour o Christofides.
- **Miglioramento con two-opt**: Successivamente, si applica l'algoritmo two-opt alla stima iniziale per migliorarla e avvicinarla alla soluzione ottimale.
- **Verifica di ottimalità**: Al termine dell'algoritmo two-opt, si verifica se la soluzione trovata è quella ottimale.
- **Fuga dai minimi locali (opzionale)**: Se la soluzione non è ottimale, si presume che l'algoritmo two-opt sia rimasto intrappolato in un minimo locale. In questo

caso, si applica l'algoritmo three-opt per cercare di uscire dal minimo locale e ottenere una soluzione migliore.

Motivazioni:

Nearest neighbour: L'algoritmo del nearest neighbour è stato scelto per la sua semplicità e velocità nel generare una stima iniziale della soluzione.

Christofides: Questo algoritmo, anche se più lento rispetto al nearest neighbour, permette di avere con certezza una stima sufficientemente buona del percorso iniziale su cui eseguire il nostro algoritmo

Two-opt: L'algoritmo two-opt è stato scelto per la sua efficienza nel convergere verso la soluzione ottimale.

Three-opt (opzionale): L'algoritmo three-opt è stato aggiunto come opzione per aumentare la probabilità di uscire dai minimi locali e ottenere una soluzione migliore.

1.4.2 Algoritmo two-opt

Proseguiamo ora vedendo qual è stata la nostra implementazione dell'algoritmo two-opt.

```
def two_opt (self, path):

    curr = path
    cost = self.calculate_cost(path)

    num_nodes = self.graph.number_of_nodes()

    for it in range (self.iterations):

        idxs = sorted(random.sample(range(1, num_nodes), 2))
        i, j = idxs[0], idxs[1]

        a, b, c, d = curr[i], curr[j], curr[i+1], curr[j+1]
        delta = self.graph[c][d]['weight'] + self.graph[a][b]['weight'] - self.graph[a][c]['weight'] - self.graph[b][d]['weight']

        if delta < 0:
            curr[i+1:j+1] = list(reversed(curr[i+1:j+1]))
            cost += delta
            if (self.debug):
                print (it, cost)

    return curr, cost
```

Nel nostro contesto, per ottimizzare al massimo il tempo computazionale dell'algoritmo risolutivo per il problema del commesso viaggiatore, abbiamo scelto di non adottare l'algoritmo two-opt classico, bensì di utilizzare una sua variante chiamata "delta two-opt". Questa variante, pur mantenendo il funzionamento generale dell'algoritmo two-opt, introduce un'ottimizzazione significativa nel calcolo della distanza del nuovo percorso generato, consentendo di risparmiare tempo computazionale. Questa soluzione è applicabile, dato che sappiamo che i grafi messi a disposizione dalla libreria tsplib95 sono grafi completi e simmetrici.

Il funzionamento specifico dell'algoritmo delta two-opt è il seguente:

1. **Iterazioni**: L'algoritmo itera per un numero prefissato di volte. Ad ogni iterazione, vengono selezionati casualmente due indici distinti all'interno del vettore che rappresenta il percorso del commesso viaggiatore.
2. **Scambio degli archi**: Gli indici selezionati indicano i due nodi del percorso che saranno scambiati. Si effettua quindi uno scambio tra gli archi adiacenti a questi nodi.
3. **Calcolo della nuova distanza**: Dopo lo scambio degli archi, viene calcolata la nuova distanza del percorso. Il calcolo della nuova distanza avviene in tempo costante O(1) grazie all'utilizzo di un'implementazione basata sul calcolo delta. Quest'ultimo viene definito come segue:

$$distanza(i) = distanza(i-1) + delta$$

$$dove\ delta \text{ è } calcolato\ come:$$

$$delta = cost(< i, i+1 >) + cost(< j, j+1 >) - cost(< i, j+1 >) + cost(< j, i+1 >)$$

   con i e j che rappresentano gli indici dei nodi che sono stati scambiati, <i,j> rappresenta l'arco tra i nodi i e j, e costo(<i,j>) rappresenta il costo associato all'arco tra i nodi i e j.
4. Lo scambio viene effettuato solo se il valore di delta è minore di zero, indicando che la nuova soluzione proposta è migliore della precedente.

Utilizzando questo approccio, l'algoritmo delta two-opt ottimizza il tempo computazionale associato all'ottimizzazione del percorso del commesso viaggiatore, garantendo che solo le soluzioni che comportano una riduzione nella lunghezza del percorso vengano accettate. Questo approccio può essere visto come una variante dell'iterative best improvement per la ricerca dei minimi locali.

1.4.3 Algoritmo three-opt

```python
def three_opt (self, path):

    self.curr = path
    cost = self.calculate_cost(path)

    num_nodes = self.graph.number_of_nodes()

    for it in range (self.iterations):

        idxs = sorted(random.sample(range(1, num_nodes), 3))
        i, j, k = idxs[0], idxs[1], idxs[2]

        for case in range (4):

            new_solution = self.swap_three (i, j, k, case)
            new_cost = self.calculate_cost(new_solution)

            if new_cost < cost:
                self.curr = new_solution[:]
                cost = new_cost
                if (self.debug):
                    print (it, cost)

    return self.curr, cost
```

```python
def swap_three (self, i, j, k, case):

    if case == 0:
        newtour = self.curr[:i+1]
        newtour.extend(self.curr[j+1:k+1])
        newtour.extend(reversed(self.curr[i+1:j+1]))
        newtour.extend(self.curr[k+1:])

    elif case == 1:
        newtour = self.curr[:i+1]
        newtour.extend(reversed(self.curr[j+1:k+1]))
        newtour.extend(self.curr[i+1:j+1])
        newtour.extend(self.curr[k+1:])

    elif case == 2:
        newtour = self.curr[:i+1]
        newtour.extend(reversed(self.curr[i+1:j+1]))
        newtour.extend(reversed(self.curr[j+1:k+1]))
        newtour.extend(self.curr[k+1:])

    elif case == 3:
        newtour = self.curr[:i+1]
        newtour.extend(self.curr[j+1:k+1])
        newtour.extend(self.curr[i+1:j+1])
        newtour.extend(self.curr[k+1:])

    return newtour
```

L'algoritmo three-opt viene utilizzato per raffinare ulteriormente la soluzione ottenuta dall'algoritmo two-opt. Il funzionamento è il seguente:

Iterazioni: L'algoritmo itera per un numero specificato di volte, il quale è fornito in input alla funzione. Ad ogni iterazione, vengono selezionati casualmente tre indici dall'insieme degli indici disponibili per il percorso del commesso viaggiatore.

