



LABORATORIO 2

ALGORITMI AVANZATI

Enrico Buratto (2028620)
Mariano Sciacco (2007692)
Federico Zanardo (2020284)

Maggio 2021

Indice

1	Introduzione	1
1.1	Descrizione del problema	1
1.2	NP-completezza del problema	1
1.2.1	Dimostrazione di NP-completezza	1
1.2.2	Inapprossimabilità per fattori ρ costanti di TSP	2
1.3	Definizione di disuguaglianza triangolare	3
2	Descrizione degli algoritmi	4
2.1	Struttura dati per il grafo	4
2.1.1	Introduzione	4
2.1.2	Implementazione	4
2.2	Held e Karp	5
2.2.1	Introduzione	5
2.2.2	Algoritmo	5
2.2.3	Implementazione	6
2.2.4	Analisi della qualità della soluzione	7
2.2.5	Analisi dei fattori di approssimazione	7
2.2.6	Ottimizzazioni implementate	8
2.3	Nearest Neighbor	8
2.3.1	Introduzione	8
2.3.2	Algoritmo	9
2.3.3	Analisi dei fattori di approssimazione	10
2.3.4	Implementazione	10
2.3.5	Ottimizzazioni implementate	10
2.4	2-approssimato	11
2.4.1	Introduzione	11
2.4.2	Definizione di MST	11
2.4.3	Algoritmo	12
2.4.4	Analisi della qualità della soluzione	12
2.4.5	Analisi del fattore di approssimazione	13
2.4.6	Implementazione	13
2.4.7	Ottimizzazioni implementate	13
3	Caratteristiche del programma e Misurazioni	14
3.1	Caratteristiche del programma	14
3.1.1	Introduzione	14
3.1.2	Installazione e requisiti	14
3.1.3	Avvio del programma	14
3.1.4	Caratteristiche tecniche del computer per le misurazioni	15
3.2	Introduzione alle misurazioni	15
3.3	Misurazione	15
3.3.1	Descrizione	15

3.3.2	File di output	16
4	Risposte alle domande	17
4.1	Domanda 1	17
4.1.1	Risultati e risultati temporali	17
4.2	Domanda 2	19
5	Conclusione	20

1 Introduzione

1.1 Descrizione del problema

In questa relazione illustreremo dei confronti tra tre algoritmi per risolvere un problema intrattabile, confrontando i tempi di calcolo e la *qualità* delle soluzioni che si possono ottenere con **algoritmi esatti** e con **algoritmi di approssimazione**. Il problema in questione è il **Travelling Salesman Problem** (**TSP** o *Problema del Commesso Viaggiatore*). Il nome di questo problema deriva dalla sua rappresentazione: data una rete di città, connesse tra loro tramite strade, si determini il percorso di minore distanza che un commesso viaggiatore deve fare per visitare tutte le città *una ed una sola volta*.

Il TSP si può rappresentare con un grafo non orientato, pesato e *completo* $G = (V, E)$, dove i vertici sono le città ed il peso del lato (u, v) è uguale alla distanza da u a v . Risolvere il TSP significa trovare un **circuito Hamiltoniano**, ovvero, un ciclo di costo minimo che visita tutti i vertici *esattamente una volta*.

1.2 NP-completezza del problema

1.2.1 Dimostrazione di NP-completezza

Prima di tutto dimostriamo che $\text{TSP} \in \mathcal{NP}$. Data un'istanza del problema, usiamo come *certificato* la sequenza degli n vertici del ciclo di peso minimo. L'algoritmo di verifica del certificato deve controllare che la sequenza contenga ogni vertice di G esattamente una volta, sommare il peso dei lati e controllare che il peso totale del ciclo sia inferiore a k . Questo controllo si può svolgere in tempo polinomiale ($O(n)$).

Per poter stabilire con precisione la complessità del problema, è necessario prima “trasformare” tale problema in un *problema di decisione*, aggiungendo un limite k per il peso del ciclo all'input del problema.

Definizione di TSP decisionale: Dato un grafo non orientato, completo e pesato $G = (V, E)$ e un valore $k > 0$, esiste un ciclo in G che attraversa tutti i vertici una sola volta di peso inferiore a k ?

Teorema: TSP è un problema \mathcal{NP} -completo.

Dimostrazione: è già stato dimostrato che $\text{TSP} \in \mathcal{NP}$, quindi si procede nel dimostrare che TSP è \mathcal{NP} -hard. Si mostri una riduzione del problema del circuito Hamiltoniano a TSP. Prendiamo un grafo non orientato $G = (V, E)$ e costruiamo un'istanza di TSP che ci permetta di risolvere il problema

del circuito Hamiltoniano su G . Costruiamo un grafo non orientato e completo $G' = (V, E')$ con gli stessi vertici di G e $E' = \{(u, v) | u, v \in V\}$. Il peso degli archi di G' viene assegnato come segue

$$w(u, v) = 0, \text{ se } \{u, v\} \in E$$

$$w(u, v) = 1, \text{ se } \{u, v\} \notin E$$

È possibile notare che si può costruire il grafo G' in tempo polinomiale rispetto al numero di vertici $|V|$ del grafo di partenza G . Il grafo G contiene un circuito Hamiltoniano se e solo se G' ha un ciclo di peso minore o uguale a 0. Supponiamo che G contenga un circuito Hamiltoniano $h = v_1, \dots, v_n$. Ogni lato che compone h è presente in E e quindi ha peso 0 in G' . Di conseguenza, h è un ciclo di G' di peso uguale a 0. Viceversa, supponiamo che G' contenga un ciclo semplice t che attraversa tutti i vertici e di peso minore o uguale a 0. Poiché i pesi dei lati in G' sono solo 0 oppure 1, tutti i lati che compongono t devono avere costo 0. Quindi tutti i lati del ciclo sono presenti anche in E e t è un circuito Hamiltoniano per G .

