Ottimizzazione spaziale dell’algoritmo di Held – Karp per la risoluzione esatta del problema del commesso viaggiatore asimmetrico

Michele Maione[1]

1 – Abstract

Aaaaaa.

2 – Introduzione

Il problema del commesso viaggiatore (TSP, Traveling Salesman Problem) è trovare un ciclo hamiltoniano a costo minimo, dato un insieme di nodi, archi pesati. Nella sua forma originale, il problema era trovare il tour più breve di tutte le capitali degli Stati Uniti. Da allora, sono state avanzate innumerevoli variazioni per risolvere anche problemi correlati. Sfortunatamente, tutte le varianti del problema sono NP-Hard, viene quindi, principalmente, calcolato tramite euristiche.

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.

3 – 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 sottopercorso di un percorso di minima distanza è esso stesso di minima distanza; quindi calcola le soluzioni di tutti i sottoproblemi partendo dal più piccolo. Purtroppo non possiamo sapere quali sottoproblemi dobbiamo risolvere, quindi li risolviamo tutti.

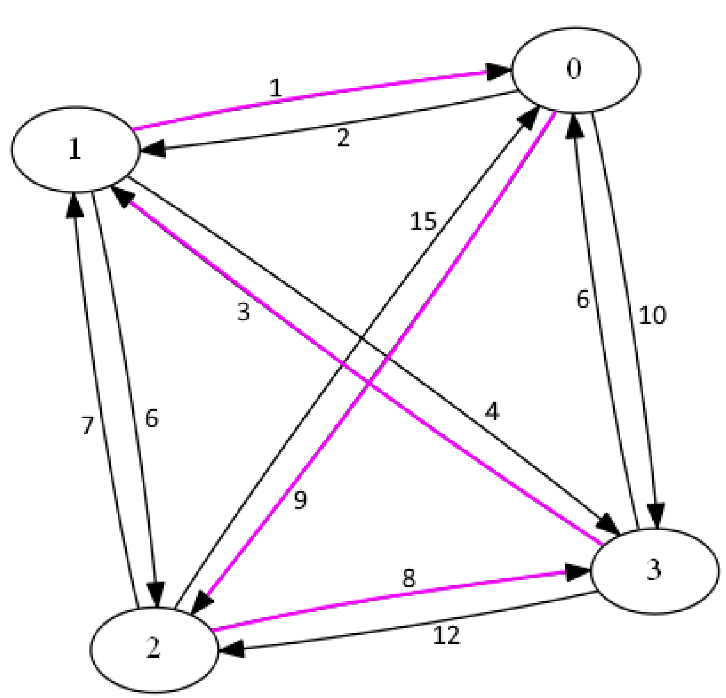

Figura 1: aTSP su 4 nodi

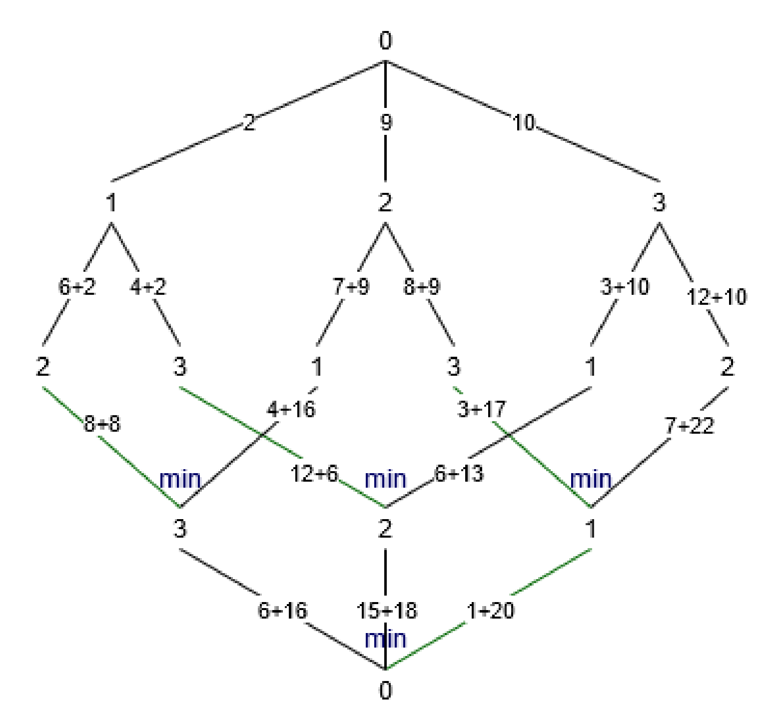

Figura 2: esecuzione algoritmo di Held – Karp

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.