Scambio degli archi: Dopo aver selezionato gli indici, viene eseguito uno scambio di archi utilizzando la funzione `swap_three`. Questa funzione ottimizza i calcoli dell'algoritmo - three-opt gestendo quattro casi di scambio. In teoria, ci sarebbero 2^3 possibili scambi per i tre elementi. Tuttavia, considerando che una permutazione non comporta alcuno scambio e che

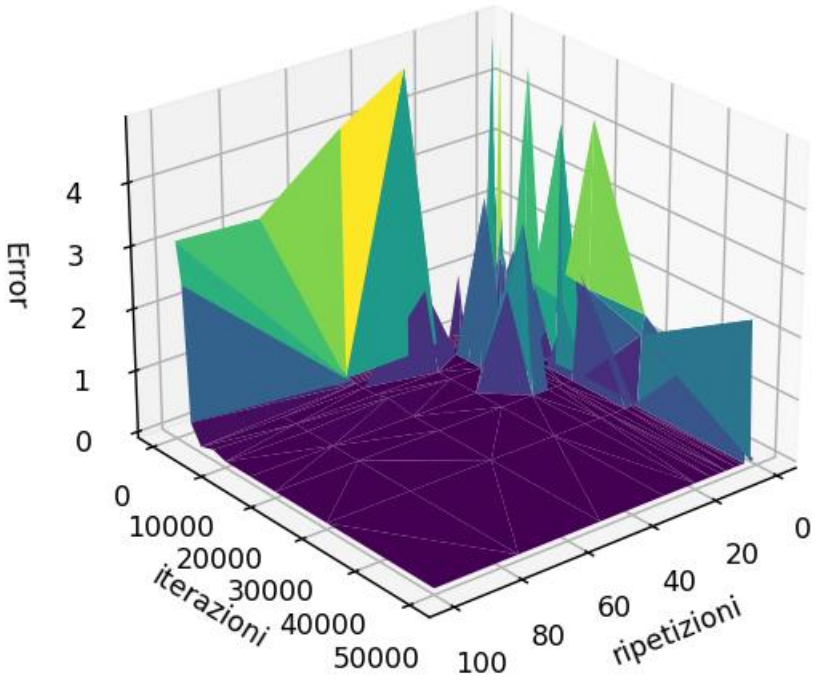
alcune permutazioni potrebbero essere state già esplorate dal two-opt, ci restano solo questi quattro casi. L’esclusione degli altri casi comporta un dimezzamento delle operazioni da svolgere, e quindi un raddoppio della velocità di esecuzione.

Valutazione delle soluzioni: Le soluzioni ottenute tramite la funzione `swap_three` vengono confrontate con la migliore soluzione attualmente individuata. Se una delle nuove soluzioni è migliore di quella precedentemente memorizzata, questa viene conservata come migliore soluzione.

L'approccio dell'algoritmo three-opt permette di raffinare la soluzione del two-opt esplorando una serie di scambi di archi più complessi.

1.4.4 Scelta dei parametri

In questo caso, sono stati effettuati i medesimi test effettuati sull’algoritmo genetico, ottenendo i seguenti risultati:



Come si può osservare, in questo caso il grafico tende ad avere un valore di error molto più basso rispetto a quello dell'algoritmo genetico.

Al termine di questa analisi grafica siamo andati a scegliere i seguenti parametri:

- Ripetizioni: 20
- Iterazioni: 30000

Questi parametri garantiscono un'alta velocità di esecuzione e l'ottimalità della soluzione per i grafi di dimensione non eccessivamente elevata.

1.5 Cluster k-opt

Questo algoritmo rappresenta una valida alternativa al k-opt, dato che oltre ad implementare gli algoritmi k-opt discussi nella sezione precedente, implementa anche un

meccanismo di individuazione di una prima soluzione parziale che risulta essere estremamente affidabile.

Riduzione della complessità dei sotto-problemi: Questo algoritmo opera una suddivisione del grafo originale in sotto-grafi, basata sulla vicinanza degli elementi. Questo approccio consente di ridurre notevolmente la complessità dell'algoritmo, poiché si concentra sul trattamento di sotto-problemi più piccoli e gestibili.

Semplice calcolo dei cluster: L'individuazione dei cluster non è un'operazione complessa in termini di computazione. Questo significa che il processo per identificare e creare i cluster non richiede uno sforzo computazionale significativo. Di conseguenza, questa fase non influisce in modo sostanziale sul tempo di esecuzione complessivo dell'algoritmo.

1.5.1 Implementazione del cluster three opt

1.5.1.1 Implementazione della funzione k-means

```
def kmeans( graph,k):
    centroids = random.sample(
        list(graph.nodes), k)  # indici dei centroidi
    clusters = helper_agent.assign_clusters(graph = graph, k = k, centroids = centroids)
    for i in range(len(clusters)):
        print('cluster', i, 'ha lunghezza', len(clusters[i]))
    return clusters
```

La prima operazione che siamo andati a svolgere è stata quella di creare la funzione k-means, che è progettata per generare i k cluster (con k come parametro della funzione) basandosi sulla disposizione degli elementi rispetto ai centroidi. Ecco come funziona il processo:

- **Inizializzazione dei centroidi**: I centroidi vengono scelti casualmente dall'insieme dei nodi. Mediante la funzione random sample, andiamo a

prendere k indici dall'insieme dei nodi. Gli elementi associati a quell'indice diventeranno i nostri centroidi.

- **Assegnazione dei nodi ai cluster**: Successivamente, utilizziamo la funzione assign_cluster per calcolare la distanza di ogni nodo rispetto a ciascun centroide e per assegnare ogni nodo al cluster più vicino.
- **Risoluzione del problema sui singoli cluster mediante three – opt**: A questo punto abbiamo applicato l'algoritmo three-opt ad ogni cluster. Questo approccio nella maggioranza dei casi non permette di raggiungere la soluzione ottimale, ma comunque la soluzione trovata è molto spesso una buona stima.
- **Raffinamento della soluzione mediante two-opt e three-opt:** La soluzione trovata al passo precedente, viene ulteriormente raffinata mediante l'applicazione del two-opt e del three-opt sul grafo totale, le cui caratteristiche sono state esplicitate nei paragrafi precedenti.

1.5.1.2 Implementazione della funzione k-means

```
# Definizione della funzione per assegnare i punti ai cluster
def assign_clusters(graph, k, centroids):
    clusters = [[] for _ in range(k)]
    for node in graph.nodes:
        distances = [helper_agent.distance(graph, node, centroid)
                     for centroid in centroids]
        cluster = np.argmin(distances)
        clusters[cluster].append(node)
    return clusters
```

La funzione assign cluster viene utilizzata per l'assegnazione dei nodi che non sono centroidi ai cluster.

Per farlo, inizializziamo k liste vuote che corrispondono alle k liste dei cluster che abbiamo intenzione di generare. A questo punto, andiamo ad iterare su tutti i nodi ed assegniamo ogni nodo al cluster il cui centroide è più vicino.

Al termine di questa procedura abbiamo generato i k cluster a cui sono stati assegnati i relativi nodi.

1.5.2 Implementazione dell'hybrid three-opt

```
    for r in range(rep):
        _, cost = self.hybrid_three_opt(self.graph)
        ans = min(ans, cost)
        print(helper_agent.calculate_cost(self.graph, _))
        print('rep:', r+1, ', cost:', cost, ', perc:',
            helper_agent.perc_error(cost, self.problemName, self.env), '%')

    helper_agent.print_solution(ans, self.env, problemName=self.problemName)

def hybrid_three_opt(self, graph):

    optimized_paths = []

    for cluster in self.clusters:
        if len(cluster) > 3:
            subgraph = graph.subgraph(cluster)
            print('cluster', self.clusters.index(cluster)+1)
            optimized_path, _ = self.three_opt(
                subgraph, list(subgraph.nodes()))
            optimized_paths.append(optimized_path)
        else:
            optimized_paths.append(cluster)

    final_path = list(itertools.chain.from_iterable(optimized_paths))
    final_path.append(final_path[0])

    path, _ = self.two_opt(self.graph, final_path)
    return self.three_opt(graph, path)
```

Una volta terminate le operazioni di creazione dei cluster, andiamo a richiamare la funzione hybrid three opt. Questa funzione viene richiamata "r" volte e, per ognuna di queste chiamate, vengono svolte le seguenti operazioni:

- **Ricerca di soluzioni per i cluster:** per prima cosa andiamo a risolvere il tsp sui cluster che abbiamo individuato. La soluzione per ogni cluster viene individuata mediante l'utilizzo del three-opt.