1.2.2 Inapprossimabilità per fattori ρ costanti di TSP

Il TSP è un problema simile al problema del MST: il MST è un cammino di peso minimo che collega tutti i vertici del grafo G , mentre il TSP è un ciclo di peso minimo che collega tutti i vertici del grafo G . Nonostante questa somiglianza tra questi due problemi, il TSP è un problema molto difficile anche solo da approssimare. Se esistesse un algoritmo di approssimazione con un ρ costante, allora saprei risolvere in tempo polinomiale un problema \mathcal{NP} -hard.

Teorema: se $\mathcal{P} \neq \mathcal{NP}$, non può esistere alcun algoritmo (polinomiale) di ρ -approssimazione per TSP con $\rho = \mathcal{O}(1)$.

Dimostrazione: per assurdo, si supponga che esista un algoritmo A_ρ polinomiale di ρ -approssimazione per TSP. Si dimostri come costruire $A_{Hamilton}$ che decide il problema del ciclo Hamiltoniano in tempo polinomiale. Sia $I = G = (V, E)$ e $O = "G \text{ contiene un ciclo Hamiltoniano?}"$. Si effettui una riduzione:

$$G \rightarrow G' = (V, E') \text{ è completo, dove}$$

$$c(e \in E') = 1 \text{ se } e \in E, \text{ altrimenti } c(e \in E') = \rho|V| + 1$$

Possiamo quindi creare delle rappresentazioni di G' e di c a partire dalle rappresentazioni di G in tempo polinomiale in $|V|$ ed $|E|$.

Eseguiamo $A_\rho(G') \rightarrow C$ (ciclo), $c(C)$ (funzione di costo di C , $c : V \times V \rightarrow \mathcal{N}$) e si determina:

1. $G \in HAMILTON \Rightarrow c(C^*) = |V| \Rightarrow A_\rho$, ritorna un ciclo C con $c(C) \leq \rho|V|$;

2. $G \notin \text{HAMILTON} \Rightarrow C$ contiene almeno un lato non in G (più precisamente in E) $\Rightarrow c(C) \geq \rho|V| + 1$. Poichè i lati che non appartengono a G sono costosi, c'è un *gap* di almeno $\rho|V|$ tra il costo di un cammino che è Hamiltoniano in G (di costo $|V|$) e il costo di qualsiasi altro cammino (di costo almeno $\rho|V| + |V|$). Pertanto, il costo di un cammino che non è un ciclo Hamiltoniano in G è almeno un fattore $\rho + 1$ maggiore del costo di un cammino che è un ciclo Hamiltoniano in G .

È possibile notare che si può costruire il grafo G' in tempo polinomiale rispetto al numero di vertici $|V|$ del grafo di partenza G . Il grafo G contiene un circuito Hamiltoniano se e solo se G' ha un ciclo di peso minore o uguale a 0. Supponiamo che G contenga un circuito Hamiltoniano $h = v_1, \dots, v_n$. Ogni lato che compone h è presente in E e quindi ha peso 0 in G' . Di conseguenza, h è un ciclo di G' di peso uguale a 0. Viceversa, supponiamo che G' contenga un ciclo semplice t che attraversa tutti i vertici e di peso minore o uguale a 0. Poiché i pesi dei lati in G' sono solo 0 oppure 1, tutti i lati che compongono t devono avere costo 0. Quindi tutti i lati del ciclo sono presenti anche in E e t è un circuito Hamiltoniano per G .

1.3 Definizione di disuguaglianza triangolare

La disuguaglianza triangolare è definita come

$$\forall u, v, w \in V, \text{ vale che } c(u, v) \leq c(u, w) + c(w, v) \quad (1)$$

dove c è la *funzione di costo*. Questo implica che il cammino con un lato $c(u, v)$ ha un costo minore o uguale al costo del cammino con due lati $c(u, w, v)$.

Questa definizione è necessaria alla definizione delle proprietà degli algoritmi *nearest neighbor* e di 2-approssimazione. Si applichi l'algoritmo di approssimazione A_ρ a TSP. Poichè A_ρ è garantito che restituisca un cammino di costo superiore a ρ volte il costo di un cammino ottimale, se G contiene un ciclo Hamiltoniano, A_ρ deve resituirlo. Se G non ha un ciclo Hamiltoniano, A_ρ resituisce un ciclo di costo maggiore di $\rho|V|$. Pertanto, possiamo usare A_ρ per risolvere il problema del ciclo Hamiltoniano in tempo polinomiale. Quindi sarei in grado di risolvere un problema \mathcal{NP} -completo con un algoritmo polinomiale. Assumendo che $\mathcal{P} \neq \mathcal{NP}$, questo non è possibile. Di conseguenza, non può esistere un algoritmo che ρ -approssima TSP.

2 Descrizione degli algoritmi

2.1 Struttura dati per il grafo

2.1.1 Introduzione

Il TSP prende in input un grafo *completo*, ovvero, un grafo in cui:

$$\forall u, v \in V \exists (u, v) \in E$$

Il grafo in questione viene definito *denso* e la struttura dati indicata per rappresentarlo è la *matrice di adiacenza*, di dimensione $|V| \times |V|$. Le celle della matrice contengono i pesi dei lati che congiungono i due vertici.

