Università degli Studi di Torino

Dipartimento di informatica

Tesina di Ottimizzazione Combinatoria

Branch and Bound per TSP

Leonardo Magliolo
844910

Gianmario Cuccuru
984100

ANNO ACCADEMICO 2023/2024

Abstract

Il seguente lavoro è stato sviluppato come tesina implementativa per il corso di Ottimizzazione Combinatoria e verte sull'analisi del Travelling Salesman Problem, concentrandosi in particolare sul rilassamento basato su 1-Tree proposto da Held & Karp negli anni '70. Dopo un'introduzione teorica al problema, verrà esaminato nel dettaglio il rilassamento proposto da Held & Karp, il quale consente di ottenere un problema più semplice da risolvere. Successivamente, sarà proposto uno schema di Branch del problema per la risoluzione tramite il metodo del Branch and Bound. Infine, verrà fornita un'analisi dell'implementazione realizzata e dei risultati ottenuti.

Indice

Capitolo 1

Introduzione

Dato un insieme di città e il relativo costo di transito tra ogni coppia di città, il problema del commesso viaggiatore (Traveling Salesman Problem), o in breve TSP, consiste nel trovare il percorso più economico che consenta di visitare tutte le città una ed una sola volta e tornare infine al punto di partenza. Il percorso rappresenta semplicemente l'ordine in cui le città vengono visitate ed è chiamato tour o circuito.

Il problema del commesso viaggiatore, pur essendo semplice, è stato oggetto di numerose ricerche e ha ampie applicazioni pratiche oltre a quella classica del routing, come ad esempio lo scheduling delle macchine, il clustering e il cablaggio di computer.

Il problema del commesso viaggiatore è però un problema NP-completo e, allo stato dell'arte, non sono noti algoritmi capaci di risolverlo in tempo polinomiale. La base delle tecniche di risoluzione per il TSP è costituita dal Branch and Bound, per il quale sono essenziali delle buone tecniche di bounding. In questo caso, la bontà delle tecniche di bounding è determinata da due criteri spesso in conflitto tra loro: il limite deve essere stretto e veloce da calcolare.

1.1 Il problema del commesso viaggiatore

Il Traveling Salesman Problem può essere rappresentato mediante un grafo non orientato $G = (V, E)$ con costi c_{ij} associati agli archi. L'obiettivo è trovare un insieme di archi $C^* \subseteq E$ con il costo minimo possibile. Questo insieme C^* deve formare un circuito Hamiltoniano, ovvero un ciclo che attraversi ogni nodo del grafo una ed una sola volta. In questo lavoro viene considerata la versione simmetrica del TSP, in cui per ogni coppia di nodi $i, j \in V$ vale $c_{ij} = c_{ji}$. In questa versione, ogni arco può essere percorso in entrambe le direzioni con lo stesso costo. Nel contesto del traveling salesman problem, i vertici sono talvolta chiamati città e il circuito Hamiltoniano è comunemente noto come traveling salesman tour.

1.2 NP-completezza del TSP

Come accennato prima, il problema del commesso viaggiatore è noto essere un problema NP-completo. Per dimostrare ciò dobbiamo per prima cosa dimostrare che il TSP appartiene ad NP. Per verificare la validità di un tour, controlliamo che il tour visiti ciascun vertice una ed una sola volta. Poi sommiamo il costo totale degli archi e verifichiamo se il costo totale è minimo. Tutto ciò può essere completato in tempo

polinomiale, dimostrando quindi che il TSP appartiene ad NP. In secondo luogo, dimostriamo che il TSP è NP-hard. Un modo per fare ciò è dimostrare che il problema del ciclo Hamiltoniano, che è noto essere NP-completo, può essere ridotto al TSP. Supponiamo di avere un grafo $G = (V, E)$, il quale rappresenta un'istanza del problema del ciclo Hamiltoniano. Costruiamo poi un grafo completo $G' = (V, E')$, dove ogni coppia di vertici è collegata da un arco. La funzione di costo è definita come:

$$t(i,j) = \begin{cases} 0 & \text{se } (i,j) \in E \\ 1 & \text{se } (i,j) \notin E \end{cases}$$

Se G possiede un ciclo Hamiltoniano, allora esiste un ciclo in G' con costo totale 0, poiché tutti gli archi del ciclo Hamiltoniano originale sono presenti in E e hanno quindi costo 0 in G'. Viceversa, se G' ha un ciclo di costo totale 0, tutti gli archi di questo ciclo devono essere presenti in E, il che significa che G possiede un ciclo Hamiltoniano.

Questo mostra che se possiamo risolvere il TSP, possiamo risolvere anche il problema del ciclo Hamiltoniano, dimostrando così che il TSP è NP-hard.

Combinando questi due punti, possiamo concludere che il TSP è NP-completo.