- **Merge delle soluzioni parziali trovate:** al termine del passo precedente andiamo ad unire le soluzioni parziali trovate, in modo tale da creare un path unico su cui operare.
- **Raffinamento delle soluzioni mediante k-opt**: la soluzione trovata viene ri-elaborata sul grafo totale mediante l'algoritmo two-opt e three-opt visti in precedenza.

Motivazioni per cui è stato scartato

Questo algoritmo è stato scartato dato che non è in grado di fornire soluzioni in tempi comparabili con quelli dell'algoritmo genetico e del k-opt. La lentezza di questo algoritmo è legata alla necessità di svolgere il k-opt sia sui cluster che sul path generato dal merge dei cluster.

Ulteriori alternative testate:

Durante lo svolgimento dei vari test sugli algoritmi, abbiamo anche testato l'utilizzo di una neural network per la valutazione del nearest neighbour di un nodo.
L'implementazione utilizzata è la seguente:

1. Generazione di grafi randomici per l'addestramento: Per prima cosa siamo andati a generare un set di grafi networkx randomici, con numero di nodi uguale al numero di nodi del grafo che stiamo considerando. Una volta generati i grafi siamo andati a splittarli in proporzione 80/20 in train e test.

```python
# Funzione per generare grafi casuali come dati
def generate_random_graph(self):
    G = nx.complete_graph(range(self.num_nodes))
    for node1 in G.nodes:
        for node2 in G.nodes:
            if node1 != node2:
                G[node1][node2]['weight'] = random.randint(1, 50)
    return G

# Funzione per generare il dataset di addestramento e test
def generate_dataset(self):
    X_train = []
    y_train = []
    X_test = []
    y_test = []

    for _ in range(self.num_graphs):
        G = self.generate_random_graph(self.num_nodes)
        nodes = list(G.nodes())
        random.shuffle(nodes)

        # Calcola le distanze tra i nodi
        distances = {}
        for i in range(self.num_nodes):
            for j in range(self.num_nodes):
                if i != j:
                    distances[(i, j)] = G[i][j]['weight']
                    distances[(j, i)] = G[j][i]['weight']
                else:
                    distances[(i, i)] = 500

        # Genera il training set
        for i in range(self.num_nodes):
            distances_from_current_node = [
                distances[(i, j)] for j in range(self.num_nodes)]
            X_train.append(distances_from_current_node)
            y_train.append(np.argmin(distances_from_current_node))

        # Genera il test set
        start_node = random.choice(list(G.nodes()))
        distances_from_start_node = [
            distances[(start_node, j)] for j in range(self.num_nodes)]
        X_test.append(distances_from_start_node)
        y_test.append(np.argmin(distances_from_start_node))

    return (
        torch.tensor(X_train).float(),
        torch.tensor(y_train).long(),  # Cambio il tipo di dato a 'long'
        torch.tensor(X_test).float(),
        torch.tensor(y_test).long()  # Cambio il tipo di dato a 'long'
    )
```

2. Abbiamo quindi addestrato il modello della neural network sui grafi di test.

   L'addestramento è stato eseguito mediante l'utilizzo dei seguenti parametri, individuati dopo diversi test in cui abbiamo fatto uso del costrutto grid di python:

```python
# Definizione dei possibili valori degli iperparametri
param_grid = {
    'hidden_size': [64],
    'lr': [0.01],
    'num_epochs': [1000]
}
```

Come criterio di valutazione abbiamo deciso di utilizzare la cross-entropy Loss, dato che l'addestramento del modello con questo criterio di loss tendeva a far migliorare l'accuracy dei risultati ottenuti.

```
    for params in ParameterGrid(param_grid):
        model = NeuralNetwork(
            input_size=X_train.shape[1], hidden_size=params['hidden_size'], output_size=self.num_nodes)
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=params['lr'])
        self.num_epochs = params['num_epochs']
```

3. Abbiamo quindi reiterato il processo di addestramento per un totale di 100 epoch.

```
            if epoch % 100 == 0:
                print(
                    f'Epoch [{epoch}/{self.num_epochs}], Loss: {loss.item()}')

        # Valutazione sul set di test
        with torch.no_grad():
            correct = 0
            total = 0
            predicted_nodes = model(X_test)
            for i in range(len(X_test)):
                predicted_node_index = predicted_nodes[i].argmax().item()
                if predicted_node_index == y_test[i]:
                    correct += 1
                total += 1

            accuracy = correct / total
            print(f"Parameters: {params}, Accuracy: {accuracy*100}%")

            # Aggiornamento dei migliori iperparametri
            if accuracy > best_accuracy:
                best_accuracy = accuracy
                best_params = params

    print(
        f"Best parameters: {best_params}, Best accuracy: {best_accuracy}")

    return model
```

4. Abbiamo creato una classe NeuralNetwork che va ad addestrare il modello che abbiamo creato ed ad effettuare le predizioni sullo scenario fornito.

```
class NeuralNetwork(nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(NeuralNetwork, self).__init__()
        self.hidden = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.hidden(x)
        x = self.relu(x)
        x = self.output(x)
        return x
```

I risultati ottenuti sono stati estremamente soddisfacenti, in quanto abbiamo ottenuto un'accuracy superiore al 90% sul test set.

Perché è stata scartata:

Come abbiamo detto, i risultati ottenuti sono ottimi, ma il tempo di addestramento ed il tempo necessario per effettuare la predizione costituivano un rallentamento importante per i nostri algoritmi. Infatti, il tempo richiesto per l'addestramento era troppo elevato, e la predizione richiedeva poco meno di 10 secondi per predire un percorso completo con grafi di dimensione importante. Ovviamente, l'utilizzo di questa alternativa durante le iterazioni dei nostri algoritmi rappresentava un ostacolo in quanto questa procedura doveva essere ripetuta n volte, con n numero di ripetizioni dell'algoritmo che la utilizzava.

Valutazione delle runtime distribution:

Dopo le considerazioni effettuate nei paragrafi precedenti, abbiamo riscontrato che i migliori algoritmi sono il k-opt e il genetico per i seguenti motivi:

- Il k-opt è riuscito a fornire sempre un risultato accettabile, e per grafi sotto i 100 nodi è quasi sempre ottimale. Inoltre, anche con dei test su grafi da

2000/3000 nodi, è riuscito a fornirci risultati sub-ottimali (sotto il 5% di errore).

- Il genetico, diversamente, non ha ottenuto le stesse prestazioni, ma risulta comunque una valida alternativa nel caso in cui si considerino grafi con meno di 100 nodi, o si cerchi la sub-ottimalità.

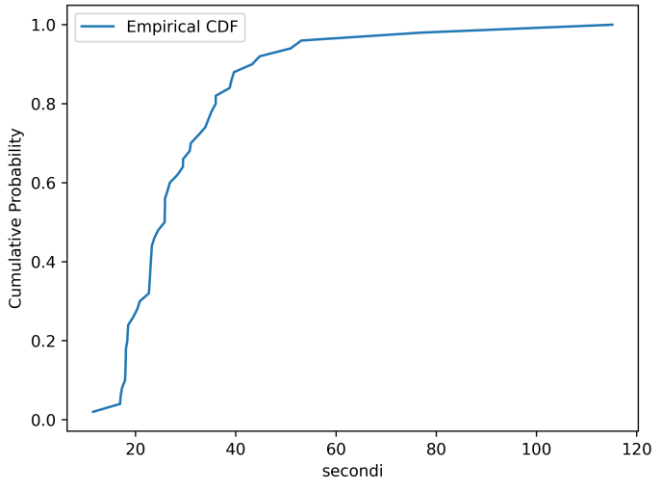
Per i motivi sopra citati, abbiamo effettuato dei test per individuare le runtime distribution dei due algoritmi. Siamo andati a effettuare i test sia per la soglia di sub-ottimalità, sia per l'ottimalità (quest'ultima solo per il k-opt).

