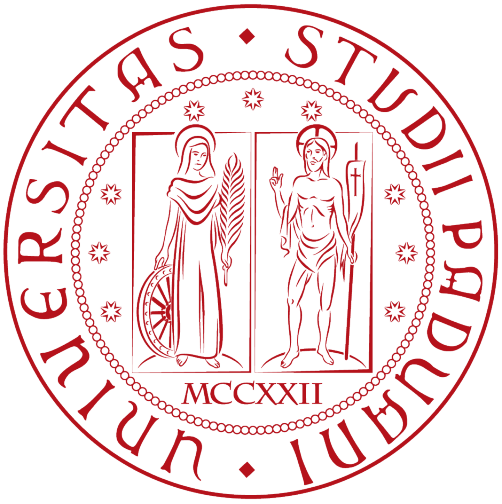

Algoritmi avanzati

Laboratorio 2

Traveling Salesman Problem

Studenti:
Marco Pozza

Matricole:
1219870

Introduzione

Questo documento è una relazione riguardante il secondo esperimento di laboratorio del corso di Algoritmi Avanzati della laurea Magistrale in Informatica dell'università degli studi di Padova. Lo scopo dell'esercitazione è quello di andare a risolvere il problema intrattabile del **Traveling Salesman Problem** mettendo a confronto tre algoritmi, di cui uno esatto e due di approssimazione, su tempi e qualità delle soluzioni trovate.

Descrizione del dataset

Il dataset utilizzato contiene un totale di 13 grafi sia reali che generati in modo randomico i quali contengono un numero di nodi che varia tra 14 e 1000. I grafi sono descritti nella seguente modalità:

```
NOME
COMMENT
TYPE
DIMENSION
EDGE_WEIGHT_TYPE
NODE_COORD_SECTION
1  37  52
2  49  49
3  54  64
...
```

In ordine abbiamo:

- **nome** del grafo;
- breve **descrizione** del grafo;
- **tipologia** di problema che si vuole risolvere;
- **numero di nodi** che formano il grafo;
- **tipologia di coordinate**. In questo esperimento i grafi possono avere due differenti tipologie di coordinate:
  - **EUC_2D** che sono coordinate di tipo Euclideo in cui la distanza tra due punti viene calcolata dalla seguente formula:

    Dati due punti descritti come: $P = (X_p, Y_p)$ e $Q = (X_q, Y_q)$

    $$dist(P, Q) = \sqrt{(X_p - X_q)^2 + (Y_p - Y_q)^2}$$

  - **GEO** che sono coordinate Geografiche in cui la distanza tra due punti viene calcolata tramite i passaggi descritti al seguente link.

- sezione in cui si descrivono i nodi che formano un grafo. Qui abbiamo:
  - **id** univoco del nodo;
  - coordinata **X** che, nel caso della tipologia GEO rappresenta la latitudine;
  - coordinata **Y** che, nel caso della tipologia GEO rappresenta la longitudine.

Descrizione del problema

Quello che si vuole fare con questo esperimento è andare a mettere a confronto tre algoritmi che tentano di risolvere il Traveling Salesman Problem cosi definito:

Date le coordinate x e y dei nodi che formano il grafo e una funzione di peso $w(u, v)$ che descrive il peso del lato tra due nodi e che viene definita per ogni coppia di nodi (grafo completo), voglio il ciclo semplice di peso minimo che visita tutti i nodi del grafo. La funzione di peso $w(u, v)$ viene definita in base alle tipologia di coordinate (EUC_2D o GEO).

Gli algoritmi che vengono confrontati per tempi e qualità della soluzione sono:

- l'algoritmo esatto di **Held e Karp**;
- l'algoritmo di 2-approssimazione implementato con **Prim**;
- l'euristica costruttiva **Nearest Neighbor**.

Algoritmo di Held e Karp

L'algoritmo di Held e Karp è un algoritmo di programmazione dinamica per risolvere, in modo esatto, il Traveling Salesman Problem (TSP).
L'approccio che viene utilizzato è quello a programmazione dinamica perché per risolvere il TSP bisogna risolvere dei sotto-problemi che, a loro volta, hanno dei sotto-problemi condivisi. In questo modo si può risolvere un unica volta questi sotto-problemi, salvarli in una tabella e recuperare i risultati quando necessario senza doverli calcolare nuovamente.