2.1.2 Implementazione

La struttura dati per il grafo è rappresentata dalla classe **TSP**; questa possiede i seguenti campi dati:

- **name**: il nome del dataset che un oggetto TSP rappresenta;
- **dimension**: la dimensione, ossia il numero di nodi, del dataset;
- **etype**: il tipo di rappresentazione delle coordinate dei vertici del dataset. Nel nostro programma questa assume uno dei due seguenti valori:
 - **EUC_2D** nel caso in cui le coordinate siano rappresentate in forma euclidea;
 - **GEO** nel caso in cui le coordinate siano rappresentate da latitudine e longitudine;
- **nodes**: struttura dati di tipo `defaultdict(list)`, ossia lista a dizionario. Questo campo dati contiene i vertici del grafo, indicizzati per indice riportato nel dataset;
- **adjMatrix**: matrice di adiacenza dei nodi. La matrice è di dimensione $|V| * |V|$, e conserva la seguente proprietà: *Detti i, j gli indici di due nodi, $adjMatrix[i][j] = adjMatrix[j][i] = w(i, j)$.*

La classe presenta inoltre i seguenti metodi:

- **add_node**: aggiunge un vertice al grafo;
- **get_weight**: restituisce il peso tra due vertici;
- **calculateAdjMatrix**: calcola la matrice di adiacenza del grafo. Questo metodo viene chiamato una sola volta, appena sono stati caricati tutti i metodi;

- `delete_node`: elimina un nodo dal grafo;
- `get_min_node`: restituisce il nodo più vicino al nodo che viene dato in *input*. Tale metodo è particolarmente utile per l'implementazione dell'algoritmo **nearest neighbor**, come specificato in seguito;
- `printAdjMatrix`: funzione di utilità per la stampa della matrice di adiacenza.

2.2 Held e Karp

2.2.1 Introduzione

L'algoritmo di Held and Karp è un algoritmo di programmazione dinamica che permette di determinare in modo esatto il risultato di TSP sfruttando la scomposizione del problema in sotto-problemi e in sotto-sotto-problemi che vengono ricorsivamente risolti. Per ciascun sotto-problema viene generato un risultato che viene salvato all'interno di una tabella contenente tutte le soluzioni che portano poi alla determinazione del cammino minimo di TSP.

L'algoritmo si basa sulla proprietà secondo la quale ogni sottocammino di un cammino minimo è a sua volta un cammino minimo, e ciò ci permette di calcolare i sottoproblemi ricorsivamente nel seguente modo:

- pongo $1 \dots n$ città (i.e. nodi) da visitare;
- pongo $S \subseteq V$ e $v \in S$, dove V sono le tutte le città in esame, S le città visitate fino a quel momento e v la città corrente;
- definisco $d[v, S]$ peso del cammino minimo nella città v a partire dalla città 1 percorrendo S città;
- definisco $\pi[v, S]$ predecessore di v nel cammino minimo percorrendo S città.

L'esecuzione dell'algoritmo a partire dalla città 1 visitando tutti i nodi S porterà al risultato del cammino minimo che sarà contenuto nel vettore $d[1, S]$, ripercorribile poi con il vettore π . Chiaramente questo tipo di algoritmo esegue in modalità *brute force* tutte le possibili soluzioni e restituisce il cammino minimo una volta raccolti tutti i possibili cammini.

2.2.2 Algoritmo

Il seguente pseudocodice mostra la funzione HK-INIT utilizzata per l'inizializzazione dei vettori e del tempo iniziale.


```
HK-INIT(G)
  (V,E) = G
  d = NULL
  pi = NULL
  global_time_start = time.now
  return HK-VISIT(0,V)
```

Lo pseudocodice della funzione HK-VISIT mostra esattamente i casi base e i casi ricorsivi per la risoluzione del problema in modo ricorsivo sfruttando l'algoritmo di Held and Karp.

```
HK-VISIT(v,S)
  // Caso base 1: ritorno il peso dell'arco {v,0}
  if |S| = 1 and S = {v} then
    return w[v,0]
  // Caso base 2: se la distanza è già stata calcolata, ritorno peso dell'arco
  else if d[v,S] is not empty then
    return d[v,S]
  // Caso ricorsivo: cerco il cammino minimo tra tutti i sottocammini
  else
    mindist = inf
    minprec = NULL
    for all u in S \ {v} do
      if time.now - global_time_start < 180 then
        dist = HK-VISIT(u,S \ {v})
        if dist + w[u, v] < mindist then
          mindist = dist + w[u, v]
          minprec = u
        end if
      else
        return mindist
      end if
    end for
    d[v,S] = mindist
    pi[v,S] = minprec
    return mindist
  end if
```

2.2.3 Implementazione

L'implementazione dell'algoritmo si è basata a partire da una visita in profondità (dall'alto) dei nodi mediante l'uso di due funzioni principali `hk_init` e `hk_visit`. Queste due funzioni sono contenute

dentro una classe Python chiamata `HeldKarp` che contiene delle variabili per i vettori d e π , nonché per il tempo di esecuzione di inizio usato per bloccare l'algoritmo se il tempo totale supera i tre minuti, come richiesto dalla consegna del progetto.