Abbiamo individuato il grafico della runtime distribution di tre grafi della libreria tsplib95, dato che questi rispecchiano delle istanze in cui si può incorrere quando ci si interfaccia con la risoluzione del tsp. I tre grafi utilizzati in questa fase sono stati scelti dato che: il grafo Ulysses22 rappresenta un grafo con 22 nodi, quindi un grafo per cui la soluzione deve essere veloce ed ottimale. Il grafo Berlin52 rappresenta un grafo con 52 nodi, quindi un grafo per cui non ci aspettiamo necessariamente una soluzione ottimale, ma che i tempi di computazione siano contenuti e che una soluzione sub-ottimale sia trovata. Il grafo Bier127 rappresenta un grafo di grandi dimensioni, avendo 127 nodi, e per cui gli algoritmi devono trovare una soluzione sub-ottimale in tempi più dilatati.

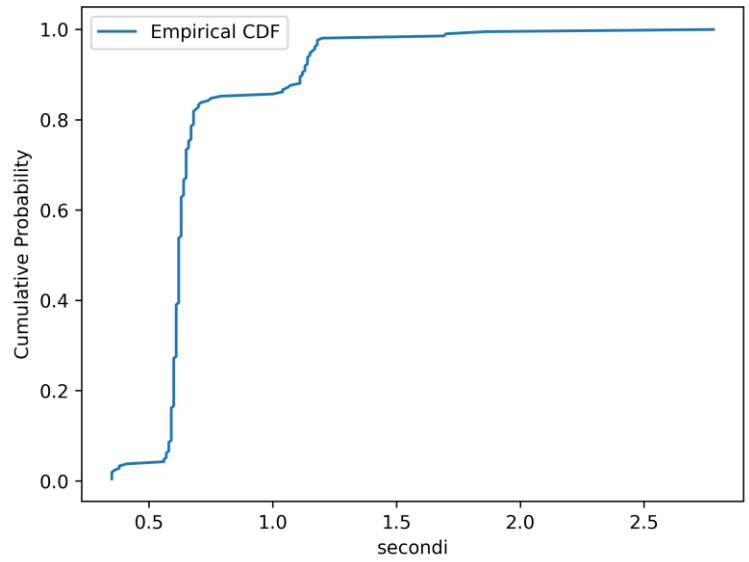
Runtime del genetico

Come abbiamo detto nel paragrafo precedente la runtime distribution dell'algoritmo genetico è stata calcolata per il fattore di sub-ottimalità (5%) ottenendo i seguenti risultati:

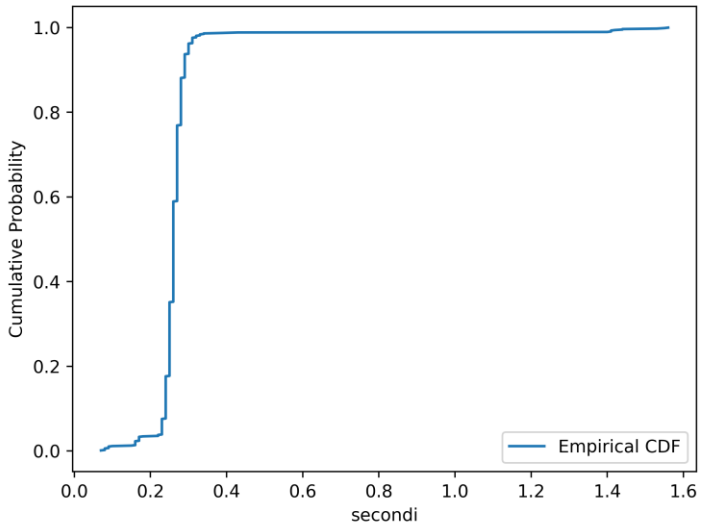
- Bier127: si può notare che, in più del 90% dei casi, si è trovata una soluzione sub-ottimale entro gli 80 secondi.



- Berlin52: si può notare che in questo caso in più dell'80% dei casi si è trovata una soluzione sub-ottimale entro il secondo.



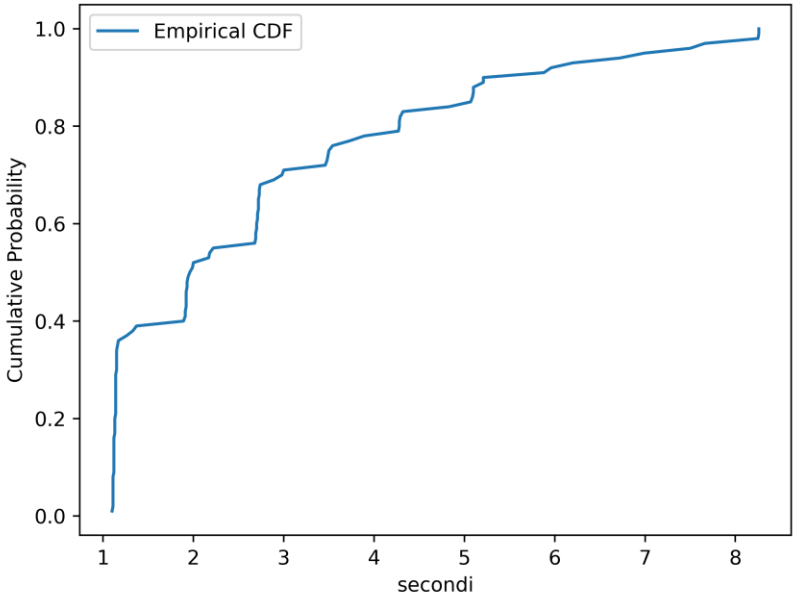
- Ulysses22: si può notare che, in questo caso, in quasi il 100% dei casi si è trovata una soluzione sub-ottimale entro il secondo.



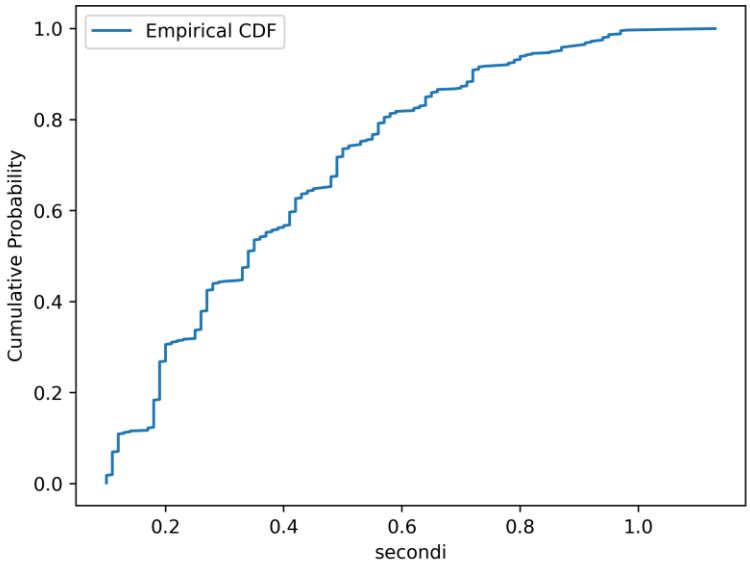
Runtime del k-opt:

In questa sezione andiamo ad analizzare la runtime distribution del k-opt. Questo algoritmo è quello che poi abbiamo utilizzato per lo svolgimento del progetto finale.

