Branch & Bound per TSP simmetrico

Lorenzo Sciandra, Stefano Vittorio Porta

Anno accademico 20202021

Abstract – In questa tesina sarà analizzato e risolto il problema del TSP simmetrico con un Branch & Bound che fa uso degli 1-tree. A seguito della formulazione matematica del TSP sarà mostrato come un rilassamento lagrangiano su un insieme di vincoli ci conduca ad un problema molto più semplice da risolvere, polinomiale a tutti gli effetti. Dopo aver mostrato una procedura risolutiva del calcolo dell'1-tree ed uno schema di Branch del problema originario seguirà, a fine della trattazione teorica, l'analisi dell'implementazione effettuata in Java ed un esempio esplicativo.

1 Introduzione

Il problema del commesso viaggiatore, spesso indicato come **Travelling Salesman Problem** (**TSP**), è uno dei casi di studio classici dell'informatica teorica e della teoria della complessità computazionale. Il nome nasce dalla sua più tipica rappresentazione: "*Dato un insieme di città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta e ritornare alla città di partenza*". Al problema TSP è associabile un grafo non orientato $G = (V, E)$ con costi c_{ij} associati agli archi, da cui si richiede di determinare un insieme di archi $C^* \subset E$ con costo minimo possibile. Tale insieme C^* deve formare un **circuito hamiltoniano**, cioè un ciclo che passa una ed una sola volta per ogni nodo del grafo. Il problema che analizzeremo è la versione simmetrica del TSP ossia $c_{ij} = c_{ji}\ \forall i, j \in V$. Ogni arco è quindi percorribile in entrambe le direzioni spendendo lo stesso costo.
Il TSP è un problema NP-hard, nella precisione NP-completo e quindi algoritmi che lo risolvono all'ottimo richiedono una complessità in tempo più che polinomiale nell'istanza trattata ($P \neq NP$).
Analizzeremo in questa tesina un approccio risolutivo basato sul **Branch & Bound**, che garantisce di trovare l'ottimo del problema in un tempo che può essere esponenziale. L'idea alla base di questa tecnica algoritmica è quella di partizionare con la fase di **Branching** lo spazio delle soluzioni in più sottospazi (non per forza disgiunti) la cui unione restituisca però tassativamente la regione iniziale. Avendo a che fare con spazi ridotti questi risulteranno di più facile analisi attraverso l'applicazione della fase di **Bounding**, in cui saranno valutate le soluzioni raggiungibili da una ipotetica ricerca nello spazio degli stati, permettendo di potare interi sottoalberi in caso di inammisbilità o qualora la migliore soluzione garantita fosse peggiore di quella già ottenuta.

2 Formulazione Matematica

Un circuito è detto hamiltoniano se su ogni nodo incidono esattamente due archi e, rimuovendo un nodo n qualsiasi e i suoi due archi incidenti, si ottiene un albero sui rimanenti $|V|-1$ nodi. Una possibile formalizzazione matematica del tsp che tiene conto di questo risulta quindi la seguente:

$$\min z = \sum_{(i,j)\in E} c_{ij} \cdot x_{ij}$$

$$(1) \quad \sum_{j\in V, i\neq j} x_{ij} = 2 \quad \forall i \in V$$

$$(2) \quad \sum_{(i,j)\in E, i,j\neq n} x_{ij} = |V| - 2$$

$$(3) \quad \sum_{(i,j)\in E(U)} x_{ij} \leq |U| - 1 \quad \forall\, U \subseteq V \setminus \{n\} : |U| \geq 3$$

