# Ottimizzazione spaziale dell'algoritmo di Held – Karp per la risoluzione del TSP e comparazione con l'algoritmo di Christofides

Michele Maione[1]

## 1 – Abstract

Ho opportunamente modificato l'algoritmo di programmazione dinamica di Held – Karp 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 l'algoritmo euristico di Christofides per il TSP euclideo, per fare una comparazione con quello di Held – Karp.

## 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-Hard, e viene quindi, principalmente, calcolato tramite euristiche.

Ad ogni arco del grafo $G=(V,E)$ è associato un peso $c_{ij}$, e una variabile $x_{ij}=\{0,1\}$ che indica se l'arco appartiene al percorso. La formulazione matematica del problema è:

| | | |
|---|---|---|
| $\min \sum_{1\leq i<j\leq n}(c_{ij}\cdot x_{ij})$ | | Minimizzazione costo del percorso |
| S.T. | | |
| $\sum_{j>i}(x_{ij})+\sum_{j<i}(x_{ji})=2$ | $(i=1,\dots,n)$ | Ogni nodo ha grado entrante 1 e grado uscente 1 |
| $\sum_{i,j\in S;\ i<j}(x_{ij})\leq\lvert S\rvert-1$ | $(\forall S\subset\{2,\dots,n\})$ | Assenza di sotto-percorsi |
| $x_{ij}\in\{0,1\}$ | | Arco è presente o non presente |

## 3 – Algoritmo 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 Held – Karp con limite inferiore fornisce un limite inferiore per il costo del tour TSP ottimale di un grafo. Avere il limite inferiore per un particolare grafo è utile per verificare le prestazioni di un dato euristico.

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.

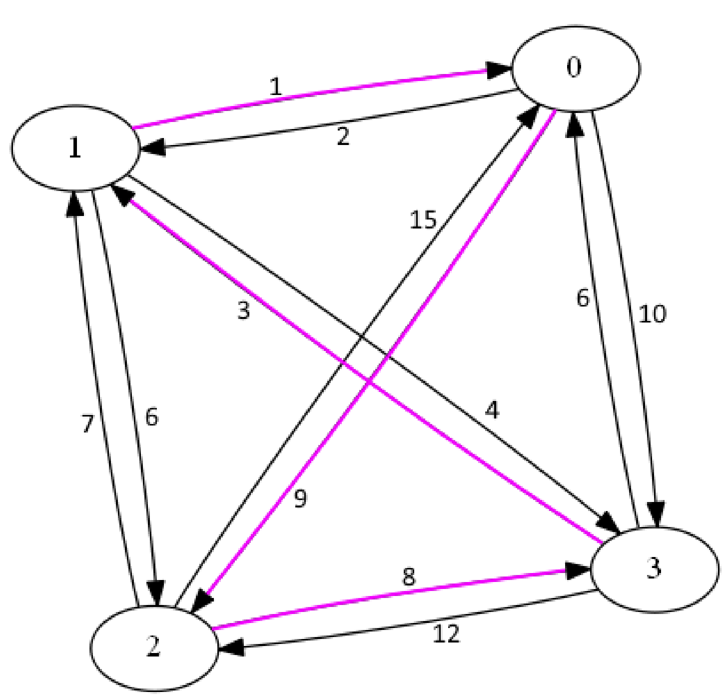


*Figura 1: aTSP su 4 nodi*

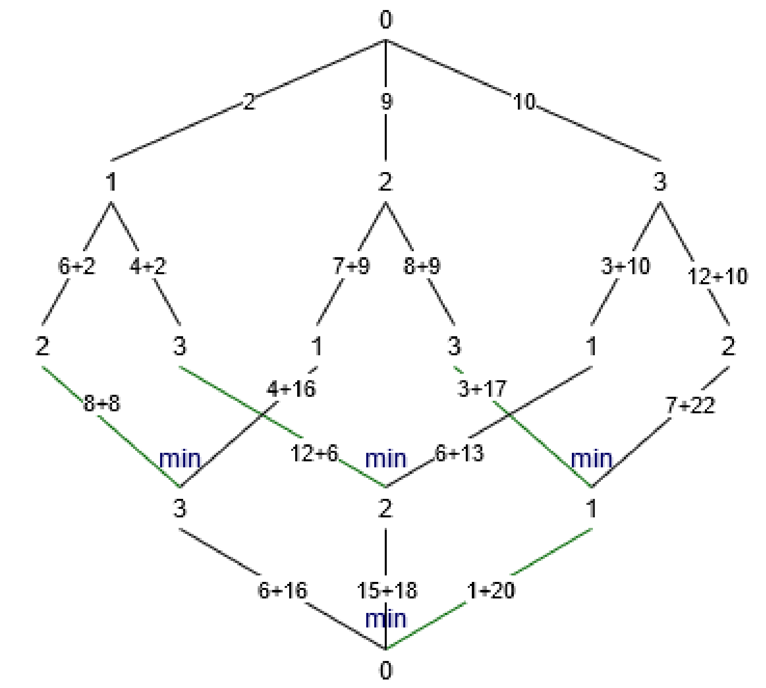


*Figura 2: esecuzione algoritmo di Held – Karp*

## 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 38);
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 28).

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: $Hash(S):\mathbb{N}^n \rightarrow \mathbb{N}=\Sigma 2^i : \forall i \in S$. Questo mi ha permesso di utilizzare 32bit per memorizzare il set in modo univoco.

Ho utilizzato un albero rosso – nero, 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.