- Bier127: si può notare che in questo caso nel 100% dei casi si è trovata una soluzione sub-ottimale entro gli 8 secondi.



- Berlin52: si può notare che, in questo caso, nel 100% dei casi si è trovata una soluzione sub-ottimale entro il secondo.



- Ulysses22: si può notare che, in questo caso, in quasi il 100% dei casi si è trovata una soluzione sub-ottimale entro i 0,20 secondi.

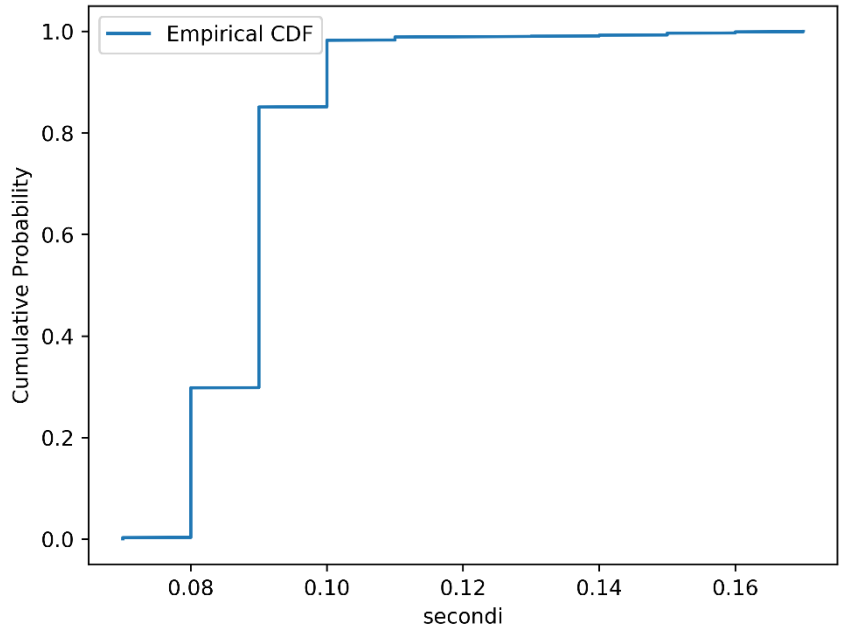


Tabella dei risultati del k-opt su diversi grafi:

Abbiamo quindi provveduto ad effettuare diversi test su grafi di diversa dimensione, e siamo andati ad analizzare i risultati ottenuti nel caso medio e nel caso migliore.

Nome grafo	Medio	Migliore
ulysses22	0,00	0,00
berlin52	0,00	0,00
eil76	0,56	0,00
rat99	0,40	0,08
lin105	0,48	0,26
bier127	2,20	1,17
u159	4,20	2,46
pr226	2,60	1,40
gr431	3,84	3,14
si1032	0,30	0,16
vm1748	4,12	3,79

Come possiamo notare, l'algoritmo riesce addirittura a trovare la soluzione ottimale per grafi di circa 100 nodi, superando le prestazioni previste anche paragonandole a quelle delle ricerche con cui abbiamo effettuato dei confronti. In caso di grafi con particolari caratteristiche, come il grafo si1032, avente più di 1000 nodi e più di 1 milione di archi, siamo riusciti a trovare una soluzione quasi ottimale.

Algoritmo finale

Sulla base delle valutazioni derivanti dei risultati trovati nei precedenti paragrafi l'algoritmo che abbiamo deciso di utilizzare è stato il k-opt. La decisione è legata ai seguenti fattori:

- Tempi di esecuzione migliori rispetto agli altri algoritmi.

- Questo approccio ci ha permesso di risolvere in tempi contenuti un’istanza della libreria tsplib95 che aveva circa 1000 nodi.
- Rispetto all’algoritmo genetico è più efficiente in spazio occupato.

Una volta terminata la fase di test, ci siamo quindi dedicati alla creazione di un caso reale in cui utilizzare il nostro algoritmo.

Creazione del set di dati

Per la creazione del set dei dati, abbiamo seguito le indicazioni fornite nella fase di individuazione dello scenario. Abbiamo quindi simulato la situazione meteorologica, quella stradale e l'eventuale presenza di lavori in corso sulla strada. Queste operazioni sono state svolte mediante un algoritmo in Python che attribuiva randomicamente i valori influenzanti alle strade.

I valori meteorologici non sono stati attribuiti per ogni città, dato che abbiamo tenuto in considerazione che a volte questi dati potrebbero non essere disponibili per alcune città. Per ricavare questi dati, abbiamo utilizzato un algoritmo di k-Nearest Neighbors (k-NN) implementato in Prolog. Questo algoritmo attribuisce a una città di cui non si conosce la situazione meteorologica il valore più appropriato sulla base della vicinanza di altre città di cui conosciamo la situazione.

Le situazioni in cui ci possiamo trovare sono due:

1. Tra le k città vicine, si ha una condizione meteorologica dominante: In questo caso, attribuiamo alla città il valore più comune tra le k città più vicine di cui la situazione meteorologica è nota.
2. Tra le k città vicine, si hanno più condizioni meteorologiche dominanti: ovvero abbiamo delle condizioni meteorologiche attribuite in egual numero alle k città. In

questo caso, alla città presa in considerazione attribuiamo la peggiore tra le due situazioni, cioè quella che influenza di più la percorrenza della strada.

IMPLEMENTAZIONE DEL CODICE PROLOG

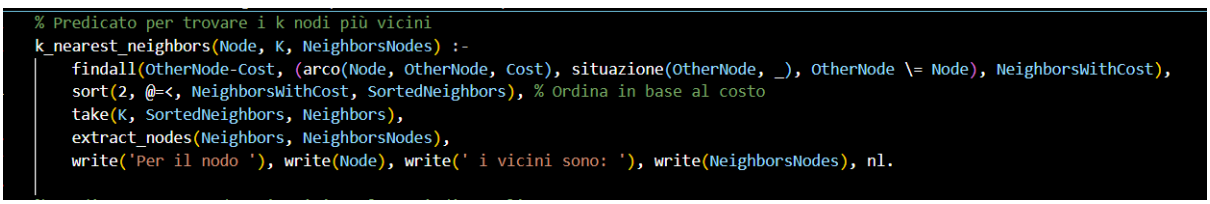
Adesso andiamo a vedere in maniera dettagliata la nostra implementazione del codice in Prolog.

Il nostro codice Prolog preleva i fatti da un file chiamato “predici.pl”. Questo file contiene una lista di fatti che sono stati pre-elaborati in python. Nello specifico, in python, andiamo a leggere i dati riguardanti le città italiane e siamo andati a generare due tipologie di fatti:

- Arco(nodo1, nodo2, costo): questo fatto definisce l’individuo “arco”. Questo ha come termini:
  - Nodo1: primo nodo coinvolto nell’arco.
  - Nodo2: secondo nodo coinvolto nell’arco
  - Costo: costo dell’arco che porta da nodo1 a nodo2.
- Situazione(Nodo, situazione metereologica): Questo fatto definisce l’individuo “situazione metereologica”. I termini di questo fatto sono i seguenti:
  - Nodo: città di cui si vuole specificare la situazione metereologica.
  - Situazione metereologica: ovvero il valore della situazione della città.

