Laboratorio 2

Algoritmi Avanzati

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

Maggio 2021

Corso di Laurea Magistrale in Informatica, Dipartimento di Matematica, UniPD

Indice

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

Precedentemente è stato detto che il fatto di risolvere il TSP corrisponde a risolvere il problema della ricerca di un circuito Hamiltoniano. Quest'ultimo problema è noto essere un problema *NP-completo*, pertanto anche TSP sarà NP-completo. Il fatto di stabilire che TSP è un problema NP-completo, indica che molto probabilmente non esiste una soluzione polinomiale al problema.

Per poter stabilire con precisione la sua complessità dobbiamo prima "trasformarlo" in un problema di decisione, aggiungendo un limite k per il peso del ciclo all'input del problema.

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

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

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

Eseguo $A_\rho(G') \to C$ (ciclo), $c(C)$(costo di C) 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. $G \notin HAMILTON \Rightarrow C$ contiene almeno un lato non in $G \Rightarrow c(C) \geq \rho|V| + 1$.

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 $h = v_1, ..., v_n$. 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.

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 stuttura 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.

La struttura dati per il grafo è stata implementata nel seguente modo:

2.1.2 Implementazione

2.2 Held e Karp

2.2.1 Introduzione

2.2.2 Implementazione

2.2.3 Ottimizzazioni implementate

2.3 Nearest Neighbor

2.3.1 Introduzione

L'algoritmo *Nearest Neighbor* per il problema del Traveling Salesman Problem rientra nelle cosidette *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 fasi: inizializzazione, selezione, inserimento. Viene qui illustrato l'algoritmo in pseudocodice, e di seguito vengono analizzate nello specifico le tre fasi.

```
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 di distanza minima dall'ultimo nodo inserito nel path
    nextNode <- MinAdj(path, path.lastV)

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

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

return path


MinAdj(G=(V,E), path, lastV)
    return minimum distance node from lastV in {E-path}
```

Le tre fasi quindi sono:

1. **Inizializzazione**: si parte con il cammino composto unicamente dal primo vertice;
2. **Selezione**: sia $(v_0, ..., v_k)$ 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.

Le fasi 2 e 3 vengono ripetute 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 infatti necessaria per definizione del problema.

2.3.3 Fattori di approssimazione

2.3.4 Implementazione

2.3.5 Ottimizzazioni implementate

2.4 2-approssimato

2.4.1 Introduzione

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

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

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 $c(u, w, v)$. 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 dalll'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 $H^{ottimo} =< v_{j1}, ..., v_{jn}, v_{j1} >$. Sia $H'^{ottimo} =< v_{j1}, ..., v_{jn} >$ un cammino (è uno spanning tree, non un ciclo) hamiltoniano. Quindi, $c(H') \geq c(T)$, dove T è il MST, e $c(H^{ottimo}) \geq c(H'^{ottimo})$ prechè 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 Risposte alle domande

3.1 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 $(SoluzioneTrovata - SoluzioneOttima)/SoluzioneOttima$*. Potete aggiungere altra informazione alla tabella che ritenete interessanti.*

3.2 Domanda 2

Commentate i risultati che avete ottenuto: come si comportano gli algoritmi rispetti 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 ...

4 Conclusione