1.3 Formulazione matematica

Il TSP simmetrico è definito su un grafo non orientato G=(V,E) con costi c_e associati ad ogni arco $e \in E$. Una possibile formulazione matematica per il TSP utilizza delle variabili per ogni arco:

$$\forall e \in E, x_e = \begin{cases} 1 & \text{se } e \text{ è nel ciclo} \\ 0 & \text{altrimenti} \end{cases}$$

$$\begin{aligned} & \min \sum_{e \in E} c_e x_e \\ (1) \quad & s.t. \sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \\ (2) \quad & \sum_{e \in E} x_e \leq |S| - 1 \quad \forall\, S \subset V : 3 \leq |S| \leq |V| - 1 \\ (3) \quad & x_e \in \{0, 1\} \quad \forall e \in E \end{aligned}$$

L'obiettivo è quello di minimizzare la somma dei costi degli archi selezionati. Il vincolo (1) impone che ogni nodo debba essere connesso esattamente da due archi e assicura che ogni nodo del grafo sia incluso nel circuito Hamiltoniano. Il vincolo (2) equivale invece all'eliminazione di tutti i sotto-cicli: $E(S) \subseteq E$ è l'insieme degli archi che hanno entrambi gli estremi in $S \subset V$. I sotto-cicli di ordine 2 sono già esclusi dal vincolo (1). Il vincolo (3), infine, definisce il dominio delle variabili decisionali che assumono valore 1 se l'arco viene inserito nel circuito Hamiltoniano, 0 altrimenti.

Capitolo 2

Il lavoro di Held and Karp

Il problema del commesso viaggiatore (TSP), pur essendo semplice, è stato oggetto di molte ricerche e ha ampie applicazioni pratiche. Essendo NP-completo, sono stati proposti diversi approcci per risolverlo in modo efficiente, come ad esempio i metodi esatti e quelli euristici. I solver esatti non devono solo identificare la soluzione ottimale, ma dimostrarne anche la bontà, spesso tramite un limite duale.

Held and Karp (1970) hanno proposto un rilassamento che fornisce dei limiti duali forti. Ad esempio, nella pratica, questi limiti sono utilizzati in Concorde, il solver presente allo stato dell'arte (Applegate et al. 2006) o nel design di vincoli globali nella constraint programming (Benchimol et al. 2010, 2012). Successivamente è stato proposto sempre da Held and Karp (1971) un algoritmo di branch-and-bound che utilizza questo rilassamento e che ha permesso di dimostrare l'ottimalità per diverse istanze di benchmark disponibili al momento della sua pubblicazione.

Questo algoritmo sfrutta una struttura chiamata minimum 1-tree che può servire come rilassamento valido per il TSP e ottenere così dei limiti duali.

2.1 1-tree

Trovare soluzioni ottimali per istanze del TSP di grandi dimensioni richiede degli approcci sofisticati a causa dell'esplosione combinatoria dello spazio delle soluzioni. Nel metodo del branch and bound, i limiti vengono utilizzati per potare l'albero di ricerca e accelerare di conseguenza la ricerca stessa, consentendo così ai solver di dimostrare l'ottimalità senza esplorare l'intero albero. Per ottenere ciò, il rilassamento di Held and Karp (1970) offre un limite duale robusto basato su una variante del minimum spanning tree.

Sia G(V,E) un grafo completo con costi c_e associati ad ogni arco $e \in E$. Un minimum 1-tree è un minimum spanning tree di $G \setminus \{1\}$ a cui viene aggiunto il nodo 1 insieme alla coppia di archi di costo minimo che lo collegano all'albero. È possibile notare che la scelta del nodo 1 è arbitraria e dipende dall'etichettatura di V. La formulazione matematica del minimum 1-tree è la seguente:

$$
\begin{aligned}
& \min \sum_{e \in E} c_e x_e \\
(1) \quad & s.t. \sum_{e \in \delta(1)} x_e = 2 \\
(2) \quad & \sum_{e \in E} x_e = |V| \\
(3) \quad & \sum_{\substack{i,j \in S \\ i<j}} x_{i,j} \leq |S| - 1 \quad \forall\, S \subset V \backslash \{1\} \wedge |S| \geq 3 \\
(4) \quad & x_e \in \{0, 1\}
\end{aligned}
$$

I vincoli (1) e (2) definiscono la struttura del 1-tree e il vincolo (3) impone l'eliminazione dei subtour. Il problema consiste quindi nel trovare un algoritmo che possa risolvere efficientemente il problema dell'individuazione di un minimum spanning tree sul grafo G. Nella formulazione di cui sopra, $\delta(v)$ denota gli archi contenenti il vertice $v \in V$, mentre $c_e \in \mathbb{R}$ rappresenta il costo di un arco $e \in E$. Infine, $x_e \in \{0, 1\}$ è la variabile di decisione che indica se l'arco e è incluso o meno nel 1-tree.