Al fine di realizzare i vettori d e π si è deciso di utilizzare la struttura dati `defaultdict` in modo analogo a una mappa una chiave con un suo valore relativo. Questo è stato particolarmente utile per inserire delle chiavi complesse formate da una stringa generata la seguente struttura:

$$v[p \dots q]$$

dove v rappresenta il nodo corrente, mentre p e q indicano rispettivamente il nodo iniziale e finale visitato in S , tale per cui $p \in S$ e $q \in S$.

Per quanto concerne la complessità, si può affermare che, sebbene l'approccio originale dell'algoritmo porta a concludere la risoluzione dell'intero problema seguendo un approccio "*brute force*", il caso peggiore è certamente inferiore rispetto $O(n!)$. Nello specifico, analizzando i singoli valori:

- $O(n)$ relativo alle $n - 1$ iterazioni richieste per il ciclo che itera tutti i vertici in S v ;
- $O(n \cdot 2^n)$ relativo alle coppie usate per l'indicizzazione con v e S che sono rispettivamente al più n e 2^n nel vettore d .

Pertanto in totale la complessità può essere riassunta come $O(n) \cdot O(n \cdot 2^n)$, ossia $O(n^2 \cdot 2^n)$.

2.2.4 Analisi della qualità della soluzione

La soluzione riportata risulta essere esatta dal momento che questo tipo di algoritmo utilizza la programmazione dinamica con un approccio *brute force* per calcolarsi tutte le possibili soluzioni. Per evitare un grande utilizzo di memoria dovuto alle ricorsioni, l'algoritmo è stato bloccato allo scadere di 3 minuti di computazione così da evitare una ricerca troppo lunga. Inoltre, si è dovuto estendere il limite di ricorsione fino a 5000 chiamate ricorsive di profondità per limitazioni dovute al linguaggio di programmazione Python.

2.2.5 Analisi dei fattori di approssimazione

Nell'algoritmo non ci sono state particolari approssimazioni se non nei risultati dei dataset che non hanno completato per intero la risoluzione dei sottoproblemi. Infatti, allo scadere dei 3 minuti i risultati ottenuti in output sono pari alle soluzioni dei cammini minimi fino ad allora trovate. Una

giusta approssimazione pertanto non può essere determinata, se non mettendo in relazione l'effettiva soluzione ottima con la soluzione attualmente presa in carico, determinando manualmente e in modo dinamico il tempo richiesto per il raggiungimento dell'effettiva soluzione.

2.2.6 Ottimizzazioni implementate

Nell'algoritmo ci sono state piccole ottimizzazioni a livello di codice di programmazione che hanno permesso di evitare delle operazioni computazionalmente semplici ma che con la complessità analizzata aumentavano di molto il tempo necessario alla computazione. Per prima cosa, l'uso di variabili temporanee ha permesso il riutilizzo di valori precedentemente trovati evitando di doverne ri-eseguire il ricalcolo, come nel caso della generazione della chiave. Nello specifico, la chiave viene generata all'inizio dell'algoritmo a partire dalla conversione in stringa del nodo v corrente concatenato alla lista S , a sua volta convertita in stringa. Ci si è accorti che l'uso di una funzione che potesse generare questo tipo di stringa avrebbe aumentato di molto il tempo richiesto per la risoluzione del problema, mentre l'uso di una variabile iniziale ha ridotto di molto il tempo per ricavare il valore della chiave. Altre soluzioni al posto della chiave interpretata come stringa potevano richiedere l'uso di array per S che fosse immutabile se usato come chiave, ma si è preferito non indagare ulteriormente visto e considerato che ai fini dell'esercizio una stringa ci è risultata più semplice da indicizzare rispetto a un'intera struttura dati.

Secondariamente, una seconda ottimizzazione è stata fatta prima di arrivare alla guardia del primo ciclo dove è necessario ricavare un vettore $S - v$ che richiede inevitabilmente la copia profonda e la rimozione di v con complessità $O(n) + O(n - 1) = O(n)$. Al fine di evitare l'uso della libreria di copia profonda, si è pensato trivialmente di eseguire un ciclo di cui copiare i valori di S evitando v e ottenendo pertanto una complessità di $O(n - 1) = O(n)$. Contrariamente a quanto si possa pensare, sebbene sulla teoria la complessità è la medesima, il valore ricavato ottimizza di molto le istruzioni effettive necessarie per attuare una copia di questo tipo. A titolo esemplificativo e puramente informale si è rilevato nel caso del dataset *burma14* un decremento del 35% del tempo impiegato per l'esecuzione sfruttando un ciclo di copia con un *if statement* al posto della copia profonda e della successiva rimozione.

2.3 Nearest Neighbor

2.3.1 Introduzione

L'algoritmo *Nearest Neighbor* per il problema del Traveling Salesman Problem rientra nelle cosiddette *euristiche costruttive*; queste sono una famiglia di algoritmi che, a partire da un vertice iniziale, procedono aggiungendo un vertice alla volta al sottoinsieme corrispondente alla soluzione parziale.

Questa aggiunta viene eseguita secondo regole prefissate, fino all'individuazione di una soluzione approssimata.