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 stato utilizzare una matrice, ma sarebbe stata una matrice sparsa, per via della memorizzazione del set.

Ho scelto di codificare i set con questa funzione: $\mathtt{Hash(S)} : \mathbb{N}^n \rightarrow \mathbb{N} = \Sigma 2^i : \forall\ i \in S$. 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.

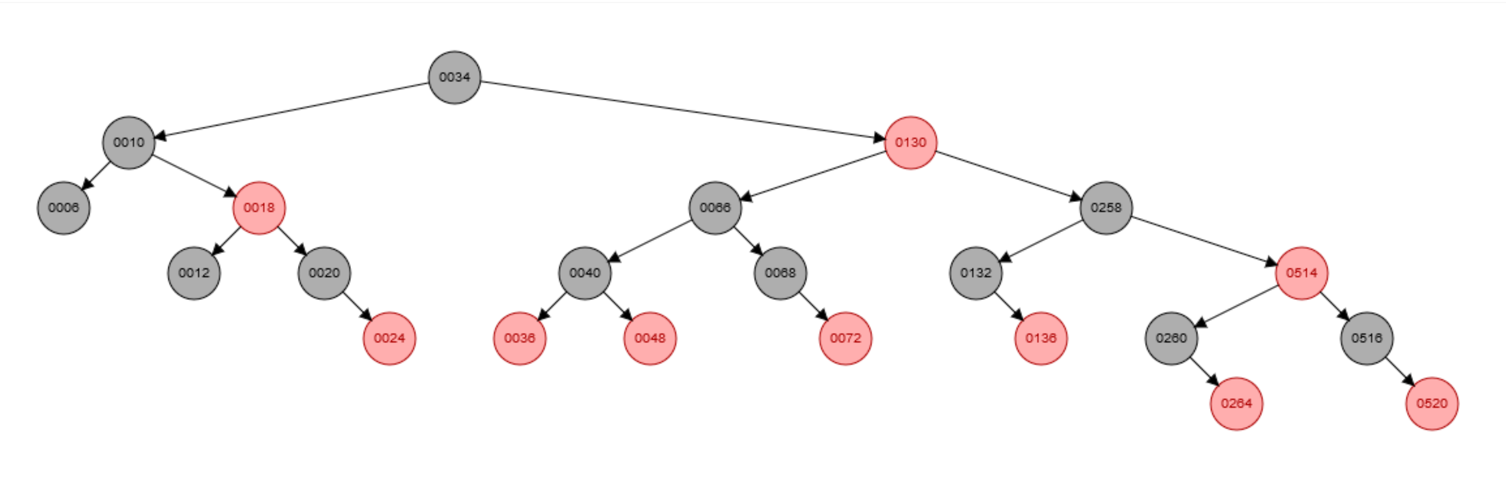

Figura 3: Albero rosso – nero contenente le rappresentazioni in int32 dei sets

3.1.3 – Algoritmo

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

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
end

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
      end

      i = S[0]
    end

    Q.dequeue()
  end

  opt = min_{k≠0}(C[Hash({1, ... , n - 1})][k] + d_{k,0})
end
```

4 – Risultati ottenuti

Aaaaaa.

4.1 – Memoria allocata

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

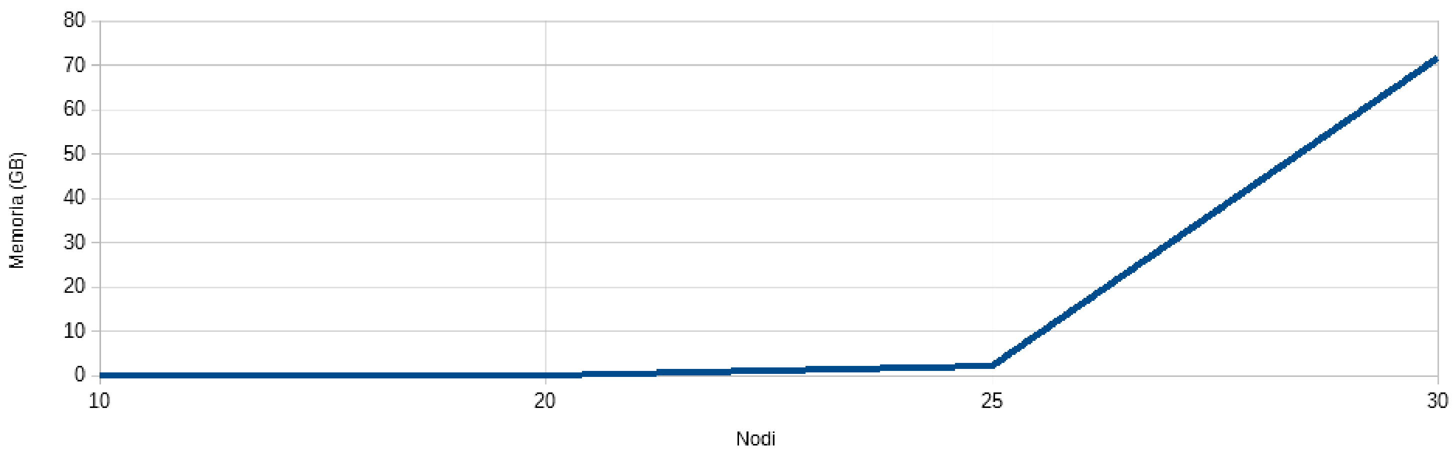

Figura 4: Memoria allocata, in GB, per l'elaborazione con 10, 20, 25 e 30 nodi

4.1.1 – Tempi d'esecuzione

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

