Branch & Bound per TSP simmetrico

Lorenzo Sciandra, Stefano Vittorio Porta

A.A. 20202021

Università degli Studi di Torino

Indice argomenti

Indice argomenti

Introduzione al TSP simmetrico

TSP

Il problema del commesso viaggiatore, spesso indicato come **Travelling Salesman Problem** nella sua più tipica rappresentazione chiede:
"*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.*"

Il TSP è un problema NP-hard, nella precisione NP-completo e quindi gli algoritmi che lo risolvono all'ottimo richiedono una complessità in tempo più che polinomiale nell'istanza trattata ($P \neq NP$).

TSP simmetrico

Per modellare il TSP simmetrico si usa 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.

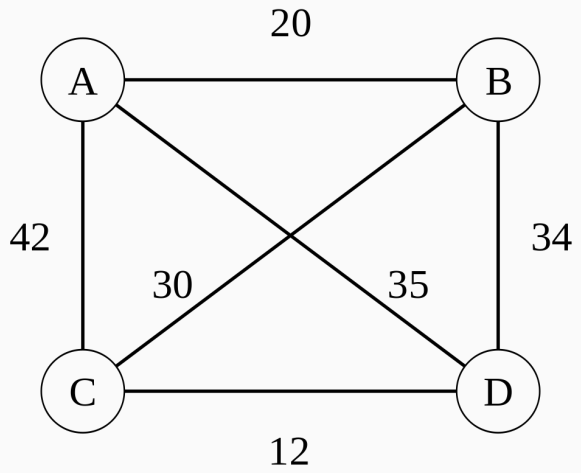


Figure 1: TSP simmetrico con 4 città

Formulazione matematica del TSP

Formulazione matematica del TSP

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:

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

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

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

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

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

Formulazione matematica del TSP - Dettagli

In dettaglio, il vincolo (1)

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

impone che su ogni nodo $i \in V$ incidano esattamente due archi.

I vincoli (2) e (3) invece

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

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

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.

Formulazione matematica del TSP - Dettagli

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

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). Imponiamo $|U| \geq 3$, poiché 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) infine

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

modella il dominio delle variabili decisionali che assumono valore 1 se il corrispondente arco viene inserito nel circuito hamiltoniano e 0 altrimenti.

Formulazione matematica del TSP - Idea di rilassamento

L'approccio che abbiamo scelto prevede un rilassamento Lagrangiano sul vincolo (1).
In questo modo il problema da risolvere risultante richiede di individuare 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.

Rilassamento Lagrangiano

Rilassamento Lagrangiano

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

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

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

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

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

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

Rilassamento Lagrangiano

Per valori fissati dei vari λ_k il rilassamento lagrangiano è facilmente risolvibile con una procedura (mostrata a breve) che consente di individuare l'1-tree di costo minimo.

Riscrivendo la funzione obiettivo otteniamo:

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

I costi associati agli archi risultano quindi aggiornati secondo:

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

Questo dà luogo ad un possible approccio risolutivo basato sul duale lagrangiano, che prevede di partire da una combinazione ammissibile di moltiplicatori come $\lambda_k = 0 \ \forall k$, migliorando successivamente la soluzione provando nuovi moltiplicatori.

1-Tree

1-Tree

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 sono presenti esattamente 2 archi incidenti sul nodo n;

1-Tree

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 sono presenti 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.

1-Tree

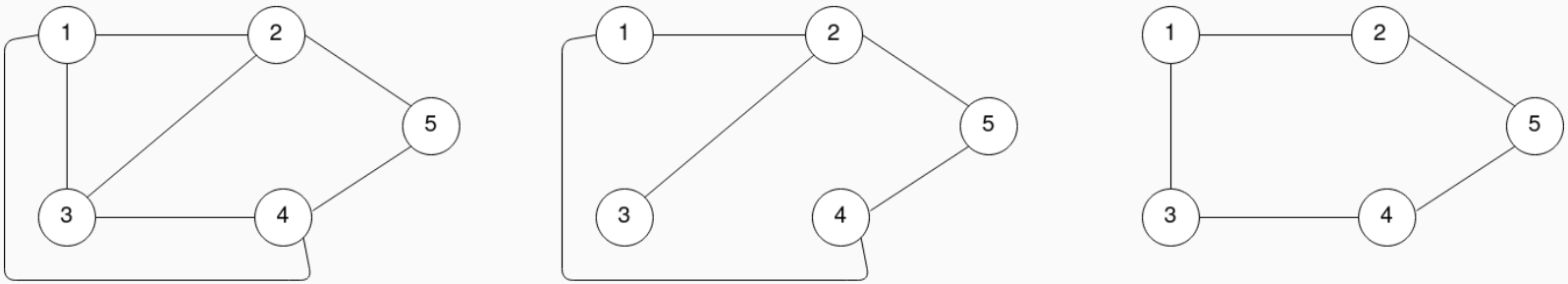
Dato un grafo $G = (V, E)$ non orientato ed un suo nodo n chiamiamo **1-tree** un sottografo $H = (V, E_H)$ di G con $E_H \subset E$ e con le seguenti proprietà:

1. in E_H sono presenti 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 $|E_H| = |V|$.

1-Tree

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 $S \subset S'$.



1 2 3

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

1-Tree

La formulazione matematica dell'1-tree risulta essere la seguente:

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

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

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

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

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

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

1-Tree

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;

1-Tree

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.

1-Tree

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 $E_H = E_T \cup \{(n, k), (n, h)\}$.

1-Tree

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 $O(m)$.

Per quanto riguarda il calcolo e l'aggiornamento dell'**upper bound**, 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 $+\infty$.

Schema di Branch

Schema di Branch

Ci occuperemo ora di mostrare come sarà partizionata la regione ammissibile S in più sottoinsiemi.

Se non stiamo analizzando il 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.

Schema di Branch

Indichiamo con $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ gli archi che compongo tale sottocircuito.

Schema di Branch:

1. Il primo nodo figlio verrà generato imponendo che l'arco (i_1, j_1) non faccia parte della soluzione, ossia impostando $x_{i_1 j_1} = 0$;

Schema di Branch

Indichiamo con $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ gli archi che compongo tale sottocircuito.

Schema di Branch:

1. Il primo nodo figlio verrà generato imponendo che l'arco (i_1, j_1) non faccia parte della soluzione, ossia impostando $x_{i_1 j_1} = 0$;
2. 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 $x_{i_2 j_2} = 0$;

Schema di Branch

Indichiamo con $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ gli archi che compongo tale sottocircuito.

Schema di Branch:

1. Il primo nodo figlio verrà generato imponendo che l'arco (i_1, j_1) non faccia parte della soluzione, ossia impostando $x_{i_1 j_1} = 0$;
2. 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 $x_{i_2 j_2} = 0$;
3. 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 $x_{i_r j_r} = 0$.

Schema di Branch

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.

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 precedentemente 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.

Schema di Branch - 1

In particolare si risolverà sempre l'MST con 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, verranno aggiunti gli archi incidenti in n meno costosi.
Nel caso in cui E_1 dovesse contenere tali archi, essi saranno selezionati.
In caso contrario si sceglieranno i migliori non presenti in E_0.

Schema di Branch - 2

Per il branching di un nodo interno, al sottocircuito individuato $\{(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)\}$ 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 nella stessa maniera e verranno estesi gli insiemi E_0 ed E_1 del padre per generare i vari figli necessari per proseguire la ricerca.

Chiusura dei nodi

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**;

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**;

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**.

Chiusura dei nodi

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 il Lower Bound minore non ancora visitato.
Il nodo scelto in altri termini è 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**.

Analisi dell'implementazione

Analisi dell'implementazione

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

Analisi dell'implementazione - Grafo

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.

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 $A(u) = \{v \mid (u, v) \in E\}$.

Analisi dell'implementazione - MST

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.

Analisi dell'implementazione - DFS

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.

Analisi dell'implementazione

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.

Analisi dell'implementazione

Il metodo solveProblem() richiama gli algoritmi precedentemente descritti assieme a 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.

Analisi dell'implementazione

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.

Analisi dell'implementazione - Prestazioni

Terminiamo la relazione con l'analisi delle prestazioni della nostra implementazione.

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 manciate di minuti.

Analisi dell'implementazione - Prestazioni

Per ottenere un'analisi significativa delle prestazioni, abbiamo creato una classe Java BasicCompleteGraphGenerator che 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.

Analisi dell'implementazione - Prestazioni

Mostriamo ora i dati che abbiamo raccolto con un calcolatore con processore AMD Ryzen 7 4.6GHz e 32 GB di RAM DDR4:

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	...

Analisi dell'implementazione - Prestazioni

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 esponenzialmente sempre maggiore.

Analisi dell'implementazione - Prestazioni

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.