$$(4) \quad x_{ij} \in \{0,1\} \quad \forall (i,j) \in E$$

Il vincolo (1) impone che su ogni nodo $i \in V$ incidano esattamente due archi, mentre (2) e (3) garantiscono che una volta scelto un nodo n, $V \setminus \{n\}$ risulti un albero e quindi abbia $|V|-2$ archi e sia privo di circuiti.
Dato un sottoinsieme di nodi $U \subseteq V$ definiamo $E(U)$ come l'insieme di archi $\{(i,j) \mid i,j \in U\}$. Osservando che ogni circuito sui nodi in U deve avere $|U|$ archi in $E(U)$, per eliminare i circuiti abbiamo inserito il vincolo (3). Consideriamo $|U| \geq 3$, poichè se fosse per esempio uguale a 2 allora $E(U)$ conterrebbe solamente l'arco tra i due nodi e non ci sarebbe alcun circuito. Il vincolo (4) modella invece il dominio delle variabili decisionali che assumono valore 1 se il corrispondente arco viene inserito nel circuito hamiltoniano e 0 altrimenti.
Una volta che si ha a che fare con la formulazione matematica del TSP un possibile rilassamento e quindi un **Lower Bound** al problema potrebbe essere quello di applicare il rilassamento continuo al vincolo d'interezza delle variabili (4).
L'approccio che abbiamo noi adottato prevede invece un rilassamento Lagrangiano sul vincolo (1) che ci conduce al problema di trovare il minimo 1-Tree, una volta impostati i moltiplicatori lagrangiani λ_i a 0, ossia eliminando il vincolo (1) per tutti i nodi tranne che per il nodo selezionato n.

3 Rilassamento Lagrangiano

Una volta introdotti dei moltiplicatori lagrangiani λ_i per ogni nodo e portato in funzione obiettivo il vincolo otteniamo:

$$\min z = \sum_{(i,j)\in E} c_{ij} \cdot x_{ij} + \sum_{k\in V} \lambda_k \left(2 \; - \sum_{j\in V, k\neq j} x_{kj}\right)$$

$$(5) \quad \sum_{j\in V, j\neq n} x_{nj} = 2$$

$$(2) \quad \sum_{(i,j)\in E, i,j\neq n} x_{ij} = |V| - 2$$

$$(3) \quad \sum_{(i,j)\in E(U)} x_{ij} \leq |U| - 1 \quad \forall\, U \subseteq V \setminus \{n\} : |U| \geq 3$$

$$(4) \quad x_{ij} \in \{0,1\} \quad \forall (i,j) \in E$$

Per comodità notazionale abbiamo portato in funzione obiettivo anche il moltiplicatore λ_n per il nodo n, che verrà però impostato a 0. Dato che abbiamo rilassato vincoli di uguaglianza si

potranno considerare i migliori moltiplicatori lagrangiani non solo maggiori o uguali a 0, ma anche negativi. Per valori fissati dei vari λ_k il rilassamento lagrangiano è facilmente risolvibile con una procedura che mostreremo a breve, che consente di individuare l'1-tree di costo minimo. Riscrivendo la funzione obiettivo otteniamo:

$$l(\lambda) = \min \sum_{(i,j)\in E} (c_{ij} - \lambda_i - \lambda_j) \cdot x_{ij} + 2 \cdot \sum_{k\in V} \lambda_k$$

I costi associati agli archi risultano quindi aggiornati:

$$c'_{ij} = c_{ij} - \lambda_i - \lambda_j$$

Da questa riformulazione si può facilmente passare al duale lagrangiano che consisterà nell' individuazione dei valori $\lambda^* = (\lambda^*_k)_{k\in V}$ per cui il valore di $l(\lambda)$ sia il più grande possibile. Un possibile modo di procedere in questa direzione sarebbe quella di partire da possibili moltiplicatori lagrangiani (come $\lambda_k = 0, \forall k$) e poi migliorare la soluzione diminuendo il peso dei nodi λ_i con grado superiore a 2 e accrescere il peso dei nodi con grado inferiore a 2. Una semplice formuletta applicabile potrebbe essere la seguente:

$$\lambda'_i = \lambda_i + 2 - deg(i) \text{ (grado di i nell'1-tree minimo)}$$

4 1-Tree

Come precedentemente accennato una possibile e valida assegnazione dei moltiplicatori lagrangiani risulta quella di impostarli tutti a 0, andando di fatto ad eliminare i vincoli per ricondurci al problema di trovare l'1-tree di costo minimo.
Dato un grafo $G = (V, E)$ non orientato ed un suo nodo n chiamiamo **1-tree** un sottografo $H = (V, E_H)$ di G con $E_H \subset E$ e con le seguenti proprietà:

1. in E_H ci sono esattamente 2 archi incidenti sul nodo n;
2. se escludiamo da H il nodo n ed i suoi 2 archi incidenti su di esso ne risulta un albero sull'insieme di nodi $V \setminus \{n\}$. Alternativamente si può affermare che H contiene un circuito passante per il nodo selezionato n.