Vediamo ora le regole che abbiamo realizzato:

k_nearest_neighbors



```
% Predicato per trovare i k nodi più vicini
k_nearest_neighbors(Node, K, NeighborsNodes) :-
    findall(OtherNode-Cost, (arco(Node, OtherNode, Cost), situazione(OtherNode, _), OtherNode \= Node), NeighborsWithCost),
    sort(2, @=<, NeighborsWithCost, SortedNeighbors), % Ordina in base al costo
    take(K, SortedNeighbors, Neighbors),
    extract_nodes(Neighbors, NeighborsNodes),
    write('Per il nodo '), write(Node), write(' i vicini sono: '), write(NeighborsNodes), nl.
```

La funzione k_nearest_neighbors(Node, K, NeighborsNodes) calcola i K vicini più vicini ad un nodo specificato in un grafo. La funzione è strutturata nel seguente modo:

- k_nearest_neighbors(Node, K, NeighborsNodes) :- definisce un predicato con tre argomenti: Node (il nodo di partenza), K (il numero di vicini da trovare), e NeighborsNodes (una variabile di output che conterrà i nodi vicini trovati).
- findall(OtherNode-Cost, (arco(Node, OtherNode, Cost), situazione(OtherNode, _), OtherNode \= Node), NeighborsWithCost), trova tutti i nodi OtherNode che sono collegati al nodo di partenza con un arco e che soddisfano la condizione situazione(OtherNode, _). Crea una lista di coppie OtherNode-Cost, dove Cost è il costo dell'arco tra Node e OtherNode. Questa lista è assegnata alla variabile NeighborsWithCost.
- sort(2, @=<, NeighborsWithCost, SortedNeighbors), ordina la lista NeighborsWithCost in base al secondo elemento di ciascuna coppia (cioè il costo), in ordine crescente. La lista ordinata è assegnata alla variabile SortedNeighbors.
- take(K, SortedNeighbors, Neighbors), prende i primi K elementi dalla lista SortedNeighbors e li assegna alla variabile Neighbors.
- extract_nodes(Neighbors, NeighborsNodes), estrae i nodi dalla lista Neighbors (che contiene coppie di nodi e costi) e li assegna alla variabile NeighborsNodes.
- write('Per il nodo '), write(Node), write(' i vicini sono: '), write(NeighborsNodes), nl. stampa un messaggio che indica i nodi vicini trovati per il nodo di partenza, utile per il debug.

extract_node:

```
extract_nodes([], []).
extract_nodes([Node-_ | Rest], [Node | Nodes]) :-
    extract_nodes(Rest, Nodes).
```

il predicato extract_nodes viene utilizzato per estrarre le coppie nodo – costo. Il suo funzionamento è il seguente:

- extract_nodes([], []). è il caso base della ricorsione. Quando la lista di input è vuota, la lista di output sarà anch'essa vuota.
- extract_nodes([Node-_ | Rest], [Node | Nodes]) :- extract_nodes(Rest, Nodes). è il caso ricorsivo. Questa clausola viene richiamata quando la lista di input non è vuota. Il primo elemento della lista di input è una coppia Node-_, dove Node è il nodo e _ è il costo, che viene ignorato. Il resto della lista di input è Rest. Il primo elemento della lista di output è Node, e il resto della lista di output è Nodes. Quindi, questa clausola prende il primo nodo dalla lista di input, lo aggiunge alla lista di output, e poi chiama se stessa ricorsivamente sul resto della lista di input.

Predict_weather

```
predict_weather(Node, K, Weather) :-
    k_nearest_neighbors(Node, K, Neighbors),
    find_most_common_weather(Neighbors, MostCommonWeather),
    secondo_elemento(MostCommonWeather, Weather).
```

Questa regola viene utilizzata per predire la situazione metereologica di un dato nodo basandosi sui k vicini. Nello specifico questo nodo richiama il predicato k_nearest_neighbors per estrarre i k nodi più vicini e da questi risalire alla situazione metereologica. Come possiamo notare questo metodo richiama il predicato find_most_common_weather, il quale conta per ogni tipologia di meteo quante volte questo occorre nei k elementi prelevati. Questa

operazione viene effettuata mediante il conteggio effettuato dal predicato conteggio_meteo e aggiorna_conteggio.

Predicato find_most_common_weather:

```
find_most_common_weather(Neighbors, MostCommonWeather) :-
    conteggio_meteo(Neighbors, ListaConteggiMeteo),
    sort(ListaConteggiMeteo, SortedCounts),
    reverse(SortedCounts, DescendingCounts),
    write('Meteo dei vicini e conteggio: '), write(DescendingCounts), nl,
    prioritize_weather(DescendingCounts, MostCommonWeather).
```

Questa regola ha il seguente funzionamento:

- find_most_common_weather(Neighbors, MostCommonWeather) :- definisce un predicato con due argomenti: Neighbors (la lista dei nodi vicini) e MostCommonWeather (una variabile di output che conterrà il tipo di tempo più comune).
- conteggio_meteo(Neighbors, ListaConteggiMeteo), chiama un predicato conteggio_meteo/2 che conta la frequenza di ogni tipo di tempo tra i nodi vicini. Il risultato è assegnato alla variabile ListaConteggiMeteo.
- sort(ListaConteggiMeteo, SortedCounts), ordina la lista ListaConteggiMeteo in ordine crescente e assegna il risultato alla variabile SortedCounts.
- reverse(SortedCounts, DescendingCounts), inverte l'ordine della lista SortedCounts per ottenere una lista in ordine decrescente, che viene assegnata alla variabile DescendingCounts.
- write('Meteo dei vicini e conteggio: '), write(DescendingCounts), nl, stampa un messaggio che indica la frequenza di ogni tipo di tempo tra i nodi vicini.

- prioritize_weather(DescendingCounts, MostCommonWeather) chiama il predicato prioritize_weather/2, che seleziona il tipo di tempo più comune dalla lista DescendingCounts. Il risultato è assegnato alla variabile MostCommonWeather.

prioritize_weather

```
prioritize_weather(Counts, Prioritized) :-
    max_count(Counts, MaxCount),
    filter_max(Counts, MaxCount, Filtered),
    %write(Filtered), nl,
    predsort(compare_priority, Filtered, Prioritized),
    write('Priorita ordinata: '), write(Prioritized), nl.
```

A questo punto, prioritize_weather ordina le situazioni metereologiche dalla peggiore alla migliore, in maniera tale che, in caso di parità di conteggio di situazioni metereologiche, venga preferita la peggiore. Il funzionamento è il seguente:

- Il predicato prioritize_weather/2 riceve in input una lista Counts contenente coppie (conteggio, situazione meteorologica) e restituisce la lista Prioritized
- La lista Prioritized è ordinata in base alla priorità delle situazioni meteorologiche. Per fare ciò, viene prima identificato il conteggio massimo tra le situazioni meteorologiche e poi la lista viene ordinata attraverso un predicato specifico per la nostra situazione, chiamato compare_priority utilizzato nel predsort.

