Ottimizzazione spaziale dell'algoritmo di Held – Karp per la risoluzione del TSP e comparazione con algoritmi di approssimazione

Michele Maione[1]

1 – Abstract

Ho opportunamente modificato l'algoritmo di programmazione dinamica di Held – Karp[2] per risparmiare memoria. Tramite l'uso di alberi rosso – neri, e generando i sotto-percorsi in un determinato ordine, sono riuscito a liberare memoria durante l'esecuzione risparmiando il 15%. Il risparmio di memoria si traduce in una maggiore velocità di esecuzione fisica (per via del paging) e in un aumento del massimo numero di nodi elaborabili.

Ho implementato poi gli algoritmi di approssimazione di Held – Karp con rilassamento lagrangiano[3], di Christofides[5], di Volgenant – Jonker[4] (solo la parte di rilassamento) e un algoritmo 2-approssimato per il TSP euclideo[1].

2 – Introduzione

Il problema del commesso viaggiatore (TSP, Traveling Salesman Problem) è trovare il percorso minimo passante per un insieme di città, tale che, si passi una ed una sola volta per la stessa città, e si ritorni alla città di partenza. Originariamente, il problema era trovare il tour più breve di tutte le capitali degli Stati Uniti.

Matematicamente può essere rappresentato come un grafo pesato. Se il grafo è diretto allora il TSP è definito asimmetrico. Il problema consiste nel trovare un ciclo hamiltoniano a costo minimo sul grafo, purtroppo è un problema NP-completo, e viene quindi, principalmente, calcolato tramite approssimazione.

Ad ogni arco del grafo $G=(V,E)$ è associato un peso c, e una variabile booleana x che indica se l'arco appartiene al percorso. La formulazione matematica del problema è:

$min \sum_{e \in E} c_e \cdot x_e$		Minimizzazione costo del percorso
$S.T.$		
$\sum_{e \in E} x_e = 2$	$(\forall\, e \in E)$	Ogni nodo ha grado 2
$\sum_{e \in E(S)} x_e \leq \lvert S \rvert - 1$	$(\forall\, S \subset \{2,\dots,n\})$	Assenza di sotto-percorsi
$x_e \in \{0,1\}$	$(\forall\, e \in E)$	Arco è presente o non presente

3 – Algoritmo D.P. di Held – Karp

L'algoritmo Held – Karp, è un algoritmo di programmazione dinamica proposto nel 1962 per risolvere il TSP. L'algoritmo si basa su una proprietà del TSP: ogni sotto-percorso di un percorso di minima distanza è esso stesso di minima distanza; quindi calcola le soluzioni di tutti i sotto-problemi partendo dal più piccolo. Purtroppo non possiamo sapere quali sotto-problemi dobbiamo risolvere, quindi li risolviamo tutti. L'algoritmo ha una complessità temporale $O(2^n \cdot n^2)$ e una complessità spaziale di $O(2^n \cdot n)$.

1 Michele Maione - 931468 - michele.maione@studenti.unimi.it, Ottimizzazione Combinatoria, A/A 2019-2020, Università degli Studi di Milano, via Celoria 18, Milano, Italia.