Da questa definizione segue che $|E_H| = |V|$.
L'aspetto importante degli 1-tree è che ogni circuito hamiltoniano risulta un 1-tree, non è invece vero il viceversa. Se indichiamo con S' l'insieme di tutti gli 1-tree calcolabili dato un grafo G e con S la regione ammissibile del TSP, abbiamo che $S \subset S'$. In altre parole il problema

$$\min_{H\in S'} \sum_{(i,j)\in E_H} c_{ij}$$

risulta essere un rilassamento per il problema del TSP simmetrico e la sua risoluzione restituisce quindi un lower bound per il valore ottimo del problema del TSP.
La formulazione matematica dell'1-tree è la seguente:

$$\begin{aligned}
&\min z = \sum_{(i,j)\in E} c_{ij} \cdot x_{ij} \\
&(5) \quad \sum_{j\in V, j\neq n} x_{nj} = 2 \\
&(2) \quad \sum_{(i,j)\in E, i,j\neq n} x_{ij} = |V| - 2 \\
&(3) \quad \sum_{(i,j)\in E(U)} x_{ij} \leq |U| - 1 \quad \forall\, U \subseteq V \setminus \{n\} : |U| \geq 3 \\
&(4) \quad x_{ij} \in \{0, 1\} \quad \forall (i,j) \in E
\end{aligned}$$

Come si può notare questa appare identica al rilassamento lagrangiano prima mostrato, una volta impostati i vari $\lambda_i = 0$.

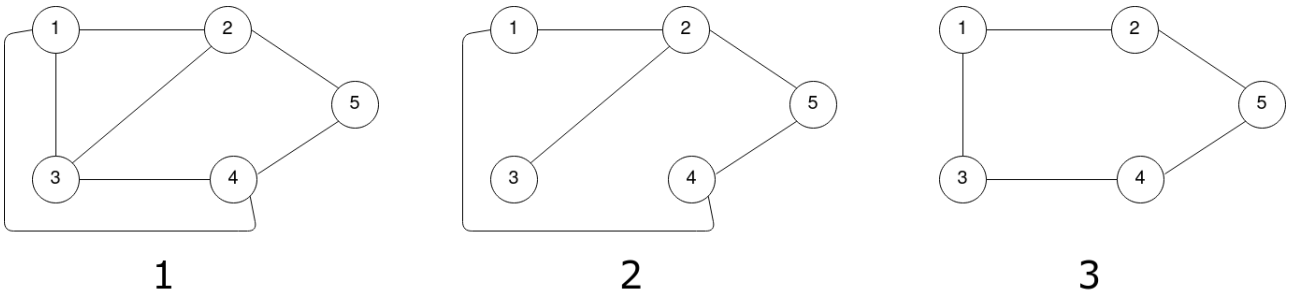


Figura 1 – Nell'immagine 1 è proposta un'istanza di grafo $G(V, E)$ non orientato. I grafi 2 e 3 rappresentano invece due suoi 1-tree di cui il 3 è anche un circuito hamiltoniano.

Mostreremo ora la procedura per risolvere in tempo polinomiale il problema del minimo 1-tree.

5 Calcolo del Lower Bound e Upper Bound

Una semplice procedura per calcolare un 1-tree di costo minimo e generare quindi un **lower bound** al problema del TSP segue i successivi passi:

1. Si calcoli l'MST T sul grafo ottenuto da G scartando il nodo prescelto n e tutti gli archi incidenti su di esso. Sia E_T l'insieme degli archi della soluzione trovata;
2. Si aggiungano ad E_T i due archi (n, k) e (n, h) a distanza minima tra quelli incidenti sul nodo n.
3. Si restituisca l'1-tree $H = (V, E_H)$ con $E_H = E_T \cup \{(n, k), (n, h)\}$.