L'algoritmo di Held e Karp basa la sua logica sulla seguente proprietà dei cammini minimi:

Ogni sottocammino di un cammino minimo è un cammino minimo

Questa permette di formulare il problema del TSP in modo ricorsivo.
L'algoritmo utilizza due strutture dati definite nel seguente modo:
Dato un insieme di vertici $S \subseteq V$, un vertice $v \in S$ e n nodi identificati univocamente da un numero che va da 1 a n:

- **d[v,S]** è il peso del cammino minimo che va dal nodo 1 al nodo v passando per tutti i nodi del sottinsieme S;
- **π [v,S]** è il predecessore di v nel cammino minimo che va da 1 a v passando per tutti i nodi del sottinsieme S.

Alla fine dell'esecuzione dell'algoritmo **d[1,V]** conterrà il peso del cammino minimo che parte da 1, termina in 1 e che passa per tutti i nodi del grafo. Questo equivale ad avere la soluzione per il TSP. L'informazione contenuta in $\boldsymbol{\pi}$ mi consentirà di ricostruire il ciclo risalendo i padri dei nodi.

Nonostante questo algoritmo permetta di risolvere in modo esatto il problema del TSP la sua complessità è $O(n^2 \cdot 2^n)$ il che lo rende inutilizzabile. Per questa ragione il tempo di calcolo dell'algoritmo è stato limitato ad un massimo di circa 3 minuti dopo il quale l'algoritmo si ferma e ritorna il miglior risultato trovato fino a quel momento. La scelta del tempo limite è stata fatta in base al hardware in cui è stato fatto girare il codice che, dopo circa 3 minuti, va in errore per saturazione dello heap.

2-approssimazione con Prim

L'algoritmo di 2-approssimazione basa la sua logica sul fatto che è possibile approssimare la soluzione del TSP andando a calcolare il Minimum Spanning Tree (MST) del grafo considerato e, tramite un apposita visita, andare a generare un ciclo che passa per tutti i suoi nodi. Questo è vero solamente se la disuguaglianza triangolare è rispettata ovvero quando il cammino diretto verso un nodo è più conveniente che passare per nodi intermedi.
L'algoritmo di 2-approssimazione esegue i seguenti passaggi:

- scelgo il nodo 1 come nodo iniziale e finale per chiudere il ciclo;
- tramite un apposito algoritmo calcolo l'MST del grafo partendo dal nodo 1;
- visito l'MST tramite una visita *preorder* e creo la lista dei nodi visitati secondo quest'ordine;
- chiudo il ciclo andando ad aggiungere a questa lista il nodo 1.

La complessità finale dell'algoritmo dipende, quindi, fortemente dall'algoritmo di calcolo del MST scelto che, nel mio caso, è stato l'algoritmo di Prim che, tramite l'utilizzo di un minHeap, raggiunge una complessità di $O(mlogn)$.
Un altro fattore che influisce sulla complessità finale è il modo in cui viene implementata la visita preorder del MST.
Siccome l'algoritmo di Prim genera due strutture dati, una contenente i pesi e una contenente i padri dei nodi nel MST, è stata sfruttata quest'ultima per andare a creare un algoritmo di visita preorder che esegue i seguenti passi:

1. salva il nodo corrente nella lista;
2. recupera i figli del nodo considerato (tutti i nodi che lo hanno come padre) e li salva in una lista;
3. se la lista è vuota significa che è un nodo foglia e l'algoritmo non fa nulla;
4. se la lista non è vuota significa che è un nodo interno quindi, per ogni figlio, ripete la procedura dal punto 1.

In questo modo si può vedere come la creazione della lista contenente i figli dei nodi abbia una complessità $O(n)$ che viene eseguita tante volte quanti sono il numero di nodi interni (non foglia). La complessità finale può essere considerata $O(n)$.

Euristica costruttiva Nearest Neighbor

Le euristiche costruttive sono una famiglia di algoritmi che arrivano ad una soluzione procedendo un nodo alla volta seguendo delle regole ben precise.
Tutte le euristiche condividono la stessa struttura composta da tre passi distinti:

- **Inizializzazione**: scelta del punto di partenza o di un ciclo parziale;
- **Selezione**: scelta del nodo da inserire nella soluzione;
- **Inserimento**: scelta della posizione in cui inserire il nodo scelto.

L'euristica *Nearest Neighbor* è un algoritmo che visita il nodo più vicino non visitato a quello corrente. Questa euristica implementa i passi descritti nel seguente modo:

1. **Inizializzazione**: si parte da una soluzione parziale che contiene solamente il vertice 1;
2. **Selezione**: sia $V_0, ..., V_k$ il cammino calcolato fin'ora. Trovo il vertice V_{k+1} non ancora inserito con distanza minore dal vertice V_k;
3. **Inserimento**: inserisco V_{k+1} dopo V_k;
4. ripeto il punto 2. finché non ho inserito tutti i nodi nel cammino;
5. chiudo il circuito inserendo il vertice 1 alla fine del cammino.