Compare_priority

```
compare_priority(Risultato, (_, Situazione1), (_, Situazione2)) :-
    priority(Situazione1, Priorita1),
    priority(Situazione2, Priorita2),
    %write(Situazione1), nl, write(Situazione2), nl,
    %write(Priorita1), nl, write(Priorita2), nl,
    (   Priorita1 > Priorita2 ->
        Risultato = '<'
    ;   Priorita1 < Priorita2 ->
        Risultato = '>'
    ).
```

Questa regola viene utilizzata per effettuare un ordinamento sulla situazione metereologica a parità di conteggio delle occorrenze. Il funzionamento di questa regola è la seguente:

- Viene chiamato il predicato priority/2 per ottenere il livello di priorità di ciascuna situazione meteorologica (Situazione1 e Situazione2). Questo predicato associa a ogni situazione meteorologica un numero che rappresenta il suo livello di priorità.
- I livelli di priorità ottenuti vengono confrontati: se il livello di priorità della Situazione1 è maggiore di quello della Situazione2, allora Risultato viene unificato con il carattere <, indicando che la Situazione1 deve essere inserita prima nella lista rispetto a Situazione 2. Questo avviene perché si preferisce, secondo le nostre specifiche, un ordinamento in base alla situazione meteorologica peggiore.

Main

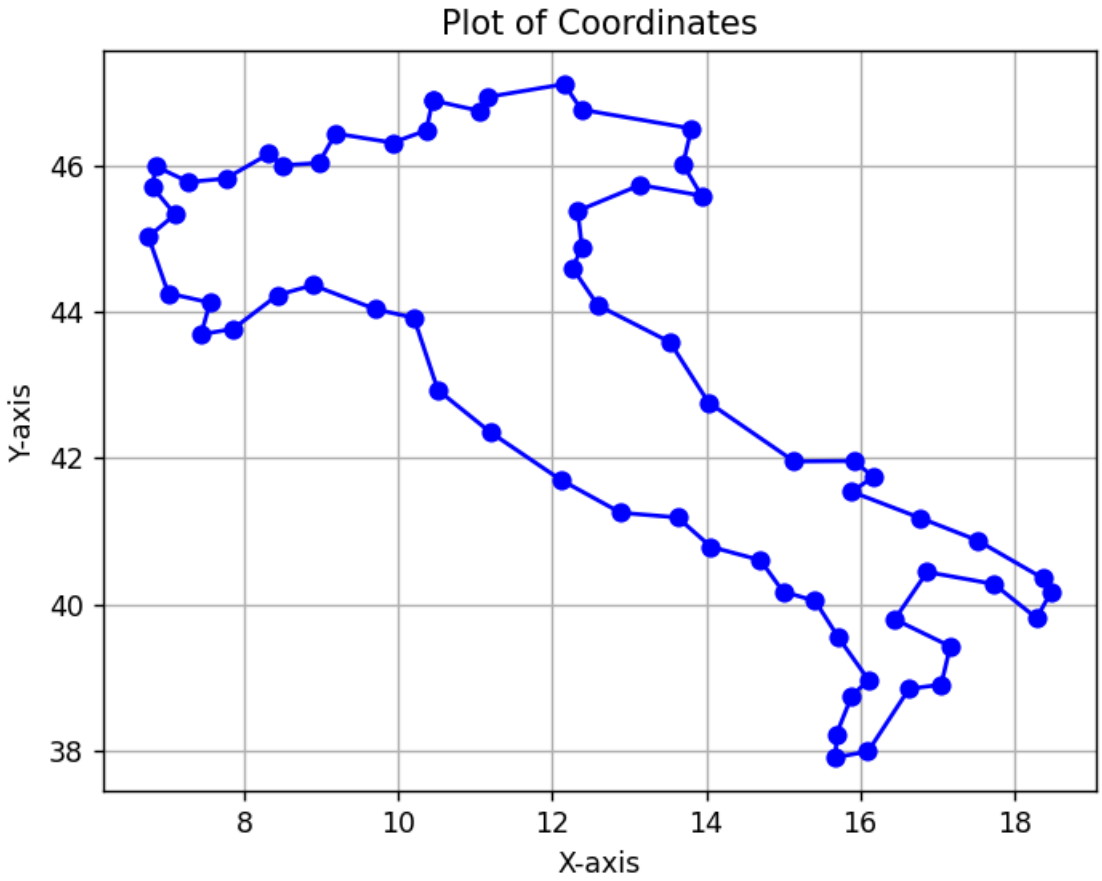
```
main :-
    consult('predici.pl'),
    open('predizione.txt', write, Stream), % Apre il file in modalità di scrittura
    findall(Node, (arco(Node, _, _), \+ situazione(Node, _)), Nodes), % Trova tutti i nodi senza situazione
    list_to_set(Nodes, UniqueNodes), % Rimuove i duplicati
    process_nodes(UniqueNodes, Stream), % Passa lo stream del file come argomento
    close(Stream). % Chiude il file dopo aver scritto tutti i risultati
```

Il predicato main/0 è la parte principale del programma che coordina l'esecuzione complessiva. Ecco come funziona:

- consult('predici.pl'): Questa istruzione carica il file predici.pl che contiene i fatti e le regole necessarie per il funzionamento del programma. Questo file contiene le informazioni iniziali sul grafo e le situazioni meteorologiche dei nodi.
- open('predizione.txt', write, Stream): Questa istruzione apre il file predizione.txt in modalità di scrittura (write) e restituisce uno stream di output (Stream) che sarà utilizzato per scrivere i risultati della previsione.
- findall(Node, (arco(Node, _, _), \+ situazione(Node, _)), Nodes): Questa istruzione trova tutti i nodi della rete che hanno un arco uscente, ma non hanno ancora una situazione meteorologica definita. I nodi trovati vengono memorizzati nella lista Nodes.
- list_to_set(Nodes, UniqueNodes): Questa istruzione rimuove i duplicati dalla lista Nodes, creando una nuova lista UniqueNodes che contiene solo i nodi unici.
- process_nodes(UniqueNodes, Stream): Questa istruzione chiama il predicato process_nodes/2 passando la lista dei nodi unici (UniqueNodes) e lo stream di output (Stream) come argomenti. Questo avvierà il processo di previsione della situazione meteorologica per ciascun nodo e la scrittura dei risultati sul file.
- close(Stream): Questa istruzione chiude lo stream del file dopo che tutti i risultati sono stati scritti con successo.

Tsp solver.

A questo punto, una volta terminata l'esecuzione del programma Prolog, siamo andati a controllare che le strade non fuoriescano al di fuori dei confini. Per farlo abbiamo utilizzato la libreria geopandas, che ci ha permesso di prelevare le coordinate che permettono di definire i confini italiani all'interno di un poligono.



A questo punto, per ogni coppia di nodi, siamo andati a tracciare l'arco tra i due nodi. Se l'arco che unisce i due nodi intercetta il perimetro del poligono che definisce l'Italia, allora il costo di quell'arco viene posto a 2^30, in modo tale da renderlo impercorribile.

Una volta effettuata questa elaborazione, siamo andati quindi a calcolare come gli elementi influenzanti hanno fatto mutare il tempo di percorrenza sulle strade. Nello specifico

abbiamo utilizzato la formula per calcolare il tempo che un'autovettura impiegherebbe per percorrere lo spazio s ad una velocità costante di 90 km/h.

$$t = s * v$$

Ovvero il tempo di percorrenza è dato dallo spazio che intercorre tra due città per la velocità. A questo punto, andiamo a considerare i fattori influenzanti per modificare i tempi di percorrenza degli archi coinvolti.

Quindi, nel nostro caso, i pesi degli archi diventano i tempi di percorrenza delle strade.

La distanza tra due città è stata calcolata mediante il calcolo della distanza geodetica tra i due punti, per tenere in considerazione la curvatura terrestre e rendere i calcoli di distanza più accurati; quindi, il valore di distanza è un valore che rappresenta la distanza in linea d'aria tra i due punti.