È possibile notare che ogni tour in G è un 1-tree e se un minimum 1-tree è un tour, allora quest'ultimo è una soluzione ottimale per il TSP. Pertanto, ogni minimum 1-tree è un rilassamento valido per il TSP. Tuttavia, non è detto che una soluzione di questo programma intero sia un tour. Per farlo, è necessario applicare un nuovo set di vincoli:

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall\, v \in V \setminus \{1\}$$

Questi vincoli forzano ogni nodo ad avere solo due archi, uno in entrata e uno in uscita, trasformando così il problema nella ricerca di un ciclo Hamiltoniano di costo minimo, che sappiamo essere un problema NP-hard.

2.2 Rilassamento Lagrangiano

Per ottenere un rilassamento valido per il 1-tree in modo efficiente, è possibile spostare i vincoli di cui sopra, uno per ogni nodo, nella funzione obiettivo e penalizzare le loro violazioni associando ad essi degli opportuni moltiplicatori lagrangiani $\theta_v \in \mathbb{R}$ per ogni $v \in V \backslash \{1\}$. La formulazione matematica del minimum 1-tree con l'introduzione dei moltiplicatori Lagrangiani, ottenuta mediante rilassamento Lagrangiano del vincolo (1) sul problema del ciclo Hamiltoniano di costo minimo, è la seguente:

$$\min \sum_{e \in E} c_e x_e - \sum_{v \in V \setminus \{1\}} \theta_v \left(2 - \sum_{e \in \delta(v)} x_e \right)$$

$$(1) \quad \sum_{e \in \delta(1)} x_e = 2$$

$$(2) \quad \sum_{e \in E} x_e = |V|$$

$$(3) \quad \sum_{\substack{i,j \in S \\ i<j}} x_{i,j} \leq |S| - 1 \quad \forall S \subset V \setminus \{1\} \wedge |S| \geq 3$$

$$(4) \quad x_e \in \{0, 1\}$$

In questo modo, intuitivamente, ogni nodo con un grado diverso da due verrà penalizzato. Un rilassamento ottimale per il 1-tree può essere individuato andando ad ottimizzare sulle variabili θ_v. A tal proposito, Held e Karp (1970, 1971), hanno proposto un approccio iterativo. L'idea è quella di aggiustare i moltiplicatori lagrangiani θ, passo dopo passo, per costruire una sequenza di 1-tree che fornisca dei limiti sempre migliori. Una panoramica del processo è proposta nella figura sotto.

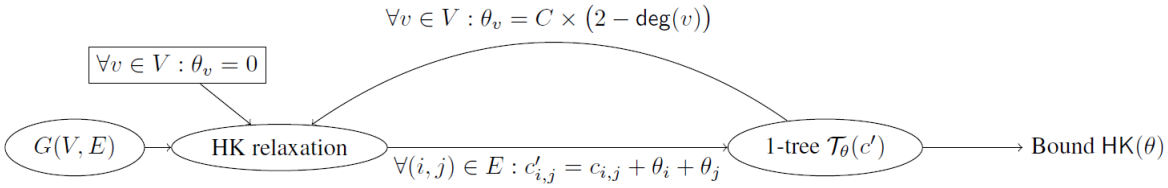

Figura 2.1: Approccio di Held and Karp

In primo luogo, viene calcolato un primo minimum 1-tree cercando un minimum spanning tree su $G \setminus \{1\}$ e aggiungendo i due archi più economici incidenti sul nodo 1. Se il minimum 1-tree ottenuto è un tour, allora esso corrisponde alla soluzione ottimale del TSP. Altrimenti, vengono penalizzati alcuni nodi in quanto almeno un nodo ha un grado superiore a 2.

L'idea principale di Held e Karp (1970, 1971) è quella di penalizzare tali nodi modificando il costo c_{ij} degli archi $(i, j) \in E$, in base ai valori di θ_i e θ_j. Siano $c'_{ij} \in \mathbb{R}$ i costi modificati. Essi sono calcolati come segue.

$$c'_{ij} = c_{ij} + \theta_i + \theta_j \quad \forall\, (i, j) \in E$$

Una proprietà teorica dimostrata da Held e Karp (1970, 1971) è che un tour ottimale per il TSP è invariante sotto questa perturbazione, mentre un 1-tree ottimale non lo è. Ciò consente di migliorare la soluzione trovando dei moltiplicatori migliori. L'equazione sotto propone una scelta standard per calcolare i moltiplicatori, dove $C \in \mathbb{R}$ è una costante arbitraria e $deg(v)$ indica il grado del nodo $v \in V$ nel 1-tree corrente.

$$\theta_v = C \times \big(2 - deg(v)\big) \quad \forall\, v \in V$$

A questo punto, può essere calcolato un nuovo minimum 1-tree dal grafo con i costi aggiornati c'_{ij}. Questo 1-tree è indicato con $T_\theta(c')$, dove $c' = \{c_1, ..., c_{|E|}\}$ è l'insieme di tutti i costi modificati e $\theta = \theta_1, ..., \theta_{|V|}$ è l'insieme di tutti i moltiplicatori. Viene utilizzata anche la notazione $cost(T_\theta(c'))$ per fare riferimento al costo totale del 1-tree.