Il costo della procedura appena proposta risulta dominato dal calcolo dell'MST, risolvibile facilmente con un algoritmo greedy come quello di Kruskal in tempo $O(m \cdot \log n)$. La selezione al passo 2 della coppia degli archi di costo minore è ottenibile invece con una semplice scansione degli archi $O(m)$.
Al costo di un maggiore sforzo computazionale si possono anche calcolare tutti gli 1-tree minimi variando la scelta del nodo n tra tutti i $|V|$ nodi del grafo. Come lower bound complessivo si potrà in questo modo scegliere il migliore ottenuto, ovvero il più grande trovato.
Per quanto riguarda il calcolo e l'aggiornamento dell'**upper bound**, ricordiamo che questo si basa sull'identificazione di soluzioni ammissibili durante l'esecuzione dell'algoritmo. Una volta calcolato un 1-tree se questo risulta anche un circuito hamiltoniano, ossia ogni nodo presenta grado 2, si può aggiornare l'upper bound se presenta un costo minore di quello per ora trovato. All'inizio del procedimento l'upper bound sarà impostato a $+\infty$.

6 Schema di Branch

Ci occuperemo ora di mostrare come sarà partizionata la regione ammissibile S in più sottoinsiemi. Se non siamo nel caso fortunato in cui la soluzione del rilassamento è un circuito hamiltoniano, tale soluzione sarà allora un 1-tree che contiene esattamente un sottocircuito. Dobbiamo introdurre una regola di suddivisione il cui scopo è impedire il formarsi nei nodi figli di tale sottocircuito. Indichiamo con: $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ gli archi che compongo tale sottocircuito. Il primo nodo 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 nodo figlio sarà ottenuto imponendo che sia presente l'arco (i_1, j_1), ma sia assente l'arco (i_2, j_2), ovvero impostando $x_{i_1 j_1} = 1$ e $x_{i_2 j_2} = 0$. Il procedimento continua in questo modo fino all' r-esimo figlio che avrà tutti i primi $r-1$ archi del sottocircuito e non conterrà l'ultimo, quindi $\forall k = 1, 2, \ldots, r-1 \;\; x_{i_k j_k} = 1$ e $x_{i_r j_r} = 0$.
Ad ogni nodo dell'albero di branch saranno quindi associati due insiemi:

1. E_0 contente tutti gli archi che non devono essere considerati;
2. E_1 contenente tutti gli archi che devono essere in soluzione.

Naturalmente sarà anche necessario che $E_0 \cap E_1 = \emptyset$.
Per ogni figlio si dovrà quindi risolvere un sottoproblema del tipo $S(E_0, E_1)$ contenente tutti i circuiti hamiltoniani formati sicuramente dagli archi in E_1 e privi degli archi in E_0. Per il calcolo del lower bound di un sotto problema $S(E_0, E_1)$ si usa la stessa procedura analizzata nel paragrafo precedente imponendo però la presenza degli archi E_1 ed escludendo quelli in E_0 per il calcolo dell'MST e nella scelta dei 2 archi per il nodo n. In particolare si risolverà sempre il problema dell'MST con l'algoritmo greedy di Kruskal, ma inizializzando l'insieme E_T con gli archi in E_1 non incidenti sul nodo n, invece che impostarlo come insieme vuoto. Una volta fatto ciò durante l'esecuzione dell'algoritmo non dovranno essere presi in considerazione gli archi in E_0. Ottenuto l'albero di copertura T a questi saranno aggiunti i migliori (con costo più basso) archi incidenti in n. Nel caso in cui E_1 contenesse tali archi saranno selezionati altrimenti si sceglieranno i migliori non presenti in E_0.
Per il branching di un nodo interno, al sottocircuito $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ individuato andranno tolti gli archi nell'insieme E_1 associato al nodo stesso, che non possono non essere presenti nei figli che saranno generati. Una volta scremato l'insieme di archi si procederà alla stessa maniera e verranno estesi gli insieme E_0, E_1 del padre per generare i vari figli.
Con questa regola di branching i nodi figli di un dato nodo continueranno ad essere sottoinsiemi della regione ammissibile del padre.

7 Criteri di chiusura dei nodi

Un nodo dell'albero di branch P_i viene chiuso quando:

1. $\hat{z} \leq LB(P_i)$, ossia quando il lower bound fornito è maggiore del migliore circuito hamiltoniano per ora trovato, in tal caso il nodo viene **chiuso per bound**;
2. non si riesce a calcolare un 1-tree per il nodo: in questo caso non esiste soluzione ammissibile per il rilassamento e il nodo sarà **chiuso per inammissibilità**;
3. l'1-tree generato dal rilassamento risulta essere un circuito hamiltoniano: il nodo verrà allora **chiuso perchè possibile candidato ottimo**.