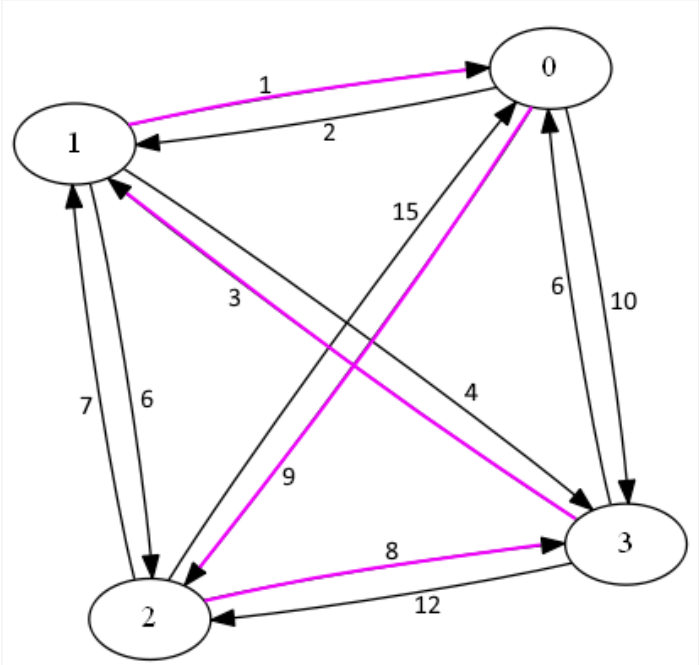

Figura 1: aTSP su 4 nodi

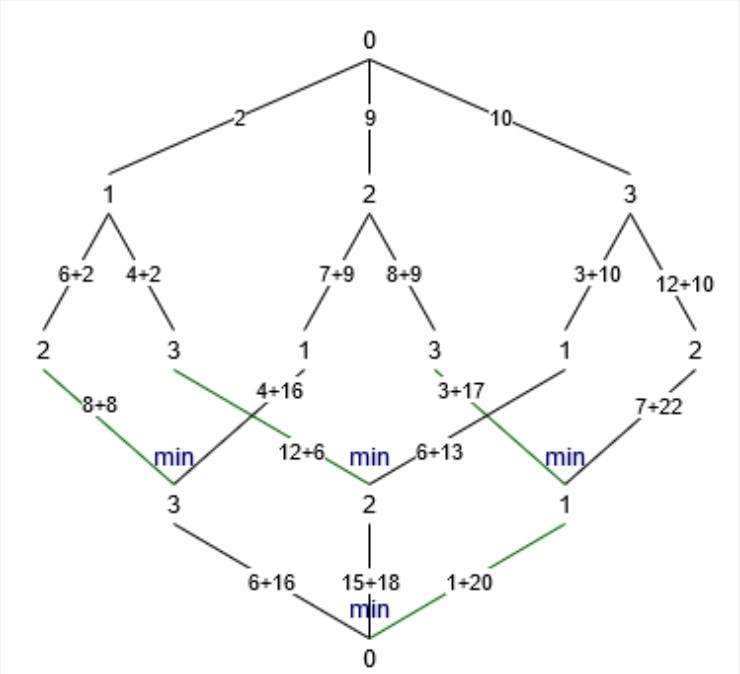

Figura 2: Esecuzione algoritmo di Held – Karp

3.1 – Ottimizzazione

Le ottimizzazioni che sono state fatte riguardano la liberazione della memoria, e le strutture dati utilizzate per memorizzare i dati.

3.1.1 – Liberazione memoria

Ci sono due ottimizzazioni spaziali che possono essere fatte durante l'esecuzione:

1. Alla fine dell'elaborazione di una cardinalità *s*, si possono eliminare gli elementi della cardinalità *s – 2* (rigo 34), portando l'algoritmo ad una complessità spaziale da $O(2^n \cdot n)$ a $O(2^n \cdot \sqrt{n})$;
2. Prima dell'elaborazione di un set, si possono eliminare gli elementi appartenenti alla cardinalità *s – 1*, che hanno il primo elemento minore dal primo elemento del set attuale (rigo 26), riducendo del 15% la memoria utilizzata.

In questa tabella di esempio, di 5 nodi, ho colorato i set che dipendono tra di loro. Resta il fatto che dipendono solo da quelli della cardinalità precedente.

Cardinalità	Set
1	{1}, {2}, {3}, {4}, {5}
2	{1,2}, {1,3}, {1,4}, {1,5}, {2,3}, {2,4}, {2,5}, {3,4}, {3,5}, {4,5}
3	{1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}, {1,4,5}, {2,3,4}, {2,3,5}, {2,4,5}, {3,4,5}
4	{1,2,3,4}, {1,2,3,5}, {1,2,4,5}, {1,3,4,5}, {2,3,4,5}
5	{1,2,3,4,5}

3.1.2 – Strutture dati

L'algoritmo ha bisogno di memorizzare una tupla formata da: set, nodo, costo, percorso. La soluzione più semplice sarebbe stata quella di utilizzare una matrice, ma sarebbe stata sparsa, per via della memorizzazione del set.

Ho scelto di codificare i set con questa funzione: $Hash(S):\mathbb{N}^n \to \mathbb{N} = \sum 2^i : \forall i \in S$. Questo mi ha permesso di utilizzare 32bit per memorizzare il set in modo univoco.

Ho utilizzato un albero rosso – nero[6], usando come chiave l'intero generato come sopra, e come valore il puntatore ad un altro albero rosso – nero. Quest'ultimo ha come chiave il numero del nodo, e come valore il puntatore ad una tupla che contiene costo e percorso. L'albero assicura le funzioni di inserimento, cancellazione e ricerca in $O(\log n)$.