2.3.2 Algoritmo

Come tutte le euristiche costruttive per il problema TSP, l'algoritmo *nearest neighbor* può essere scomposto in tre passi: inizializzazione, selezione e inserimento. Viene anzitutto illustrato l'algoritmo in pseudocodice, e a seguire vengono analizzati nello specifico i tre passi.

```
NearestNeighbor(G=(V,E))
# il cammino iniziale è composto unicamente dal primo nodo del grafo
path <- v1 in V

# si itera finché tutti i nodi del grafo non sono stati inseriti nel path
while path != V
    # si seleziona il nodo NON presente in path di distanza minima dall'ultimo
    # nodo inserito nel path
    nextNode <- MinAdj(path, path.lastV)

    # si inserisce il nodo successivo nel path
    path <- path + nextNode

# si aggiunge infine il nodo di partenza per chiudere il ciclo
path <- path + v1

return path
```

Le tre fasi, riconoscibili dalla struttura dell'algoritmo, sono:

1. **Inizializzazione:** si parte con il cammino composto unicamente dal primo vertice v_0 ;
2. **Selezione:** sia (v_0, \dots, v_k) il cammino corrente; il vertice successivo è il vertice v_{k+1} , non presente nel cammino, a distanza minima da v_k ;
3. **Inserimento:** viene inserito v_{k+1} subito dopo v_k nel cammino.

I passi 2 e 3 vengono ripetuti finché tutti i vertici non sono nel cammino finale. Una volta verificata questa condizione è necessario eseguire un'ulteriore singola istruzione: aggiungere in coda al percorso finale il nodo iniziale. Questa operazione è necessaria ai fini del mantenimento della

proprietà di percorso minimo del problema TSP.

Questo algoritmo, come le altre euristiche costruttive, ha complessità computazionale pari a $O(|V|^2)$.

2.3.3 Analisi dei fattori di approssimazione

Definita la disuguaglianza triangolare come

$$\forall u, v, w \in V, \text{ vale che } c(u, v) \leq c(u, w) + c(w, v) \quad (2)$$

l'algoritmo *nearest neighbor* permette di trovare una soluzione $\log(n)$ -approssimata a TSP quando la disuguaglianza triangolare è rispettata.

2.3.4 Implementazione

La nostra implementazione ricalca l'implementazione standard dell'algoritmo Nearest Neighbor per il problema del TSP; essa segue infatti i tre passi sopracitati mantenendo intatta la logica dell'algoritmo di riferimento.

Operativamente, il codice dell'algoritmo si divide in una classe, `NearestNeighbor`, e in alcuni metodi di utilità implementate per la classe `TSP` citata precedentemente.

La classe `NearestNeighbor` presenta un solo metodo, chiamato `algorithm`, che si occupa di effettuare l'algoritmo in questione su un singolo grafo di classe `TSP`; esso riceve in *input* il grafo e restituisce il peso totale del percorso hamiltoniano trovato.

Il metodo `algorithm` chiama, al suo interno, i metodi `delete_node` e `get_min_node` della classe `TSP`; questi sono di particolare importanza nella nostra implementazione, poiché permettono di mantenere la complessità al suo valore teorico, come viene illustrato nella sezione seguente.

2.3.5 Ottimizzazioni implementate

La principale ottimizzazione implementata risiede nel metodo con il quale viene selezionato il nodo successivo da inserire nel cammino finale. Un'implementazione *naive* dell'algoritmo, infatti, potrebbe ricercare il nodo da inserire scorrendo linearmente l'intero grafo e controllando, per ogni nodo analizzato, che questi non sia già presente nel cammino finale e che sia a distanza minima dall'ultimo nodo inserito nel percorso finale. Questa implementazione, però, è stata giudicata inefficiente, poiché richiederebbe di analizzare ogni nodo anche se è già stato visitato in precedenza.

Abbiamo quindi adottato un approccio "inverso": nella fase di inizializzazione dell'algoritmo viene infatti creata una copia (profonda) del grafo iniziale; da questa copia vengono poi, nella fase di selezione, eliminati i nodi già considerati. Questa ottimizzazione permette di effettuare la ricerca esclusivamente sui nodi non ancora considerati, il cui numero diminuisce costantemente durante l'iterazione della fase di selezione.

A questa ottimizzazione si aggiunge l'utilizzo di una semplice lista, chiamata *visited*, che permette di mantenere l'integrità dell'algoritmo nel momento in cui viene chiamata la funzione `get_min_node`: se un nodo è già stato visitato, il controllo sul peso dell'arco che lo unisce al nodo in analisi non viene neanche effettuato.

Un'ultima ottimizzazione risiede nella funzione `get_min_node` di TSP, che fa uso delle liste di adiacenza per la ricerca del nodo di peso minimo; in questo modo, invece di calcolare a *runtime* il peso tra un nodo e l'altro, è sufficiente uno scorrimento lineare della matrice di adiacenza sulla riga (o sulla colonna) relativa al nodo in esame.

2.4 2-approssimato

2.4.1 Introduzione

L'algoritmo 2-approssimato è in grado di determinare un'approssimazione del TSP sotto la condizione che le distanze rispettino la *disuguaglianza triangolare*, ovvero:

$$\forall u, v, w \in V, \text{ vale che } c(u, v) \leq c(u, w) + c(w, v) \quad (3)$$

dove c è la *funzione di costo*. Questo implica che il cammino con un lato $c(u, v)$ ha un costo minore o uguale al costo del cammino con due lati $c(u, w, v)$. Il nome di questo problema è **TRIANGLE_TSP**.

2.4.2 Definizione di MST

Sia V l'insieme dei nodi che costituiscono il grafo pesato G e sia E la collezione dei lati di tale grafo. Ai fini delle analisi della complessità degli algoritmi, sia $|V| = n$ e $|E| = m$.