Nell'ultimo caso delineato qualora il nodo presentasse anche un costo migliore e quindi minore dell'attuale soluzione trovata, questa verrebbe aggiornata.
Se alcuni nodi risultano aperti la scelta del prossimo nodo su cui fare Branch ricade su quello che presenta un Lower Bound minore, ossia quello che ci permette, in caso di ottimalità, di chiudere prima eventuali nodi aperti e terminare l'esecuzione. Un tale approccio viene detto **Best First**.

8 Analisi dell'implementazione

L'intero codice implementativo può essere visionato direttamente presso la pagina GitHub della tesina. Seguirà in questo paragrafo un'analisi delle scelte rilevanti compiute e uno sguardo sull'esecuzione degli algoritmi discussi su istanze medio/grandi di grafi.

Per quanto riguarda la struttura dati utilizzata abbiamo implementato il grafo e quindi anche l'1-tree che di volta in volta calcoliamo con delle **liste di adiacenze** realizzate con HashMap. Nello specifico il grafo è visto come un'insieme di nodi, che a loro volta contengono insiemi di archi. Entrambi questi insiemi sono stati realizzati con HashMap per rendere più efficiente il recupero dell'informazione. Questa scelta risulta la più conveniente dal punto di vista della complessità per i compiti che dobbiamo svolgere. Un'esplorazione completa del grafo, così come lo spazio necessario per la memorizzazione richiede complessità $O(n+m)$, mentre la scansione degli adiacenti di un nodo risulta di costo minimo $O(|A(u)|)$, dove $A(u) = \{v \mid (u,v) \in E\}$. Rispetto alla scelta di usare delle matrici di incidenza, risulta svantaggiato il controllo dell'esistenza di un arco nel grafo che richiede tempo $O(1)$ con le matrici di adiacenza e $O(|A(u)|)$ con le liste di adiacenza. Dato che quest'operazione non viene da noi mai eseguita, mentre risultano centrali l'operazione di scansione degli adiacenti di un nodo e dell'intero grafo, le liste di adiacenza si dimostrano ottime. Questi compiti richiedono infatti rispettivamente complessità $O(n)$ e $O(n^2)$ con una matrice di adiacenza.

Tra le procedure più spesso eseguite troviamo sicuramente il calcolo del **Minimum Spanning Tree**, che viene ripetuto per ogni nodo dell'albero di Branch che andiamo a generare. Per tale calcolo abbiamo implementato l'algoritmo greedy proposto da Kruskal che presenta complessità ottima $O(m \cdot \log n)$. Tale costo risiede principalmente nell'ordinamento decrescente iniziale degli archi, e nell'uso di Merge Find Set per controllare che un i-esimo arco possa essere incluso in soluzione con la certezza che non formi un ciclo. Ricordiamo che il Merge find set rappresenta una partizione di un insieme finito in sottoinsiemi disgiunti dette componenti o parti. Le operazioni ammesse permettono di controllare a quale componente appartiene un certo elemento e di unire due componenti diverse. Entrambe le operazioni presentano complessità $O(\log n)$, dove n rappresenta la cardinalità dell'insieme analizzato.

Per quanto riguarda l'individuazione del sottocircuito all'interno di un 1-tree che non è un circuito hamiltoniano abbiamo usato una **Depht First search** con complessità $O(n+m)$. L'idea è usare un vettore di padri per evitare di visitare più volte i nodi del grafo e per tener traccia del padre di ogni nodo. Conclusa l'esplorazione in profondità del grafo, semplicemente partendo dal nodo candidato n e procedendo con i vettori dei padri a ritroso, si individua il ciclo che ci servirà per il fare il Branch di un problema P_i.