Quando la disuguaglianza triangolare è rispettata allora si ottiene un soluzione di *log(n) - approssimazione* con una complessità finale di $O(n^2)$ con n taglia dell'input.

Rappresentazione dei grafi

In questo esperimento, dopo aver analizzato le caratteristiche delle varie strutture dati, si è scelto di rappresentare i grafi tramite **matrici di adiacenza**.
Le caratteristiche delle matrici che hanno spinto alla loro scelta sono le seguenti:

- complessità d'accesso ai dati molto bassa;
- è facile vedere se due nodi sono connessi;
- occupa molto spazio rispetto alle liste di adiacenza;
- l'aggiunta e rimozione di un lato tra due nodi è un operazione semplice e veloce;
- l'aggiunta e rimozione di un nodo è un operazione più lenta.

Date queste caratteristiche (analizzate più nel dettaglio nella relazione del primo laboratorio) i ragionamenti che ne sono seguiti sono stati:

1. siccome consideriamo grafi **completi** anche le liste di adiacenza avrebbero dovuto contenere tutti i lati da e verso ogni altro nodo. Lo spazio in memoria sarebbe quindi risultato lo stesso delle matrici di adiacenza;
2. siccome l'esperimento non prevede la manipolazione diretta del grafo non c'è alcun bisogno di aggiungere o rimuovere nodi e lati se non in fase di creazione;
3. c'è bisogno invece di recuperare le informazioni presenti all'interno delle strutture dati (pesi) tra i vari nodi per andare a risolvere il problema del TSP. Le matrici di adiacenza sono un ottima soluzione in quanto hanno un tempo di accesso costante $O(1)$.

Presentazione dei dati e analisi

Andiamo ora a presentare i dati raccolti dopo aver eseguito i tre algoritmi su tutti i grafi del dataset fornito.

Linguaggio e ambiente d'esecuzione

L'esperimento è stato condotto su una macchina con le seguenti specifiche:

- Processore Intel Core i5-7200U 3.1 Ghz;
- Memoria RAM 8GB.

Il linguaggio di programmazione scelto per la scrittura degli algoritmi è stato Typescript.

Risultati Held e Karp

			Soluzione esatta – Held-Karp		
Grafo	Numero nodi	Soluzione ottima	Soluzione	Tempo di esecuzione	Errore
burma14	14	3323	3323	0,66141 s	0,00%
ulysses16	16	6859	6859	3,62667 s	0,00%
ulysses22	22	7013	7188	200,00054 s	0,02%
eil51	51	426	1050	200,00008 s	1,46%
berlin52	52	7542	17898	200,00016 s	1,37%
kroA100	100	21282	167464	200,00011 s	6,87%
kroD100	100	21294	148066	200,00010 s	5,95%
ch150	150	6528	48362	200,91278 s	6,41%
gr202	202	40160	55127	200,00042 s	0,37%
gr229	229	134602	176922	200,97989 s	0,31%
pcb442	442	50778	205041	200,00026 s	3,04%
d493	493	35002	111947	200,00034 s	2,20%
dsj1000	1000	18659688	551746912	200,00087 s	28,57%

Risultati algoritmo di 2-approssimazione con Prim

			Soluzione 2-approssimata – Prim		
Grafo	Numero nodi	Soluzione ottima	Soluzione	Tempo di esecuzione	Errore
burma14	14	3323	4003	0,00064 s	0,20%
ulysses16	16	6859	7788	0,00042 s	0,14%
ulysses22	22	7013	8308	0,00058 s	0,18%
eil51	51	426	567	0,00140 s	0,33%
berlin52	52	7542	10402	0,01084 s	0,38%
kroA100	100	21282	30536	0,00426 s	0,43%
kroD100	100	21294	28599	0,01814 s	0,34%
ch150	150	6528	9053	0,02498 s	0,39%
gr202	202	40160	52615	0,00906 s	0,31%
gr229	229	134602	179335	0,01103 s	0,33%
pcb442	442	50778	74856	0,02905 s	0,47%
d493	493	35002	45114	0,04938 s	0,29%
dsj1000	1000	18659688	25526005	0,12396 s	0,37%