Al termine del processo del calcolo delle distanze e la conversione in tempo, siamo andati a moltiplicare tali valori per il fattore influenzante che inficia sui tempi di percorrenza.

```
dissestata = random.choices(population=[0, 1, 2, 3], weights=[0.8, 0.15, 0.04, 0.01], k=n**2)
lavori_in_corso = random.choices (population=[0, 1], weights=[0.95, 0.05], k=n**2)

for i in G.nodes:
    for j in G.nodes:

        if i == j:
            continue

        if G[i][j]['weight'] == 2 ** 30:
            continue

        dissestamento = max(dissestata[i], dissestata[j])
        lavori = max (lavori_in_corso[i], lavori_in_corso[j])
        meteo = max (listaMeteo[i], listaMeteo[j])

        #print (G[i][j]['weight'], type(G[i][j]['weight']))

        if dissestamento == 1:
            G[i][j]['weight'] = float(G[i][j]['weight']) * 1.1
            G[j][i]['weight'] = float(G[j][i]['weight']) * 1.1

        if dissestamento == 2:
            G[i][j]['weight'] = float(G[i][j]['weight']) * 1.2
            G[j][i]['weight'] = float(G[j][i]['weight']) * 1.2

        if dissestamento == 3:
            G[i][j]['weight'] = float(G[i][j]['weight']) * 1.4
            G[j][i]['weight'] = float(G[j][i]['weight']) * 1.4

        if lavori == 1:
            G[i][j]['weight'] = float(G[i][j]['weight']) * 1.5
            G[j][i]['weight'] = float(G[j][i]['weight']) * 1.5

        if meteo == 1:
            G[i][j]['weight'] = float(G[i][j]['weight']) * 1.1
            G[j][i]['weight'] = float(G[j][i]['weight']) * 1.1

        if meteo == 2:
            G[i][j]['weight'] = float(G[i][j]['weight']) * 1.2
            G[j][i]['weight'] = float(G[j][i]['weight']) * 1.2
```

Il resto del programma, ovvero la parte implementativa del calcolo del tsp resta invariata rispetto all'implementazione del two-opt/three-opt visto nel paragrafo destinato agli algoritmi.

Creazione della mappa

```
geolocator = Nominatim(user_agent="city_route")
mappa = folium.Map(location=None, zoom_start=6)

coordinate = []
lista = []

with open ('comuni.csv', 'r', encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        lista.append((row[0], row[1], row[2]))

for p in path:
    for r in lista:
        if r[0] == diz1[p]:
            coordinate.append((float(r[1]), float(r[2])))
            #print (r[0])

folium.PolyLine(locations=coordinate, color='red', weight=5).add_to(mappa)

for i in range(len(coordinate)):
    folium.Marker(coordinate[i], popup=citta[i]).add_to(mappa)

folium.Marker(coordinate[0], popup=citta[0]).add_to(mappa)

mappa.fit_bounds([[36.6, 6.2], [47.0, 19.0]])
mappa.save('city_route_map.html')
```

Per migliorare la visualizzazione dei risultati, abbiamo optato per l'impiego della libreria Python "Folium". Questa libreria consente di rappresentare punti sulla mappa utilizzando le relative coordinate geografiche. Inoltre, offre la possibilità di collegare tali punti mediante linee, aspetto che abbiamo sfruttato per tracciare il percorso del commesso

viaggiatore. Di seguito è riportato il risultato ottenuto al termine della nostra elaborazione:

Quella a sinistra è la soluzione individuata al problema senza l'applicazione di modifiche legate alle condizioni che alterano la percorribilità dei tratti stradali, mentre a destra è rappresentata quella in cui sono presenti i fattori influenzanti.

Possiamo notare che il percorso non si discosta di tantissimo, ma sono presenti comunque delle variazioni.

Lasciamo in allegato i file city_route_map_with_weight (file con fattori influenzanti) e city_route_map_pure (senza fattori influenzanti).

La visualizzazione zoomata della mappa richiede una connessione ad Internet ed è possibile visualizzarla in un qualsiasi browser.

Risultati ottenuti:

Al termine del progetto, possiamo ritenerci estremamente soddisfatti dei risultati ottenuti dai nostri algoritmi sia rispetto ai benchmark di valutazione, sia rispetto alla risoluzione dello scenario iniziale che ci eravamo preposti di risolvere.

Le difficoltà più grandi hanno riguardato l'individuazione del miglior algoritmo da utilizzare sulla base del portfolio di algoritmi che abbiamo creato. Nello specifico, la fase che ci ha portato via più tempo è stata quella di valutazione degli algoritmi.

Paragonando i nostri risultati anche rispetto alle ultime ricerche pubblicate nell'ambito della risoluzione del problema del commesso viaggiatore applicato all'individuazione di un itinerario, il nostro algoritmo offre prestazioni molto elevate garantendo ottimi risultati in termini di accuracy e risultando scalabile rispetto ad eventuali estensioni, come, ad esempio, le nostre estensioni riguardanti i controlli sulle situazioni stradali, geografiche e meteorologiche.

Come potrebbe essere migliorato il nostro progetto:

I miglioramenti che ci sentiamo di poter proporre per un eventuale sviluppo futuro sono:

- Utilizzo di un dataset real-time per la situazione metereologica.
- Utilizzo di un dataset real-time per la situazione stradale.
- Permettere all'utente la selezione manuale delle città da visitare.

Riferimenti bibliografici.

Dal libro:

CAPITOLO 1.6.2: TASKS

CAPITOLO 1.6.3: DEFINING A SOLUTION

CAPITOLO 2.3.3: OFFLINE AND ONLINE COMPUTATION

CAPITOLO 3.1: PROBLEM SOLVING AS SEARCH

CAPITOLO 3.3: GRAPH SEARCHING

CAPITOLO 3.6: INFORMED SEARCH

CAPITOLO 3.6.2: BRANCH AND BOUND

CAPITOLO 3.8.1: DIRECTION OF SEARCH

CAPITOLO 3.8.2: DYNAMIC PROGRAMMING

CAPITOLO 4.6: LOCAL SEARCH

CAPITOLO 4.6.1: ITERATIVE BEST IMPROVEMENT

CAPITOLO 4.6.2: RANDOMIZED ALGORITHMS

CAPITOLO 4.6.3: LOCAL SEARCH VARIANTS

CAPITOLO 4.6.4: EVALUATING RANDOMIZED ALGORITHMS

CAPITOLO 4.6.5: RANDOM RESTART

CAPITOLO 4.7: POPULATION-BASED METHODS

CAPITOLO 4.8: OPTIMIZATION

CAPITOLO 7.2: SUPERVISED LEARNING FOUNDATIONS

CAPITOLO 7.2.1: EVALUATING PREDICTIONS

CAPITOLO 7.2.3: TYPES OF ERRORS

CAPITOLO 7.6: CASE-BASED REASONING

CAPITOLO 8.1: FEEDFORWARD NEURAL NETWORKS

CAPITOLO 15.4: DATALOG

Ricerche scentifiche:

Learning to Optimise General TSP Instances di Nasrin Sultana, Jeffrey Chan, A. K. Qin, Tabinda Sarwar [2020]

An Improvement to the 2-Opt Heuristic Algorithm for Approximation of Optimal TSP Tour di Fakhar Uddin, Naveed Riaz, Abdul Manan, Imran Mahmood, Oh-Young Song, Arif Jamal Malik, Aaqif Afzaal Abbaso [2023]

Genetic Algorithms: Comparing Evolution With and Without a Simulated Annealing-inspired Selection di Mats Andersson e David Mellin [2019]

Worst-case Analysis Of New Heuristic For Traveling Salesman Problem di Nicos Christofides