La procedura centrale per la risoluzione del TSP è firmata `solveProblem()` e restituisce un'istanza della classe `TSPResult`, contenente il circuito hamiltoniano con costo minore e una serie di statistiche riguardo ai nodi analizzati nell'albero di Branch. Nel caso in cui non vi fosse un circuito hamiltoniano nel grafo di partenza, viene restituito il grafo originale. Questo metodo richiama gli algoritmi sopra descritti e molte altre procedure di supporto al fine di gestire una `PriorityQueue` di `SubProblem` e terminando l'esecuzione non appena quest'ultima viene completamente svuotata in seguito alla chiusura di tutti i nodi per una delle ragioni precedentemente indicate. É qui che risiede tutta la complessità della risoluzione del TSP simmetrico. Questo metodo infatti richiama sottoprocedure polinomiali in tempo, ma il numero di `SubProblem` che deve gestire può essere esponenziale.
La gestione dell'insieme dei SubProblem ancora aperti è stata realizzata con una coda di priorità

min-heap, ossia con una struttura dati che presenta complessità $O(1)$ per leggere l'elemento con lower bound minore e $O(\log\ n)$ per estrarre ed aggiungere un elemento.

Terminiamo questo paragrafo con l'analisi delle prestazioni dell'implementazione. Su grafi banali come quello che verrà mostrato nell'esempio conclusivo della tesina il codice termina in pochi millisecondi generando una manciata di nodi nell'albero di branch.
Come abbiamo potuto toccare con mano durante i nostri test, il tempo necessario per l'esecuzione dipende fortemente dal numero di archi che popolano il grafo. Esecuzioni su grafi sparsi di 100 nodi richiedono decine di secondi, mentre esecuzioni su grafi completi anche solo di 20 nodi richiedono potenzialmente diversi minuti. Per ottenere un'analisi significativa delle prestazioni, abbiamo creato una classe Java BasicCompleteGraphGenerator che, come si evince dal nome stesso, ci permette di generare grafi completi una volta specificato il numero di nodi e il range entro il quale devono risiedere i costi degli archi. Ricordiamo che un grafo completo G con n nodi avrà $\frac{n\cdot(n-1)}{2}$ archi non orientati.
Per dare maggiore rilevanza ai test effettuati abbiamo anche dato la possibilità di parallelizzare i calcoli su più thread. Una volta infatti che si è fatto il branch di un nodo e si ha generato tutti i figli di uno stesso livello questi possono tranquillamente essere computati simultaneamente per poi decidere se vadano chiusi per un qualche motivo o espansi con una ulteriore operazione di branch.
Mostriamo ora i dati che abbiamo raccolto. L'esecuzione è avvenuta su un calcolatore con le seguenti caratteristiche:

- Processore AMD Ryzen 7 5800X (4.6GHz)
- 32GB di RAM DDR4 (3200MHz)

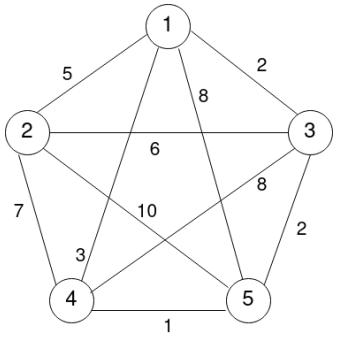
Thread	$V \in G$	$E \in G$	# sottoproblemi	Tempo medio d'esecuzione
1	5	10	9	<1ms
2	5	10	10	<1ms
4	5	10	10	<1ms
8	5	10	10	<1ms
1	10	45	3043	72ms
2	10	45	3043	66ms
4	10	45	3044	28ms
8	10	45	3044	16ms
1	13	105	130192	4.2s
2	13	105	130192	4.1s
4	13	105	130186	1.8s
8	13	105	130164	1s
1	15	105	811302	33.3s
2	15	105	811302	33.2s
4	15	105	811306	18.6s
8	15	105	811306	9.0s
1	20	190	>2.5M	...
2	20	190	>2.5M	...
4	20	190	>2.5M	...
8	20	190	>2.5M	...

Sia il tempo che il numero di sottoproblemi generati sono stati calcolati facendo una media dei risultati ottenuti su 500 esecuzioni per ogni istanza di grafo completo e per ogni numero di thread. Il campione usato è stato ridotto a 30 solamente nell'ultima serie di test a causa del tempo richiesto sempre maggiore. I puntini nella tabella rappresentano esecuzioni che non sono terminate a causa della mancanza di RAM per poter ospitare tutti i sottoproblemi ancora da valutare. Il Garbage Collector di Java si occupa di eliminare tutte le istanze di classi non più referenziate, ma il numero di SubProblem che sono ancora attivi nella frontiera dell'albero di branch può essere esponenzialmente grande. Basti pensare al fatto che non tutti i SubProblem rimangono aperti e generano figli, ma coloro che lo fanno possono avere un branching factor molto elevato.
Ovviamente tutto questo non ci sorprende più di tanto poichè rispecchia quella che è una procedura di risoluzione esponenziale per un problema NP-hard come il TSP.