Risultati euristica costruttiva Nearest Neighbor

			Euristica costruttiva – Nearest Neighbor		
Grafo	Numero nodi	Soluzione ottima	Soluzione	Tempo di esecuzione	Errore
burma14	14	3323	4048	0,00003 s	0,22%
ulysses16	16	6859	9988	0,00001 s	0,46%
ulysses22	22	7013	10586	0,00002 s	0,51%
eil51	51	426	511	0,00006 s	0,20%
berlin52	52	7542	8980	0,00023 s	0,19%
kroA100	100	21282	27807	0,00015 s	0,31%
kroD100	100	21294	26947	0,00014 s	0,27%
ch150	150	6528	8191	0,00107 s	0,25%
gr202	202	40160	49336	0,00044 s	0,23%
gr229	229	134602	162430	0,01238 s	0,21%
pcb442	442	50778	61979	0,00175 s	0,22%
d493	493	35002	41665	0,00253 s	0,19%
dsj1000	1000	18659688	24630960	0,00954 s	0,32%

Risultati medi

Algoritmo	Tempo medio esec.	Errore medio
Nearest Neighbor	0,00218 s	0,27%
Prim	0,02183 s	0,32%
Held-Karp	169,70643 s	4,35%

Analisi

E' facile osservare, dai dati riportati nella tabella dei risultati medi, che l'algoritmo di *Held e Karp* ha un tempo di esecuzione medio molto più alto rispetto agli altri due algoritmi. Questo algoritmo ha, infatti, terminato l'esecuzione solamente per i grafi *burma14* e *ulysses16*, mentre negli altri casi l'algoritmo ha ecceduto il tempo limite, impostato a circa 3 minuti, ritornando il risultato migliore calcolato fino a quel momento. Proprio perché l'esecuzione è stata interrotta, la soluzione trovata in questi casi si discosta molto da quella ottima che, in media, risulta avere un errore del 4,35%. Per quanto riguarda gli altri due algoritmi gli errori medi sono molto simili tra loro infatti abbiamo:

- l'algoritmo *Nearest Neighbor* con un errore medio dello 0.27%;
- l'algoritmo di *2-approssimazione con Prim* con un errore medio dello 0.32%.

Da questo si può dire che, in entrambi i casi, le soluzioni trovate hanno un buon grado di approssimazione e, per quanto riguarda le performance, l'algoritmo *Nearest Neighbor* risulta essere leggermente migliore. Infatti il tempo medio di esecuzione risulta essere circa 0.002 secondi, contro i 0.02 secondi impiegati per trovare la soluzione con l'algoritmo di *2-approssimazione con Prim*.
Un altra cosa che può essere osservata è come nessuno dei due algoritmi abbia sempre risultati migliori rispetto all'altro, infatti l'algoritmo *Nearest Neighbor*, per i grafi con meno nodi (*burma14*, *ulysses16* e *ulysses22*), ottiene delle percentuali di errore maggiori rispetto all'algoritmo di *2-approssimazione con Prim* ma, una volta processati questi 3 grafi, l'algoritmo *Nearest Neighbor*, riesce ad ottenere percentuali di errori minori con conseguente errore medio più basso.
Per questi motivi possiamo concludere l'analisi dicendo che, considerando le performance dei vari algoritmi eseguiti sui 13 grafi, l'utilizzo dell'euristica costruttiva *Nearest Neighbor* risulta essere la migliore sia in termini di tempo di esecuzione che di approssimazione del risultato.

Conclusioni

Con questa esperienza abbiamo potuto mettere mano su tre algoritmi per tentare di risolvere il problema del commesso viaggiatore (Traveling Salesman Problem) e di studiare i comportamenti dei vari algoritmi in termini di tempo di esecuzione e precisione della soluzione ritornata.
Tra i tre algoritmi visti abbiamo che solamente l'algoritmo di *Held-Karp* trova le soluzioni corrette ma, vista la sua complessità, riesce a farlo solamente con grafi che hanno un numero di nodi molto basso. Ecco quindi che entrano in gioco gli altri due algoritmi e una branca dell'algoritmica che non avevo mai affrontato ovvero quella che contempla l'approssimazione dei risultati.
Dei due algoritmi di approssimazione visti l'euristica *Nearest Neighbor* risulta essere leggermente migliore sia in termini di tempo di esecuzione che di precisione della soluzione approssimata trovata.
Concludo dicendo che è stato un laboratorio molto interessate sopratutto perché è stato possibile vedere e mettere mano su algoritmi che, cambiando anche di poco la taglia dei dati in input che gli vengono forniti, ottengono delle degenerazioni di prestazioni, in termini di complessità, notevoli. Questo mi ha fatto capire che la progettazione di un algoritmo è un attività ancora più profonda,complessa e ragionata rispetto a quanto visto durante la triennale e come una conoscenza consapevole e approfondita di tecniche, strutture dati ed euristiche possa aiutare notevolmente ad uscire da situazioni che sembrano non avere soluzione.