Un *minimum spanning tree* è un sottoinsieme dei lati E di un grafo G non orientato connesso e pesato sui lati che collega tutti i vertici insieme, senza alcun ciclo e con il minimo peso totale del lato possibile. Cioè, è uno spanning tree la cui somma dei pesi dei bordi è la più piccola possibile.

Un *minimum spanning tree* $T = (V, E')$ è un albero, il cui insieme dei lati E' è un sottoinsieme dei lati E di un grafo $G = (V, E)$ non orientato, connesso e pesato, che collega tutti i vertici V , la cui somma dei pesi dei lati è la minima.

L'algoritmo generico per determinare un MST è:

```
A = empty_set
while A doesn't form a spanning tree
    find an edge (u,v) that is safe for A
```

```
    A = A U {(u,v)}  
return A
```

Si forniscono alcune definizioni per gli MST:

1. un **taglio** $(S, V \setminus S)$ di un grafo $G = (V, E)$ è una partizione di V ;
2. un lato $(u, v) \in E$ **attraversa il taglio** $(S, V \setminus S)$ se $u \in S$ e $v \in V \setminus S$ o viceversa;
3. un taglio **rispetta** un insieme A di lati se nessun lato di A attraversa il taglio;
4. dato un taglio, il lato che lo attraversa di peso minimo si chiama **light edge**.

Per determinare se un lato è **safe**, si sfrutta il seguente teorema:

Teorema: Sia $G = (V, E)$ un grafo non diretto, connesso e pesato. Sia A un sottoinsieme di E incluso in una qualche MST di G , sia $(S, V \setminus S)$ un taglio che rispetta A , e sia (u, v) un *light edge* per $(S, V \setminus S)$. Allora (u, v) è *safe* per A .

2.4.3 Algoritmo

L'idea che sta dietro a questo algoritmo consiste nell'utilizzare un algoritmo per il calcolo del MST. Tuttavia, il MST è un albero e quello che vogliamo ottenere invece è un ciclo hamiltoniano. Per fare ciò, eseguiamo una visita in *preorder* del MST e aggiungiamo la radice di tale MST alla lista della determinata dalla preorder. Questo è un ciclo hamiltoniano del grafo originale, in quanto quest'ultimo è completo. Pertanto, esiste sempre un lato tra ogni coppia di vertici.

```
2-APPROXIMATION(G=(V,E), c)  
    root <- v1 in V          // scelgo un nodo di V come radice per Prim  
    T <- Prim(G, c, root)  
    H' <- preorder(root)     // visita in preorder  
    H <- H U {root}          // aggiungo il percorso che va dall'ultimo nodo  
                             // della visita in preorder alla radice  
  
    return H
```

2.4.4 Analisi della qualità della soluzione

Si illustri l'analisi della qualità della soluzione ritornata dall'algoritmo:

1. Il costo di H' è basso per la definizione di MST;
2. supponiamo che presi due vertici a e b non siano collegati da un lato (anche se sappiamo che il grafo è completo), L'idea è che il costo di (a, b) è minore rispetto al costo di fare un giro più largo, in quanto vale la *disuguaglianza triangolare*. Quindi in questo caso (a, b) è un *shortcut*.

2.4.5 Analisi del fattore di approssimazione

Si illustri l'analisi del fattore di approssimazione dell'algoritmo:

1. *Limite inferiore al costo della soluzione ottima H* : sia H il ciclo ottimo $H^{ottimo} = \langle v_{j1}, \dots, v_{jn}, v_{j1} \rangle$. Sia $H'^{ottimo} = \langle v_{j1}, \dots, v_{jn} \rangle$ un cammino (è uno spanning tree, non un ciclo) hamiltoniano. Quindi, $c(H') \geq c(T)$, dove T è il MST, e $c(H^{ottimo}) \geq c(H'^{ottimo})$ perchè i costi sono maggiori o uguali a zero.
2. *Limite superiore al costo della soluzione restituita H* : dato un albero, una *full preorder chain* è una lista con ripetizioni dei nodi dell'albero che indica i nodi raggiunti dalle chiamate ricorsive dell'algoritmo **Preorder**.

Proprietà: in una full preorder chain ogni arco di T appare esattamente due volte. Quindi $c(\text{full preorder chain}) = 2 \cdot c(T)$. Se si eliminano dalla full preorder chain tutte le occorrenze successive alla prima dei nodi interni (tranne l'ultima occorrenza della radice) otteniamo, grazie alla *disuguaglianza triangolare*:

$$2 \cdot c(T) \geq c(H) \Rightarrow 2 \cdot c(H^{ottimo}) \geq 2 \cdot c(T) \geq c(H) \Rightarrow \frac{c(H)}{c(H^{ottimo})} \leq 2$$

2.4.6 Implementazione

2.4.7 Ottimizzazioni implementate

3 Caratteristiche del programma e Misurazioni

3.1 Caratteristiche del programma

3.1.1 Introduzione

Il programma è stato realizzato interamente in Python e si struttura in 4 cartelle principali che rappresentano: il modulo degli algoritmi (*algorithms*), il modulo delle strutture dati (*data_structures*), il modulo delle misurazioni (*measurements*) e il dataset contenente i grafi (*dataset*).

3.1.2 Installazione e requisiti

Prima di eseguire il programma è necessario installare le dipendenze presenti nel file *requirements.txt* eseguendo il seguente comando: `pip install -r requirements.txt`

Per l'esecuzione è richiesto l'uso di un terminale Windows, MacOS, Linux o BSD-like con Python 3.x+ e PIP installato.