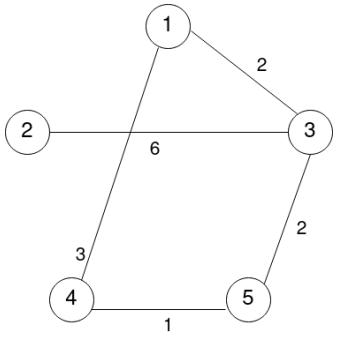
9 Esempio

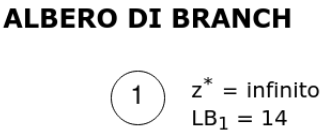
Risolveremo in quest'ultimo paragrafo della tesina un esercizio sul TSP simmetrico preso da una vecchia prova d'esame del corso di Ottimizzazione Combinatoria datata 10/06/2015.
L'istanza di grafo $G(V, E)$ è la seguente:

$$c_{ij}: \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} - & & & & \\ 5 & - & & & \\ 2 & 6 & - & & \\ 3 & 7 & 8 & - & \\ 8 & 10 & 2 & 1 & - \end{pmatrix} \end{array}$$



Come nodo candidato scegliamo arbitrariamente il nodo 1 e calcoliamo l'1-tree generando prima l'MST sui $V \setminus \{1\}$ vertici e poi aggiungendo i due archi ad esso incidenti di peso minimo. L'1-tree che otteniamo è il seguente:





Ci troviamo con un primo 1-tree che non è un circuito hamiltoniano e sul quale dobbiamo fare branch. Il sottocircuito che esso contiene è formato dai seguenti archi $\{(1,3),(3,5),(5,4),(4,1)\}$. Genereremo 4 figli così caratterizzati:

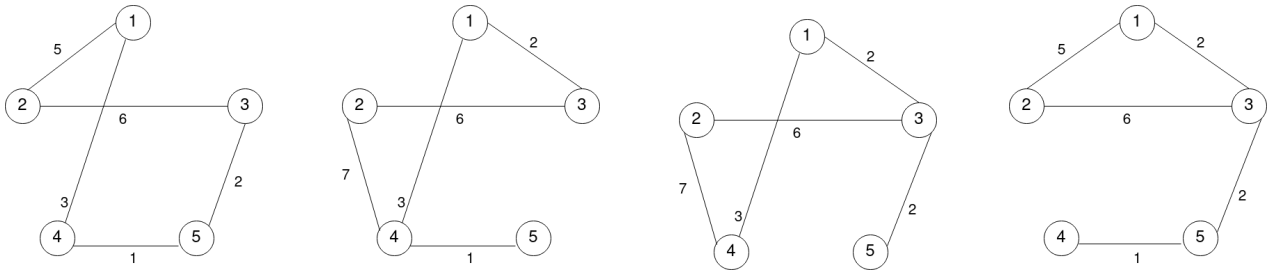
Nodo 2: $E_0 = \{(1,3)\}$ e $E_1 = \emptyset$;

Nodo 3: $E_0 = \{(3,5)\}$ e $E_1 = \{(1,3)\}$;

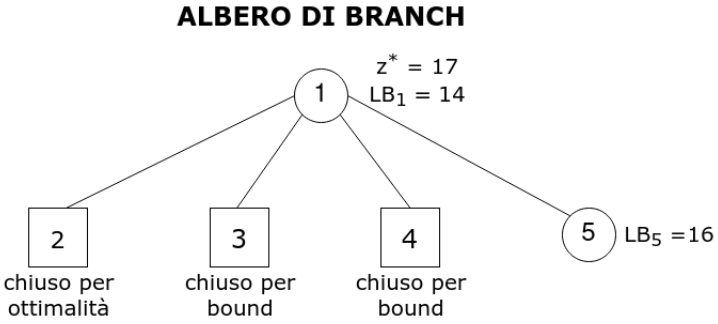
Nodo 4: $E_0 = \{(5,4)\}$ e $E_1 = \{(1,3),(3,5)\}$;

Nodo 5: $E_0 = \{(4,1)\}$ e $E_1 = \{(1,3),(3,5),(5,4)\}$.