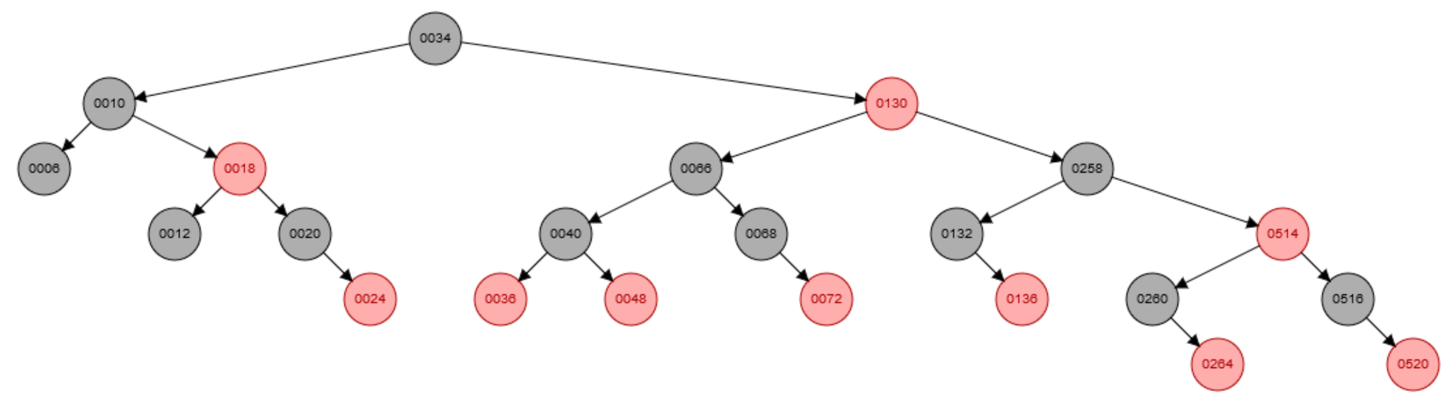


*Figura 3: Albero rosso – nero contenente le rappresentazioni in int32 dei sets*

## 3.2 – Pseudo-codice dell’algoritmo di Held – Karp

```
info = record
  cost: int
  π: List<int>

MemoryTree: RBTree<int, RBTree<int, info>>
Q: Queue<MemoryTree>
C, P: MemoryTree

function Hash(Set<int> S): int
  for i in S:
    hash = hash + 2^i
  return code

procedure TSP
  C = Q.enqueue(new MemoryTree())

  for k = 1 to n - 1:
    C(Hash({k}), k) = d_{0,k}

  for s = 2 to n - 1
    P = C
    C = Q.enqueue(new MemoryTree())

    for S in {1, … , n - 1} and |S| = s
      if i < S[0]:
        P.delete(Hash({S[0], … , n - 1}))

      for k in S
        C[S, k].cost = min_{m≠k,m∈S}(P[Hash(S\{k})][m].cost + d_{m,k})
        C[S, k].π = P[Hash(S\{k})][m].π + m

      i = S[0]

    Q.dequeue()

  return min_{k≠0}(C[Hash({1, … , n - 1})][k] + d_{k,0})
```

# 4 – Algoritmo di Christofides

L'algoritmo Christofides 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.

## 4.1 – Implementazione

Poiché gli algoritmi interni albero ricoprente minimo, accoppiamento perfetto di peso minimo, circuito euleriano e percorso hamiltoniano sono stati scritti, in questo progetto, solamente per essere usati in questo algoritmo, ho deciso di ottimizzarli, utilizzando una out-star in comune, così da non dover allocare e deallocare vari oggetti.

## 4.2 – Pseudo-codice dell’algoritmo di Christofides

```
Adj: List<int>[N]

procedure TSP
  MinimumSpanningTree()

  O = OddDegreeVertex()

  MinimumWeightedPerfectMatching()

  MultiGraph(O)

  circuit = FindEulerCircuit(H)

  return ToHamiltonianPath(circuit)
```

# 5 – Risultati ottenuti

Ho testato su varie instanze da 4, 10, 15 e 20 nodi, di grafi euclidei randomici, L’errore medio è comunque sotto il fattore massimo 3/2.

| Nodi | Errore medio |
|---|---|
| 4 | 0% |
| 10 | 9,29% |
| 15 | 3,77% |
| 20 | 10,71% |

## 5.1 – Tempi d’esecuzione

L’algoritmo ha complessità temporale $T(n)=O(n^4)$.

### 5.1.1 – Computers

È stato utilizzato un solo computer per i test.

| CPU | CPU GHz | RAM | HD | OS |
|---|---|---|---|---|
| Intel Core2 Quad Q6600 | 2.40 | 4GB DDR2 | SSD | Windows 10 |

### 5.1.2 – Risultati

Di seguito i risultati ottenuti.

| Nodi | Tempo |
|---|---|
| 25 | 00'00"01 |
| 100 | 00'00"02 |
| 500 | 00'00"08 |
| 1000 | 00'00"35 |
| 5000 | 00'15"52 |

# 6 – Commenti conclusivi

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 multi-thread, 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 euristici.

# 7 – Allegati

## 7.1 – Codice sorgente

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

# 8 – Riferimenti bibliografici


- Held, M. and Karp, R.M., 1962. A dynamic programming approach to sequencing problems. Journal for the Society for Industrial and Applied Mathematics, 1:10.
- Held, M. and Karp, R.M., 1970. The Traveling Salesman Problem and Minimum Spanning Trees. Operations Research, 18, 1138-1162.
- Christofides, N., 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Report 388, Graduate School of Industrial Administration, CMU.
- Lippman, S., Lajoie, J. and Moo, B., 2012. C++ Primer. Addison-Wesley Professional.