3.1.3 Avvio del programma

Per l'esecuzione del programma è necessario spostarsi nella cartella di progetto ed eseguire il seguente comando: `python3 main.py [--version] [-h] <type> <dataset>` dove:

- **type:** rappresenta il tipo di algoritmo o di serie di esecuzioni che si vuole avviare, nello specifico uno tra i seguenti:
 - *all* (esegue e salva su file le misurazioni singole);
 - *hk* (esegue le misurazioni con Held-Karp);
 - *nn* (esegue le misurazioni con Nearest Neighbor);
 - *2ap* (esegue le misurazioni con 2-approximation).
- **dataset:** rappresenta una cartella contenente i file dei dataset formattati come da consegna o singolo file di dataset in input.
- **--version:** è opzionale e mostra la versione del programma in esecuzione.
- **-h:** è opzionale e mostra un aiuto per i comandi da utilizzare.

3.1.4 Caratteristiche tecniche del computer per le misurazioni

L'esecuzione del programma e le relative misurazioni sono state effettuate sulla seguente macchina:

- **CPU:** Ryzen 9 3900x (12 core / 24 thread), 4.6ghz
- **RAM:** 32 GB DDR4
- **SSD:** NVME SSD 512 GB

Tutte le misurazioni sono state eseguite **sequenzialmente** monitorando l'uso di un core dedicato della CPU al 100% per tutta la durata dell'esecuzione.

3.2 Introduzione alle misurazioni

Le misurazioni sono state effettuate eseguendo i tre algoritmi per ogni grafo del dataset, misurandone il tempo di esecuzione medio; dalla misura dei tempi è stato ovviamente escluso il tempo di caricamento dei dataset negli oggetti TSPTS. Prima di ogni esecuzione dei vari algoritmi è stato inoltre disabilitato temporaneamente il *garbage collector* di Python, così da evitare rallentamenti anche minimi nel corso dell'esecuzione. Infine, ciascuno dei due moduli salva in tre file differenti - uno per ogni algoritmo - i risultati del programma.

3.3 Misurazione

3.3.1 Descrizione

La misurazione singola è organizzata sequenzialmente eseguendo in base al tipo di algoritmo l'intero dataset. Il metodo *executeSingleGraphCalculus* applica l'algoritmo richiesto al graph in input e procede nella seguente maniera:

1. Esegue una volta l'algoritmo applicato al grafo, disabilitando il *garbage collector* e salvando il tempo di esecuzione.
2. Verifica se il tempo di esecuzione è maggiore o meno a 1s.
 - Se il tempo è minore a 1s, esegue nuovamente k volte l'algoritmo, disabilitando il *garbage collector* e salvando il tempo di esecuzione medio sulle k esecuzioni, con k definito come segue:

$$k = \left\lceil \frac{10^9 \text{ ns}}{\text{tempo medio di esecuzione in ns}} \right\rceil$$

ossia il numero di ripetizioni applicate all'algoritmo la cui somma arriva a 1s.

- Altrimenti, viene mantenuto il tempo rilevato precedentemente con una singola esecuzione.

3. Esegue il salvataggio in *append* nel file di output in formato CSV.

3.3.2 File di output

Il file di output viene salvato direttamente alla fine di ogni esecuzione dell'algoritmo mantenendo la seguente formattazione:

- Nome del dataset;
- Peso finale calcolato;
- Tempo di esecuzione in nanosecondi;
- Tempo di esecuzione in secondi;
- Esecuzioni totali effettuate dell'algoritmo.

4 Risposte alle domande

4.1 Domanda 1

Eseguite i tre algoritmi che avete implementato (Held-Karp, euristica costruttiva e 2-approssimato) sui 13 grafi del dataset. Mostrate i risultati che avete ottenuto in una tabella come quella sottostante. Le righe della tabella corrispondono alle istanze del problema. Le colonne mostrano, per ogni algoritmo, il peso della soluzione trovata, il tempo di esecuzione e l'errore relativo calcolato come $(\text{SoluzioneTrovata} - \text{SoluzioneOttima})/\text{SoluzioneOttima}$. Potete aggiungere altra informazione alla tabella che ritenete interessanti.

Abbiamo implementato il codice in Python per l'esecuzione dei tre algoritmi su tutto il dataset fornito. Nella tabella che segue sono riportate le soluzioni ottime dei diversi dataset come riferimento per i risultati che abbiamo ottenuto.

File	Descrizione	N	Soluzione Ottima
<i>berlin52.tsp</i>	Berlino	52	7542
<i>burma14.tsp</i>	Birmania (Myanmar)	14	3323
<i>ch150.tsp</i>	Random	150	6528
<i>d493.tsp</i>	Foratura di circuiti stampati	493	35002
<i>dsj1000.tsp</i>	Random	1000	18659688
<i>eil51.tsp</i>	Sintetico	51	426
<i>gr202.tsp</i>	Europa	202	40160
<i>gr229.tsp</i>	Asia/Australia	229	134602
<i>kroA100.tsp</i>	Random	100	21282
<i>kroD100.tsp</i>	Random	100	21294
<i>pcb442.tsp</i>	Foratura di circuiti stampati	442	50778
<i>ulysses16.tsp</i>	Mediterraneo	16	6859
<i>ulysses22.tsp</i>	Mediterraneo	22	7013

Tabella 1: Soluzioni ottime per ogni algoritmo su ogni dataset.