Questo processo viene ripetuto e si ottiene un nuovo 1-tree $T_\theta(c')$ fino a quando non si ottiene più nessun miglioramento, ovvero quando si raggiunge un minimo locale. Il costo del 1-tree ottimale fornisce un limite inferiore alla funzione obiettivo:

$$HK(\theta) = cost\big(T_\theta(c')\big) - 2\sum_{i=1}^{|V|}\theta_i \tag{2.1}$$

Questo limite, $HK(\theta)$, è comunemente noto in letteratura come limite di Held-Karp. Questo approccio è tipicamente incorporato nell'algoritmo di branch-and-bound, che utilizza questo limite per potare la ricerca.

2.2.1 Un approccio alternativo

Nel lavoro di Parjadis et al. del 2023 [3] viene proposto un approccio alternativo per l'aggiornamento dei moltiplicatori lagrangiani. Il limite di Held-Karp $HK(\theta)$ ha due interessanti proprietà: (1) può essere parametrizzato grazie ai moltiplicatori lagrangiani θ e (2) è sempre valido, cioè non supera mai il costo ottimale del TSP. Entrambe le proprietà offrono l'opportunità di utilizzare un approccio basato sull'apprendimento per calcolare il limite. A tal fine, viene proposto di costruire un modello $\Phi_w : G(V, E) \to \mathbb{R}^{|V|}$ in grado di prevedere direttamente tutti i moltiplicatori θ per un'istanza di TSP data in input (cioè un grafo). Il modello è parametrizzato con p parametri $w = \{w_1, \cdots, w_p\}$. I vantaggi sono due. In primo luogo, si eliminano alcune parti del processo iterativo di Held e Karp (1970) e si risparmia tempo di esecuzione. In secondo luogo, consente di ottenere potenzialmente dei limiti più stretti. Il processo è illustrato nella Figura 2.2.

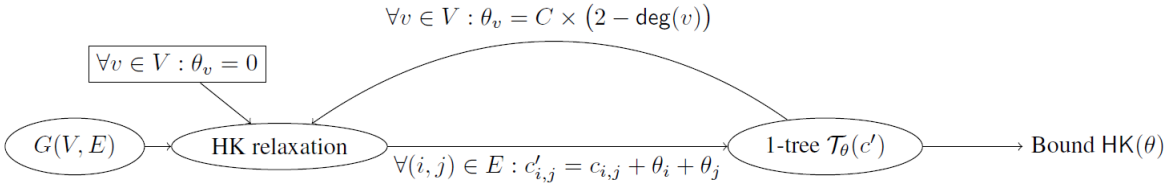

Figura 2.2: Approccio di Parjadis et al.

L'obiettivo è trovare i parametri del modello w che producono il limite più alto possibile. Ciò corrisponde a un problema di massimizzazione che può essere risolto con un'ottimizzazione basata sul gradiente. Di seguito viene indicata la formulazione per il problema di massimizzazione del bound e del suo gradiente.

$$\max_w HK\big(\Phi_w(G)\big) \to \nabla_w HK\big(\Phi_w(G)\big) \tag{2.2}$$

Capitolo 3

Branch and Bound

Esistono diversi approcci per la determinazione esatta di un problema NP-completo come il TSP; tra questi, quelli di enumerazione implicita come il branch and bound sono i più diffusi. Questi algoritmi esplorano in modo sistematico lo spazio delle soluzioni alla ricerca di una soluzione ottima. I lower e upper bound sul valore ottimo della funzione obiettivo vengono sfruttati per ottenere delle informazioni sul problema, permettendo così di escludere dalla ricerca aree dello spazio delle soluzioni in cui non è presente la soluzione ottima; queste aree si dicono visitate implicitamente dall'algoritmo

Comunque, anche usando le migliori tecnologie disponibili, non si può mai escludere l'eventualità di dover esaminare una frazione consistente dello spazio delle soluzioni.

3.1 Uno schema generale

Gli algoritmi di enumerazione implicita possono essere visti come un caso particolare del noto schema algoritmico "divide et impera", che affronta la soluzione di un problema mediante i seguenti passi:

- suddividere il problema in un certo numero di sottoproblemi "più piccoli";
- risolvere separatamente i singoli sottoproblemi, tipicamente applicando ricorsivamente lo stesso procedimento finché la soluzione non può essere ottenuta con qualche procedimento alternativo (caso base);
- combinare le soluzioni dei singoli sottoproblemi in una soluzione per il problema originale.