Per ogni nuova cardinalità da elaborare, creo un nuovo albero, e lo inserisco in una coda di dimensione 2, poiché ogni elaborazione ha bisogno solo dell'elaborazione precedente, quindi al massimo ci sono 2 cardinalità in memoria.

3.2 – Pseudo-codice

```
01  info = record
02    cost: int
03    π: List<int>
04
05  MemoryTree: RBTree<int, RBTree<int, info>>
06  Q: Queue<MemoryTree>
07  C, P: MemoryTree
08
09  function Hash(Set<int> S): int
10    for i in S
11      hash = hash + 2^i
12    return code
13
14  function aTSP-Held-Karp
15    C = Q.enqueue(new MemoryTree())
16
17    for k = 1 to n - 1
18      C(Hash({k}), k) = d_0,k
19
20    for s = 2 to n - 1
21      P = C
22      C = Q.enqueue(new MemoryTree())
23
24      for S in {1, … , n - 1} and |S| = s
25        if i < S[0]
26          P.delete(Hash({S[0], … , n - 1}))
27
28        for k in S
29          C[S, k].cost = min_m≠k,m∈S(P[Hash(S\{k})][m].cost + d_m,k)
30          C[S, k].π = P[Hash(S\{k})][m].π + m
31
32        i = S[0]
33
34      Q.dequeue()
35
36    return min_k≠0(C[Hash({1, … , n - 1})][k] + d_k,0)
```

4 – Algoritmo di Christofides

L'algoritmo Christofides[5] è un algoritmo del 1976 per trovare soluzioni approssimative al problema del TSP euclideo. È un algoritmo di approssimazione che garantisce un fattore 3/2 sulla soluzione ottima. L'algoritmo esegue i seguenti passi:

1. T = albero ricoprente minimo del grafo G;
2. O = insieme dei nodi con grado dispari in T;
3. M = accoppiamento perfetto di peso minimo sul sotto-grafo indotto di G, usando i nodi O;
4. H = multi-grafo connesso formato dagli archi di M e T . Ogni vertice ha grado pari;
5. C = circuito euleriano su H;
6. Z = circuito hamiltoniano da C, saltando i vertici ripetuti.

Z sarà la soluzione del TSP.

4.1 – Implementazione

Gli algoritmi interni utilizzati sono:

- MinimumSpanningTree: uso l'algoritmo di Prim $O(E \cdot \log V)$, e creo l'out-star;
- OddDegreeVertex: creo un vettore di vertici di grado dispari $O(V)$;
- MinimumWeightedPerfectMatching: ho utilizzato un algoritmo goloso[1] per calcolare l'accoppiamento perfetto pesato, eseguito in $O(V^3)$[2];
- FindEulerCircuit: trovo un circuito euleriano $O(E)$;
- HamiltonianPath: lo trasformo in un circuito hamiltoniano $O(E)$.

4.2 – Pseudo-codice

```
01  Adj: List<int>[N]
02
03  function sTSP-Christofides
04    MinimumSpanningTree()
05    O = OddDegreeVertex()
06    WeightedPerfectMatching(O)
07    C = FindEulerCircuit(H)
08
09    return HamiltonianPath(C)
```

5 – Algoritmi con rilassamento lagrangiano

Gli algoritmi branch and bound con rilassamento lagrangiano di Held – Karp[3] e Volgenant – Jonker[4], risalgono rispettivamente al 1969 e al 1980. Si basano sul concetto che un ciclo hamiltoniano è un 1-albero in cui ogni vertice ha grado uguale a 2. Il costo di un ciclo hamiltoniano su un grafo connesso e non diretto è maggiore del costo minimo di 1-albero su quel grafo. Dato un tour ottimo H^* e un 1-albero T, si ha: $c(H^*) \geq min\{c(T)\}$. Il rilassamento lagrangiano tenta di migliorare il bound eliminando una parte dei vincoli dal problema originale, inserendoli nella funzione obiettivo. In questo caso si eliminano i vincoli di grado sui

2 Attualmente l'algorimo "blossom" di Jack Edmonds del 1961, è il miglior algoritmo per calcolare un abbinamento massimo in $O(V^2 \cdot E)$.

vertici $\{2,\dots,n\}$, inserendoli nella funzione obiettivo come somma pesata secondo dei moltiplicatori lagrangiani π_i . Come è possibile notare nella tabella successiva, tutti i vincoli del problema corrispondono al modello matematico di un 1-albero. Le soluzioni del problema lagrangiano sono tutti gli 1-alberi del grafo.