4.1.2 – Computers

I computers utilizzati sono stati 4, la maggior parte quad-core.

Ho implementato una versione multithread dell'algoritmo, che eseguiva le righe 30-33, 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.

CPU	CPU GHz	RAM	HD	OS
AMD Ryzen5 2500U	2.00	8GB DDR4	SSD	Windows 10
Intel Celeron J1900	2.00	8GB DDR3	SSD	Windows 10
Intel Core2 Quad Q6600	2.40	4GB DDR2	SSD	Windows 10
Intel Core2 Duo E6550	2.33	4GB DDR2	SSD	Windows 10

4.1.3 – Risultati

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

PC	Nodi	Tempo
Core2 Quad	20	00'07"27
Core2 Duo	20	00'07"89
Ryzen5	20	00'08"47
Celeron	20	00'13"43
Ryzen5	25	14'51"70
Celeron	25	31'14"42

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'agoritmo viene eseguito sulla RAM, mentre per grafi più grandi viene attivato il paging.

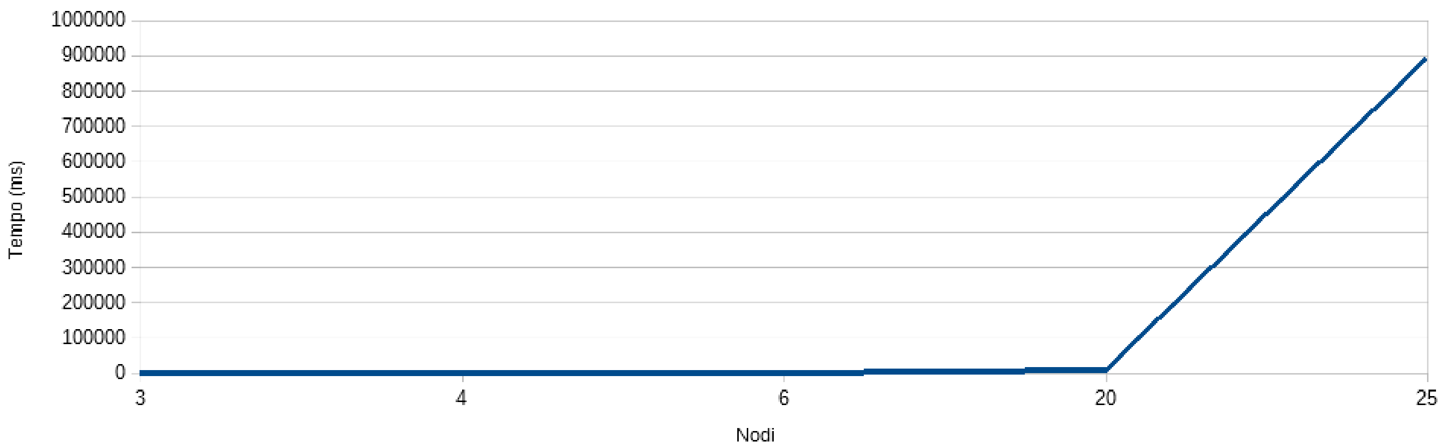

Figura 5: Tempi d'esecuzione, in ms, per l'elaborazione con 3, 4, 6, 20, 25 nodi

5 – Commenti conclusivi

Aaaaaa.

6 – 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.

Lippman, S., Lajoie, J. and Moo, B., 2012. C++ Primer. Addison-Wesley Professional.