In altre parole, suddividendo un problema in un certo numero di sottoproblemi, tale per cui l'unione delle regioni ammissibili contenga la regione ammissibile del problema originario, ed ottenendo una valutazione superiore ed inferiore sul valore ottimo della funzione obiettivo di ciascuno dei sottoproblemi individuati, si ottengono una valutazione superiore ed inferiore per il valore ottimo della funzione obiettivo del problema originale.

3.2 Calcolo del lower bound

Come accennato precedentemente, per determinare un minimum 1-tree e ottenere quindi un lower bound per il problema del TSP, è possibile calcolare un primo minimum 1-tree cercando un minimum spanning tree su $G\backslash\{n\}$ e aggiungendo i due archi più economici incidenti sul nodo n.

Il costo della procedura sopra descritta è dominato dal calcolo del minimum spanning tree, che può essere risolto facilmente utilizzando un algoritmo come quello di Kruskal, il quale ha costo computazionale pari a $O(m \log n)$. La selezione degli archi di costo minimo può essere invece effettuata con una semplice scansione degli archi in $O(m)$.

Il calcolo e l'aggiornamento del lower bound si basano sull'identificazione di soluzioni ammissibili durante l'esecuzione dell'algoritmo. Se il 1-tree calcolato risulta essere un circuito hamiltoniano, allora il lower bound può essere aggiornato se il suo costo è minore di quello attualmente noto. Il lower bound verrà inizializzato a $+\infty$ all'inizio della procedura.

3.3 Schema di branching

Introduciamo a questo punto il modo più utilizzato per implementare l'operazione di separazione, ovvero la suddivisione di X in $X_1, X_2, \ldots, X_k$. Per semplificare, supponiamo inizialmente che sia $X \subseteq \{0,1\}^n$, ossia che le soluzioni del problema possano essere descritte attraverso n decisioni binarie, ciascuna rappresentata da una variabile $x_i, i = 1, \ldots, n$: un modo possibile per suddividere X è quello di prendere decisioni su alcune delle variabili. Ad esempio, fissato un qualsiasi indice i, possiamo partizionare X come $X_0 \cup X_1$, dove

$$X_0 = \{x \in X : x_i = 0\} \quad \text{e} \quad X_1 = \{x \in X : x_i = 1\}$$

Inoltre, gli insiemi così ottenuti possono essere a loro volta partizionati secondo lo stesso schema. In effetti, è possibile rappresentare l'insieme ammissibile X attraverso un albero decisionale (o albero di branch), il quale associa a ciascuna soluzione ammissibile una sequenza di decisioni che la genera.

Nel caso particolare del TSP, se la soluzione del rilassamento ottenuto risulta essere un circuito hamiltoniano, allora il nodo corrispondente viene chiuso per ammissibilità. Se invece il nodo non viene chiuso per ammissibilità, quello che vogliamo ottenere è impedire che nei nodi figli si vada a formare lo stesso circuito presente nel nodo padre e lo facciamo introducendo la regola di suddivisione che verrà adesso illustrata e che consente di costruire un albero decisionale esattamente come illustrato in precedenza.

Indichiamo con $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ gli archi che compongono tale circuito. La regola di suddivisione opera nel seguente modo: il primo figlio verrà generato imponendo che l'arco (i_1, j_1) non faccia parte della soluzione, ossia impostando $x_{i_1 j_1} = 0$; il secondo figlio verrà generato imponendo invece che sia presente l'arco (i_1, j_1) ma assente l'arco (i_2, j_2), ovvero impostando $x_{i_1 j_1} = 1$ e $x_{i_2 j_2} = 0$. Il processo viene iterato in questo modo fino ad arrivare all'r-esimo figlio, il quale avrà tutti i primi r-1 archi presenti nel circuito ma non conterrà l'ultimo. Quindi, ad ogni nodo dell'albero di branch così costruito sarà associato sia l'insieme E_0, il quale

contiene tutti gli archi che NON fanno parte della soluzione, sia l'insieme E_1 che contiene tutti gli archi che invece fanno parte della soluzione. È essenziale che E_0 ed E_1 siano disgiunti, ossia $E_0 \cap E_1 = \varnothing$.

Ciò consente di risolvere per ogni figlio un sottoproblema del tipo $S = (E_0, E_1)$, contenente tutti i circuiti hamiltoniani formati dagli archi presenti in E_1 ma non dagli archi presenti in E_0. Per ciascun sottoproblema, il calcolo del lower bound avviene secondo la procedura descritta nel paragrafo precedente, tenendo però conto nel calcolo dell'MST degli archi che devono essere presenti (presenti in E_1) e di quelli che devono essere esclusi (presenti in E_0). Nella nostra implementazione, il calcolo dell'MST avviene utilizzando l'algoritmo di Kruscal, il quale sceglie gli archi con il minor peso in ordine crescente, evitando cicli. Durante l'applicazione di Kruscal, invece di inizializzare l'insieme E_t con l'insieme vuoto, questo verrà inizializzato con gli archi presenti in E_1 che non sono incidenti sul nodo n; ovviamente non dovranno essere presi in considerazione gli archi presenti in E_0. Una volta calcolato l'albero di copertura T, verranno ad esso aggiunti i due migliori archi incidenti su n, ossia quelli con costo minore.