Il valore $L(\lambda)$ costituisce un lower bound. Quindi per ottenere un buon lower bound si può cercare di massimizzare $L(\lambda)$. Questa massimizzazione viene effettuata con il metodo del sub-gradiente.

$min\, L(\lambda)=min\sum_{e\in E} c_e\cdot x_e-\sum_{i=2}^{n}\lambda_i\cdot(\sum_{e\in\delta(v)} x_e-2)$		Funzione lagrangiana
$S.T.$		
$\sum_{e\in\delta(1)} x_e=2$		Il nodo 1 ha grado 2
$\sum_{e\in E(S)} x_e\le\|S\|-1$	$(\forall\, S\subset\{2,\dots,n\},\|S\|\ge 2)$	Assenza di sotto-percorsi
$\sum_{e\in E\setminus\delta(1)} x_e=n-2$		
$x_e\in\{0,1\}$	$(\forall\, e\in E)$	Arco è presente o non presente

5.1 – Metodo del sub-gradiente

Si generano iterativamente una successione di λ_i secondo la seguente regola: $\lambda_i^{k+1}=\lambda_i^{k+1}+t^k\cdot s_i^k$. Con s^k indichiamo un sub-gradiente di $L(\lambda^k)$ e con t^k l'ampiezza dello spostamento.

5.1.1 – Equazioni per sub-gradienti e ampiezze

Sia Held – Karp che Volgenant – Jonker propongono una serie di equazioni per il calcolo di t^k e s^k tra loro differenti che cambiano leggermente i valori restituiti ma non l'algoritmo. Nell'algoritmo di Held – Karp abbiamo $t^k=\frac{\pi^k[UB-L(\lambda^k)]}{\sum_{i\in V} d_i(x^k)-2}$, mentre nell'algoritmo di Volgenant – Jonker $t^k=t^1\,\frac{k^2-3(M-1)k+M(2M-3)}{2(M-1)(M-2)}$.

6 – Algoritmo 2-approssimato (Cormen)

L'altro algoritmo che ho codificato è un 2-approssimato che viene eseguito in `Θ(V²)`. L'algoritmo esegue i seguenti passi:

1. Seleziona un vertice r come radice;
2. Usa l'algoritmo di Prim per calcolare un albero di connessione minimo T;
3. Sia H una lista di vertice, ordinata in base a quando un vertice viene visitato per primo in un attraversamento anticipato di T;
4. Ritorna il ciclo hamiltoniano da H.

7 – Risultati ottenuti

7.1 – Algoritmo D.P. di Held – Karp

I risultati ottenuti sono stati migliori a livello di memoria utilizzata. Ci sono stati miglioramenti nei tempi d'esecuzione, solo nei casi di saturazione della memoria RAM che sopraggiunge dopo, e posticipa l'attivazione del paging su disco. Si è riuscito anche ad elaborare, con gli stessi computer, 25 nodi, mentre l'algoritmo non ottimizzato ne elaborava al massimo 21.

7.1.1 – Memoria allocata

La complessità spaziale ottenuta con le modifiche apportate è il 75% di quella originaria $S(n)=O(2^n \cdot \sqrt{n})$.