4.1.1 Risultati e risultati temporali

I risultati sperimentali degli algoritmi da noi implementati sono riportati nella tabella alla pagina che segue.

Istanza	Held-Karp			Nearest Neighbor			2-approximation		
	Soluzione	Tempo [s]	Errore [%]	Soluzione	Tempo [s]	Errore [%]	Soluzione	Tempo [s]	Errore [%]
<i>berlin52.tsp</i>	17739	180.0000379	135.20%	8980	0.0014364	19.07%	10114	0.0025047	34.10%
<i>burma14.tsp</i>	3323	0.3493254	0%	4048	0.0001614	21.82%	3814	0.0002581	14.78%
<i>ch150.tsp</i>	48029	180.0001351	635.74%	8191	0.0109101	25.47%	8347	0.0214446	27.86%
<i>d493.tsp</i>	111941	180.0005959	219.81%	41660	0.1401718	19.02%	44892	0.2299229	28.26%
<i>dsj1000.tsp</i>	551275688	180.0014467	2854.37%	24630960	0.5707007	32.00%	25086767	0.7079459	34.44%
<i>eil51.tsp</i>	1026	180.0000473	140.85%	511	0.0013854	19.95%	581	0.0020308	36.38%
<i>gr202.tsp</i>	55127	180.0001963	37.27%	49336	0.0205211	22.85%	51990	0.0374635	29.46%
<i>gr229.tsp</i>	176680	180.0002368	31.26%	162430	0.0284802	20.67%	180152	0.0480593	33.84%
<i>kroA100.tsp</i>	166257	180.0000981	681.21%	27807	0.004863	30.66%	27210	0.0076621	27.85%
<i>kroD100.tsp</i>	146862	180.0000945	589.69%	26947	0.0049283	26.55%	27112	0.009628	27.32%
<i>pcb442.tsp</i>	204852	180.0005189	303.43%	61979	0.1128042	22.06%	73030	0.1584171	43.82%
<i>ulysses16.tsp</i>	6859	1.9638338	0%	9988	0.0002	45.62%	7903	0.0003195	15.22%
<i>ulysses22.tsp</i>	7105	180.0000249	1.31%	10586	0.0003321	50.95%	8401	0.0005173	19.79%

Tabella 2: Risultati dell'esecuzione dei tre algoritmi sui dataset.

Riteniamo utile, inoltre, riportare nel dettaglio i tempi di esecuzione degli algoritmi e il numero di ripetizioni effettuate da ogni algoritmo su ogni dataset.

File	N	Tempo H-K [s]	Rep HK	Tempo NN [s]	Rep NN	Tempo 2-ap [s]	Rep 2-ap
<i>berlin52</i>	52	180.0000379	1	0.0014364	663	0.0025047	385
<i>burma14</i>	14	0.3493254	2	0.0001614	5913	0.0002581	3705
<i>ch150</i>	150	180.0001351	1	0.0109101	91	0.0214446	46
<i>d493</i>	493	180.0005959	1	0.1401718	7	0.2299229	4
<i>dsj1000</i>	1000	180.0014467	1	0.5707007	1	0.7079459	1
<i>eil51</i>	51	180.0000473	1	0.0013854	701	0.0020308	482
<i>gr202</i>	202	180.0001963	1	0.0205211	48	0.0374635	26
<i>gr229</i>	229	180.0002368	1	0.0284802	35	0.0480593	20
<i>kroA100</i>	100	180.0000981	1	0.004863	203	0.0076621	129
<i>kroD100</i>	100	180.0000945	1	0.0049283	201	0.009628	103
<i>pcb442</i>	442	180.0005189	1	0.1128042	8	0.1584171	6
<i>ulysses16.tsp</i>	16	1.9638338	1	0.0002	4706	0.0003195	3046
<i>ulysses22.tsp</i>	22	180.0000249	1	0.0003321	2832	0.0005173	1911

Tabella 3: Dettaglio dei tempi di esecuzione.

4.2 Domanda 2

Commentate i risultati che avete ottenuto: come si comportano gli algoritmi rispetto alle varie istanze? C'è un algoritmo che riesce sempre a fare meglio degli altri rispetto all'errore di approssimazione? Quale dei tre algoritmi che avete implementato è più efficiente?

Analizzando i tre algoritmi abbiamo appurato che non esiste una regola generale per quanto riguarda l'accuratezza del risultato. Il più preciso dei tre è, naturalmente, l'algoritmo Held-Karp, in quanto per definizione calcola la soluzione ottima; nonostante questo, avendo una complessità di gran lunga maggiore rispetto agli altri, nei tre minuti che abbiamo imposto come limite alla computazione su un singolo dataset molto spesso si trova molto più lontano dalla soluzione rispetto agli algoritmi di approssimazione. Questo è particolarmente evidente nei grafi con un elevato numero di vertici; in *dsj1000*, infatti, la soluzione trovata ha un errore di addirittura due ordini di grandezza in più rispetto alla soluzione ottima.

Anche tra gli algoritmi di approssimazione non è possibile tracciare una linea precisa su quale sia più preciso o meno: dai risultati ottenuti possiamo evincere solamente che l'algoritmo di 2-approssimazione tende a essere più preciso su grafi di piccole dimensioni, mentre l'euristica nearest neighbor tende a trovare una soluzione più vicina all'ottimo per grafi di grandi dimensioni.

5 Conclusione