3.4 Chiusura dei nodi

Vengono adesso definiti i criteri di chiusura per un generico nodo dell'albero di branch P_i.

Se $\hat{z} < LB(P_i)$, ossia quando il lower bound ottenuto è maggiore del migliore circuito hamiltoniano fin ora trovato, allora si può evitare di esplorare ulteriormente la parte dello spazio delle soluzioni rappresentata dal nodo. In questo caso si dice che il nodo è stato potato dalla valutazione superiore, oppure in altre parole viene chiuso per bound. Infatti, il nodo viene scartato e si passa a visitare un altro dei nodi attivi (se ve ne sono) Un caso particolare è quello in cui non si riesce a calcolare un 1-tree per il nodo; in questo caso non esiste alcuna soluzione ammissibile per il rilassamento considerato e il nodo viene chiuso (o potato) per inammissibilità.

È bene rimarcare che i nodi "potati" sono quelli in cui la visita si interrompe, ossia le foglie del sottoalbero che viene effettivamente generato non necessariamente sono, e sperabilmente non sono, foglie dell'intero albero decisionale. Eliminare un nodo permette di non visitare esplicitamente tutto il corrispondente sottoalbero dell'albero decisionale: per questo si parla di potatura.

Se il 1-tree generato dal rilassamento risulta essere un circuito hamiltoniano, allora non è necessario esplorare ulteriormente. Il nodo viene chiuso perché è un possibile candidato ottimo (potato per ammissibilità). Qualora il nodo presentasse anche un costo migliore, ossia minore dell'attuale soluzione trovata, allora quest'ultima verrebbe aggiornata.

3.5 Strategie di visita

La strategia di visita dell'albero decisionale è fondamentalmente dettata dalla strategia di selezione del prossimo nodo da visitare dall'insieme Q. Si distingue usualmente

tra visite topologiche e visite basate sull'informazione. Le visite topologiche scelgono il prossimo nodo da visitare unicamente sulla base della struttura topologica dell'albero decisionale; ciò corrisponde a strategie di selezione del nodo che dipendono unicamente dalla sua posizione in Q. Le due strategie di visita topologica più note sono quella a ventaglio, o breadth-first, corrispondente ad implementare Q come una coda, e quella a scandaglio, o depth-first, corrispondente ad implementare Q come una pila.

Di queste, nel contesto degli algoritmi di enumerazione implicita può essere particolarmente utile la strategia depth-first. Poichè per potare i nodi attraverso la valutazione superiore è necessario disporre di una valutazione inferiore, l'uso di una strategia depth-first può essere indicato in quanto porta la visita velocemente verso le foglie dell'albero decisionale, e quindi può consentire di generare velocemente soluzioni ammissibili. Le visite topologiche offrono alcuni vantaggi dal punto di vista dell'implementazione. In primo luogo sono semplici da realizzare ed il costo di gestione di Q è basso. In secondo luogo, la strategia depth-first si presta ad essere implementata in modo tale da mantenere molto basso il numero di nodi attivi nell'albero decisionale.

Dal punto di vista della complessità spaziale, la strategia breadth-first richiede una quantità di memoria esponenziale rispetto alla profondità dell'albero, poiché tutti i nodi a un certo livello dell'albero devono essere memorizzati contemporaneamente. Al contrario, la strategia depth-first è molto più efficiente poiché richiede di memorizzare solo una sequenza di nodi lungo un percorso dall'origine alla foglia.

Le regole di visita basate sull'informazione utilizzano invece l'informazione sui nodi in Q per determinare il prossimo nodo da estrarre; in altri termini, corrispondono ad implementare Q come una coda di priorità. La strategia con informazione più usata è la best-first, in cui ad ogni nodo viene associato il valore della valutazione superiore prodotta dal corrispondente rilassamento e viene selezionato il nodo con valore maggiore, che corrisponde al sottoalbero "più promettente", ossia nel quale si dovrebbero avere maggiori probabilità di incontrare la soluzione ottima. La strategia best-first è molto usata in quanto solitamente riesce ad indirizzare la visita verso le zone "più promettenti" dell'albero decisionale. Per contro, la strategia best-first è più complessa da implementare, e tipicamente accade che vengano esaminati in sequenza nodi "lontani" nell'albero decisionale, che corrispondono a problemi "molto diversi" tra loro. Presenta inoltre una complessità spaziale maggiore rispetto alle visite topologiche, poiché l'uso di una coda di priorità comporta la necessità di memorizzare e ordinare un numero potenzialmente elevato di nodi. Questo si traduce in un consumo significativo di memoria, soprattutto nel caso di alberi molto grandi, rendendo così la best-first meno pratica in contesti con risorse di memoria limitate.