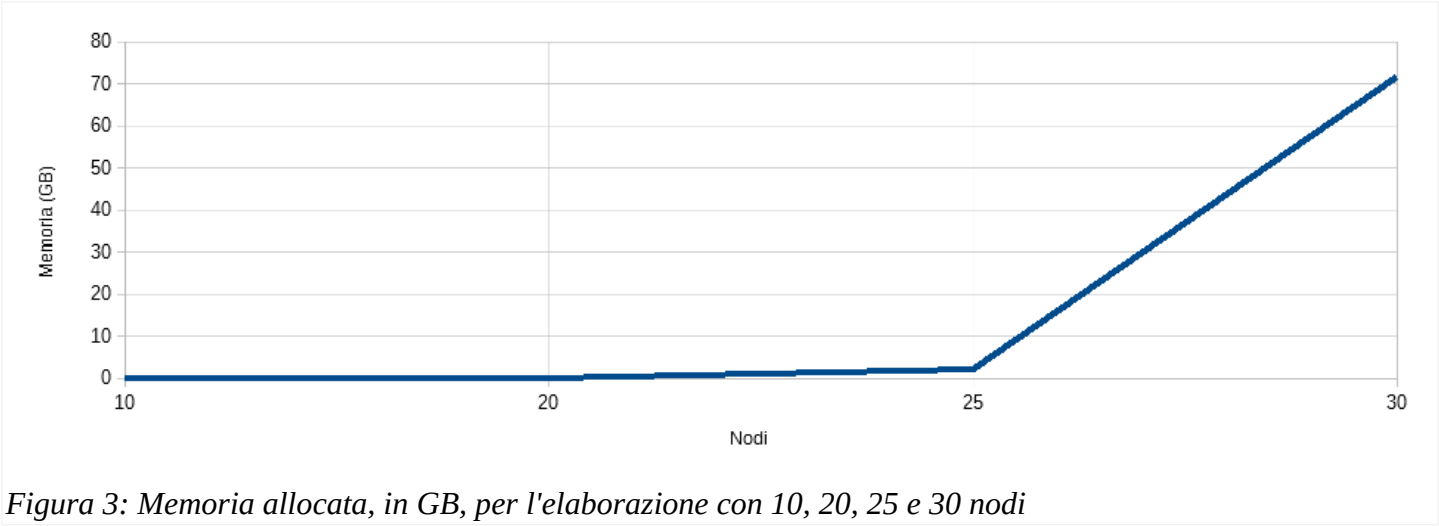

Figura 3: Memoria allocata, in GB, per l'elaborazione con 10, 20, 25 e 30 nodi

7.1.2 – Tempi d'esecuzione

L'algoritmo ha complessità temporale $T(n)=O(2^n \cdot n^2)$, che è esponenziale, ma il vero problema è la sua complessità spaziale $S(n)=O(2^n \cdot \sqrt{n})$, che obbliga il sistema operativo al paging e ad un rallentamento dell'elaborazione.

7.1.2.1 – Multithread

Ho implementato una versione multithread dell'algoritmo, che eseguiva le righe 28-30, in thread separati. Purtroppo il cambio di contesto e l'impossibilità di liberare la memoria basata sul contenuto dei set (liberazione memoria modalità 2), peggiorava i tempi d'esecuzione su tutti i problemi provati.

7.1.2.2 – Risultati

I risultati ottenuti, come accennato in precedenza, lo rendono non utilizzabile per problemi con più di 30 nodi.

Nodi	Tempo d'esecuzione
20	00'08"47
25	14'51"70

Nell'immagine seguente i tempi d'esecuzione su un processore Ryzen5. Si passa da 9 secondi per un grafo di 20 nodi a 15 minuti per un grafo di 25 nodi. Il problema risiede nel fatto che fino a 20 nodi l'algoritmo viene eseguito sulla RAM, mentre per grafi più grandi viene attivato il paging.

7.2 – Algoritmi di approssimazione

Ho testato su varie istanze da 4, 6, 10, 15 e 20 nodi, di grafi euclidei casuali, l’errore medio per l’algoritmo di Christofides è comunque sotto il fattore massimo 3/2.

Nodi	Errore medio		
	Christofides	Held – Karp	2-approssimato
4	1,00%	0,00%	1,00%
6	4,50%	0,00%	18,39%
10	1,67%	0,00%	15,74%
15	13,77%	0,00%	24,25%
20	4,84%	0,00%	21,88%

Nel seguente grafo da 6 nodi si può notare dove l’algoritmo di Christofides sceglie un tour peggiore.