Calcolando per ognuno dei nuovi nodi il corrispondente 1-tree abbiamo rispettivamente:



Il nodo 2 viene chiuso per ottimalità in quanto presenta un circuito hamiltoniano di costo 17, che diventa la nostra prima soluzione. Il nodo 3 viene chiuso per Bound in quanto presenta un lower bound pari a 19, maggiore della nostra soluzione. Discorso analogo viene fatto per il nodo 4 con lower bound 20. Il nodo 5 rimane invece aperto dato che presenta lower bound 16 e sul quale faremo branch.
L'albero di Branch aggiornato con queste nuove considerazioni risulta essere:

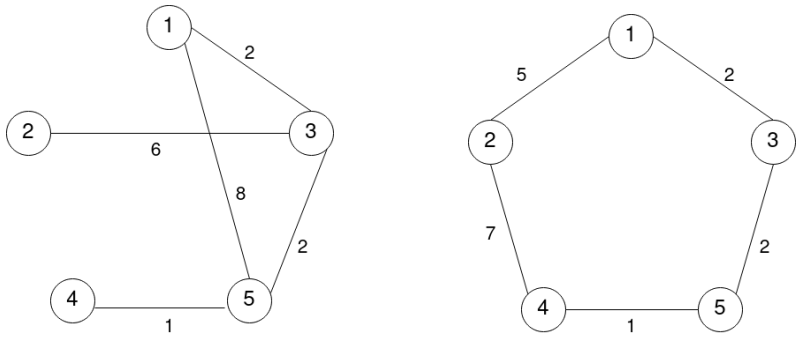


Il sottocircuito che il nodo 5 presenta è formato dagli archi $\{(1,2),(2,3),(3,1)\}$, scremato dagli archi nel suo E_1 otteniamo $\{(1,2),(2,3)\}$. I 2 figli che generiamo sono così caratterizzati:

Nodo 6: $E_0 = \{(4,1),(1,2)\}$ e $E_1 = \{(1,3),(3,5),(5,4)\}$;

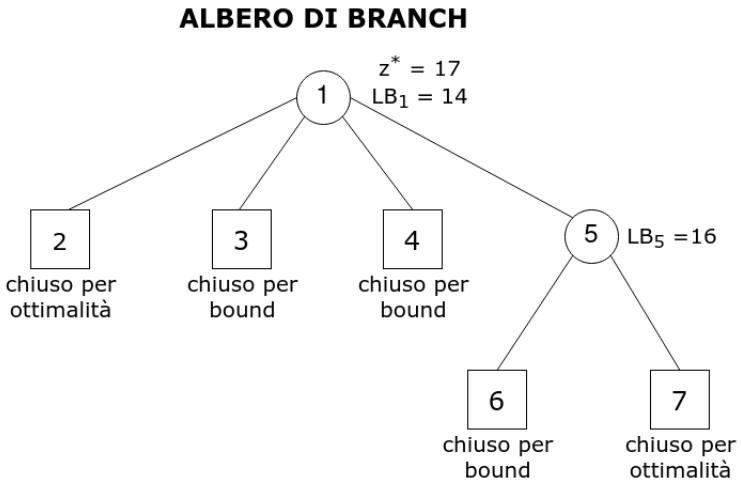
Nodo 7: $E_0 = \{(4,1),(2,3)\}$ e $E_1 = \{(1,3),(3,5),(5,4),(1,2)\}$.

Calcolando per ognuno dei nuovi nodi il corrispondente 1-tree abbiamo rispettivamente:



Il nodo 6 viene chiuso per bound, dato che ci garantisce un lower bound pari a 19, contro il costo 17 del nostro ottimo attuale. Il nodo 7 viene invece chiuso per ottimalità dato che presenta un circuito hamiltoniano con costo 17, uguale al nostro ottimo attuale che quindi non viene aggiornato.

L'albero di Branch finale risulta:



Il TSP simmetrico presenta come ottimo due possibili circuiti hamiltoniani di ugual costo: $\{(1,2),(2,3),(3,5),(5,4),(4,1)\}$ e $\{(1,2),(2,4),(4,5),(5,3),(3,1)\}$.