Capitolo 4

Implementazione

Andremo a descrivere in questa sezione le scelte implementative utilizzate per la risoluzione del TSP mediante il metodo branch-and-bound.

4.1 Strumenti utilizzati

Durante la fase di sperimentazioni abbiamo deciso di utilizzare i seguenti strumenti:

- **Java** come linguaggio di programmazione, data la sua nota efficienza e semplicità rispetto ad altri linguaggi come C. Infatti Java gestisce automaticamente la memoria e non utilizza puntatori, rendendo così il codice più leggero e manutenibile. Tuttavia, proprio la gestione implicita della memoria tramite garbage collector può causare un consumo di memoria maggiore rispetto a C.

- **TSPLib**, libreria utilizzata per testare e valutare l'algoritmo implementato. TSPLib fornisce una raccolta di istanze di problemi standardizzati, con dati e soluzioni ottimali o migliori conosciute, che consentono di confrontare le prestazioni degli algoritmi su un set comune di problemi.

4.2 Strutture dati

La struttura dati principale che abbiamo deciso di utilizzare è rappresentata dalle liste di adiacenza, implementate mediante HashMap, sia per la rappresentazione del grafo che del 1-tree. Questa struttura dati offre diversi vantaggi in termini di complessità computazionale e utilizzo di memoria.

La rappresentazione mediante HashMap richiede uno spazio di memoria proporzionale al numero di nodi più il numero di archi ($O(n+m)$), che è significativamente più efficiente rispetto ad una matrice di adiacenza che richiederebbe uno spazio di memoria $O(n^2)$. Le operazioni centrali nella nostra implementazione includono la scansione degli adiacenti di un nodo e l'esplorazione del grafo. Con le liste di adiacenza implementate mediante HashMap, la complessità di una scansione degli adiacenti di un nodo è $O(|A(u)|)$, dove $A(u)$ è l'insieme degli adiacenti del nodo u.

4.3 Calcolo del MST

Per il calcolo dell'MST abbiamo implementato l'algoritmo greedy di Kruskal. L'algoritmo ordina gli archi secondo costi crescenti e costruisce un insieme ottimo T di archi, scegliendo di volta in volta un arco di peso minimo che non forma cicli con gli archi già scelti. Per far questo, gestisce una partizione $W = \{W_1, W_2, \ldots, W_k\}$ di V, insieme dei nodi del grafo, in cui ogni W_i rappresenta un insieme di nodi per cui è stato scelto un insieme di archi che li collega. Inizialmente, $T = \varnothing$ e $W = \{\{1\}, \{2\}, \ldots, \{n\}\}$, poiché nessun arco è stato scelto e quindi nessun nodo è stato collegato. Alla prima iterazione viene scelto il nodo (u, v) di peso minimo; questo viene posto in T e gli insiemi $\{u\}$ e $\{v\}$ vengono sostituiti con l'insieme $\{u, v\}$. Alla generica iterazione i, esaminiamo l'arco (x, y) con i-esimo costo che viene aggiunto alla soluzione T solo se i nodi x e y non appartengono allo stesso insieme della partizione W (cioè se l'arco non forma cicli con gli archi inseriti in precedenza). In questo caso, dopo aver inserito l'arco (x, y) in T, si sostituiscono nella partizione W gli insiemi (distinti) contenenti x e y (Find-Set) con la loro unione (Union-Set). L'ordinamento di $m = |E|$ archi richiede $O(m \log m)$ passi. Il costo delle m iterazioni dipende dal costo delle uniche operazioni non costanti, ossia Find-Set e Union-Set.

4.4 Strategie di visita

Abbiamo scelto di implementare due versioni diverse per il branch and bound: una versione multi-thread con l'utilizzo della strategia Best-first search e una versione single-thread con l'utilizzo della strategia Depth-first search. Questa scelta è dovuta a considerazioni relative a efficienza e applicabilità.

La strategia Best-first search esplora i nodi dell'albero di ricerca che corrispondono ai sottoalberi "più promettenti", ossia nei quali si avrebbe maggiore probabilità di incontrare la soluzione ottima. In questa strategia, i problemi fratelli risultano indipendenti l'uno dall'altro. Questo significa che ciascun sotto-problema può essere risolto in parallelo senza la necessità di sincronizzazione, sfruttando appieno le potenzialità date dalla parallelizzazione. Ciò consente di ridurre significativamente il tempo di calcolo.