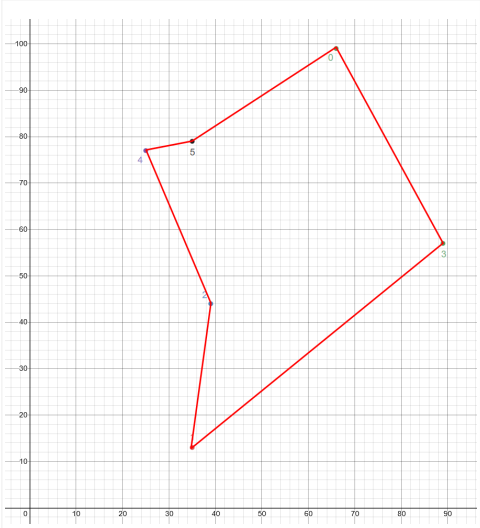

Figura 4: Held-Karp - 232

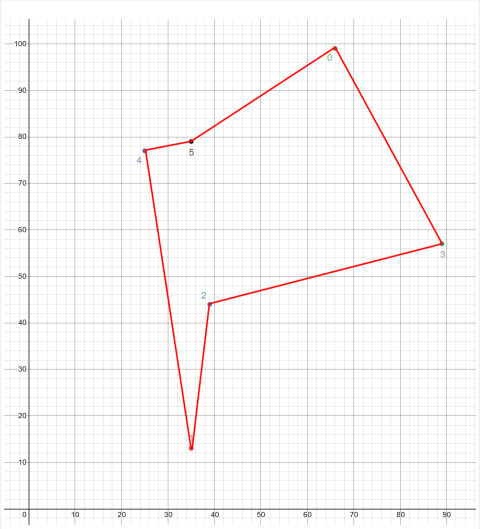

Figura 5: Christofides - 242

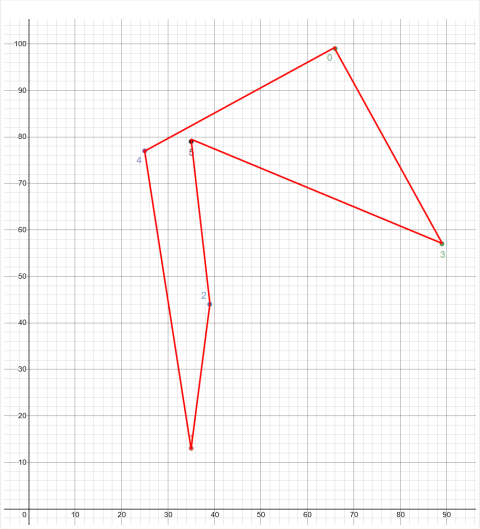

Figura 6: 2-approssimato - 284

7.2.1 – Tempi d’esecuzione

L’algoritmo di Christofides ha complessità temporale $T(n)=O(V^3)$ mentre il 2-approssimato $T(n)=O(V^2)$. Di seguito i risultati ottenuti:

Nodi	Tempo d’esecuzione		
	Christofides	Held – Karp	2-Approssimato
25	00'00"01	00'00"01	00'00"01
100	00'00"02	01'41"89	00'00"01
500	00'00"21		00'00"08
1000	00'00"92		00'00"35

8 – Commenti conclusivi

8.1 – D.P. Held – Karp

È stata scritta anche una versione, che utilizza hash-map salvate su disco, ma aumentava così tanto i tempi di esecuzione, che il calcolo di un grafo di 20 nodi diventava infattibile. Sono stati provati alcuni approcci multithread, rivelatisi non efficienti a livello di memoria, per via del cambio di contesto e per l'utilizzo dei mutex.

Ho appurato che la soluzione di programmazione dinamica non è fattibile sui moderni computer personali. L'unico modo per risolvere questo problema su grandi topologie sono gli algoritmi di approssimazione.

8.2 – Christofides

L'algoritmo di Christofides ha un margine d'errore di 3/2, ed è il migliore (qualità/tempo), tra quelli implementati, per risolvere il problema.

8.3 – Rilassamento lagrangiano e branch and bound

Questi algoritmi sono stati quelli che hanno dato i risultati migliori, ma sono molto più lenti rispetto all'algoritmo di Christofides.

8.4 – 2-Approssimato

L'algoritmo 2-approssimato è molto veloce e leggero, e può essere usato per calcolare un discreto upper-bound.

9 – Allegati

9.1 – Codice sorgente

Il codice del progetto è disponibile qui: https://github.com/mikymaione/Held-Karp-algorithm

10 – Riferimenti bibliografici

1. Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C., 2010. Introduzione agli algoritmi e strutture dati. McGraw-Hill.
2. Held, M. and Karp, R., 1962. A dynamic programming approach to sequencing problems. Journal for the Society for Industrial and Applied Mathematics, 1:10.
3. Held, M. and Karp, R., 1970. The Traveling Salesman Problem and Minimum Spanning Trees. Operations Research, 18, 1138-1162.
4. Volgenant, T. and Jonker, R., 1982. A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation. European Journal of Operational Research, 9(1):83–89.
5. Christofides, N., 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Report 388, Graduate School of Industrial Administration, CMU.
6. Lippman, S. and Lajoie, J. and Moo, B., 2012. C++ Primer. Addison-Wesley Professional.