Nella strategia Depth-first search, l'esplorazione avviene approfondendo ciascun ramo dell'albero di ricerca prima di passare al prossimo. Questo crea una dipendenza sequenziale tra i nodi, rendendo difficile l'applicazione di una strategia multi-thread, in quanto ogni nodo dipende dalla completa esplorazione del nodo precedente. La strategia DFS può essere però efficace in situazioni dove la versione multi-thread fallisce, come nel caso in cui la memoria a disposizione non sia sufficiente. Quindi, la combinazione di una versione multi-thread con Best-first search e una versione single-thread con DFS consente di avere un approccio flessibile per la risoluzione del TSP, consentendo si utilizzare la DFS nei casi in cui la Best-first search non è efficace.

4.5 Struttura dell'implementazione

Il codice sviluppato è strutturato nel seguente modo:

- **BranchAndBoundTSP**: questo package contiene tutte le classi utili al fine di risolvere il problema del TSP utilizzando appunto l'algoritmo di Branch and Bound dettagliato precedentemente. Le classi principali includono l'implementazione effettiva dell'algoritmo di Branch and Bound stesso, la gestione dei problemi intermedi durante l'esecuzione dell'algoritmo e le eccezioni necessarie per gestire i casi non risolvibili.

- **Datastructures**: questo package contiene tutte le classi necessarie per la gestione delle strutture a grafo, l'implementazione dell'algoritmo di Kruscal per il calcolo dell'MST e l'implementazione della LIFO blocking queue.

- **TSPLib**: in questo package è presente il parser utilizzato per leggere e interpretare i file presenti nella libreria TSPLib, consentendo così la generazione del grafo corrispondente. Abbiamo utilizzato un parser preesistente, adattandolo ai nostri scopi specifici.

4.6 Test

Tutti i test sono stati svolti utilizzando un PC dotato di CPU AMD Ryzen 7 5800X e 32 GB di RAM. I risultati ottenuti sono i seguenti:

Burma 14 - BestFS		
NumThreads	Nodi	Tempo
1	1732736	64 s
2	1716906	54 s
8	1744534	31 s

Burma 14 - DFS		
NumThreads	Nodi	Tempo
1	1822381	70 s

Ulysses16 - BestFS		
NumThreads	Nodi	Tempo
1	3215759	12 m 32 s
2	3376546	10 m 54 s
8	3154971	7 m 8 s

Ulysses16 - DFS		
NumThreads	Nodi	Tempo
1	3837334	16 m 2 s

Gr17 - BestFS		
NumThreads	Nodi	Tempo
1	$>$ 4.5 M	$>$ 30 m, Out of memory
2	$>$ 4.5 M	$>$ 23 m, Out of memory
8	$>$ 4.5 M	$>$ 18 m, Out of memory

Gr17 - DFS		
NumThreads	Nodi	Tempo
1	5089038	33 m 34 s

Come è possibile evincere dalle tabelle precedenti, le istanze di TSP usate per i test includono Burma14, Ulysses16 e Gr17, tutte provenienti dalla libreria TSPLib.

I risultati dei test per la strategia BestFS mostrano che per Burma14, con l'aumento del numero di thread, il tempo di esecuzione diminuisce sostanzialmente, passando da 64s con 1 thread a 31s con 8 thread; questo mette in luce come l'algoritmo riesca a scalare meglio con l'utilizzo di più thread. Un comportamento simile si osserva per Ulysses16, in cui il tempo di esecuzione diminuisce da 12m 32s con 1 thread a 7m 8s con 8 thread, mostrando ancora una volta l'efficienza dell'esecuzione multithread. Tuttavia per Gr17 tutte le configurazioni portano ad un'esecuzione che termina in out of memory, suggerendo quindi che la dimensione del problema supera la memoria a disposizione.

Per quanto riguarda la strategia DFS, Burma14 ha completato l'esecuzione in 70s, Ulysses in 16m 2s e Gr17 in 33 m 34s mostrando quindi che la strategia DFS, sebbene più lenta di BestFs, non incorra in problemi di memoria.

In conclusione, i test mostrano chiaramente l'efficacia della strategia BestFS in combinazione con l'esecuzione multi-thread. Tuttavia, l'errore di out of memory per Gr17 suggerisce che, nonostante i vantaggi derivanti dall'esecuzione multi-thread, la gestione della memoria è un punto cruciale per istanze di grandi dimensioni. L'approccio DFS invece, pur essendo meno efficiente in termini d tempo rispetto a BestFS, mostra una maggiore stabilità per istanze che richiedono un uso intensivo della memoria.

Bibliografia

[1] D.L. Applegate, R.E. Bixby, V. Chvátal, and W.J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton University Press, 2011.

[2] Michael Held and Richard M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, 1970.

[3] Augustin Parjadis, Quentin Cappart, Bistra Dilkina, Aaron Ferber, and Louis-Martin Rousseau. Learning lagrangian multipliers for the travelling salesman problem, 2023.

[4] L. De Giovanni M. Di Summa. *Metodi e Modelli per l'Ottimizzazione Combinatoria*. Università di padova, 2020.