

# Algoritmi per la risoluzione del TSP

Michele Maione – 931468 – [michele.maione@studenti.unimi.it](mailto:michele.maione@studenti.unimi.it).  
Ottimizzazione Combinatoria, A/A 2019-2020, Università degli Studi di Milano, via Celoria 18, Milano, Italia.

**Sommario** – Per la risoluzione del TSP ci sono 3 strategie e per queste ho implementato: per gli algoritmi esatti, l'algoritmo di programmazione dinamica di Held–Karp[3]; per gli algoritmi approssimati, il rilassamento lagrangiano di Held–Karp[4] e di Volgenant–Jonker[5]; per quanto riguarda i casi speciali, ho focalizzato la mia attenzione sul TSP metrico implementando l'algoritmo di Christofides[6] (1,5–approssimato) e un algoritmo 2–approssimato[1] (Thomas Cormen). Sono state fatte poi alcune comparazioni dei risultati ottenuti.

## 1 – 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.

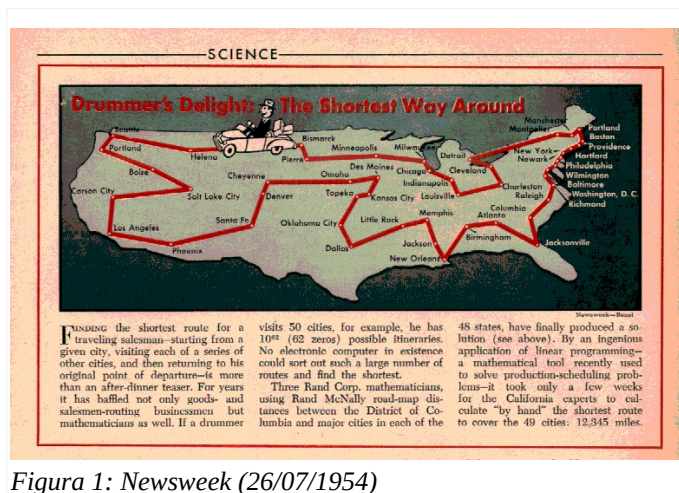


Figura 1: Newsweek (26/07/1954)

Matematicamente può essere rappresentato come un grafo pesato. Se il grafo è diretto allora il TSP è definito asimmetrico. Lo scopo consiste nel trovare un ciclo hamiltoniano a costo minimo sul grafo, questo è un problema NP-completo.

Le possibili strategie per risolvere il problema sono:

1. Algoritmi esatti, che funzionano abbastanza velocemente solo per problemi di piccole dimensioni;
2. Algoritmi approssimati o euristici, che forniscono soluzioni approssimative in un tempo ragionevole;
3. Individuazione di casi speciali del problema per i quali sono possibili euristiche migliori o esatte.

Per quanto concerne il caso 1, ho sviluppato:

- Algoritmo di programmazione dinamica di Held–Karp

Questo algoritmo, con una piccola ottimizzazione, riesce a calcolare la soluzione ottima per istanze di 25 nodi.

Invece per il caso 2 sono stati sviluppati i seguenti algoritmi:

- Rilassamento lagrangiano proposto da Held–Karp;
- Rilassamento lagrangiano proposto da Volgenant–Jonker.

Per il caso 3 è stato usato il caso speciale di TSP metrico (che è NP-difficile), in cui le distanze tra i nodi soddisfano la disuguaglianza triangolare, ed ho sviluppato:

- Algoritmo di Christofides;
- Algoritmo 2–approssimato (Cormen).

## 2 – MST

Un concetto che tornerà utile nei prossimi capitoli è l'albero di copertura di costo minimo (minimum spanning tree, MST), un albero ricoprente nel quale sommando i pesi degli archi si ottiene un valore minimo. Il modello matematico[2] è il seguente:

$$\min \sum_{(i,j) \in E} c_{ij} \cdot x_{ij} \quad (1)$$

$$S.T. \quad \sum_{(i,j) \in E} x_{ij} = n - 1 \quad (2)$$

$$\sum_{(i,j) \in (S)} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \quad (3)$$

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

La relazione (1) è la funzione obiettivo da minimizzare, che rappresenta il costo del cammino. Il vincolo (2) esprime che l'insieme degli archi contenga  $n-1$  elementi. Il vincolo (3) assicura l'assenza di sottocammini.

Gli algoritmi più conosciuti per il calcolo del MST con complessità temporale  $O(E \cdot \log V)$  sono:

- Algoritmo di Prim;
- Algoritmo di Kruskal;
- Algoritmo di Borůvka.

Ho implementato sia l'algoritmo di Prim che quello di Kruskal, ambedue le implementazioni usano le strutture dati in Tabella 2.

### Dettagli implementativi

L'algoritmo di Kruskal all'inizio dell'elaborazione ordina il vettore  $E$  per l'attributo  $cost$  in  $O(N \cdot \log N)$ .

L'algoritmo di Prim, usa un albero rosso-nero come contenitore per recuperare il nodo con la *key* minima. Un'altra soluzione sarebbe stata usare una coda di priorità ma poiché questa implementazione dell'algoritmo modifica gli attributi del nodo durante un ciclo *while*, risultava più semplice la gestione con un albero rosso-nero (in Tabella 1).

Tabella 1: Strutture dati utilizzate in Prim (Cormen[1])

```
bool comparator(Node *l, Node *r)
{
    return l->key < r->key;
};

set<Node *, comparator> rbTree;
```

Tabella 2: Strutture dati utilizzate per MST (Cormen[1])

```
struct Node
{
    Node *p;
    unsigned short rank = 0; // Kruskal
    float key = FLT_MAX; // Prim
};

struct Edge
{
    float cost = FLT_MAX;
    Node *from, *to;
};

struct Graph
{
    list<Node *> V;
    vector<Edge *> E;
};
```

### 3 – TSP

Il problema del commesso viaggiatore può essere rappresentato come un grafo pesato  $G = (V, E)$  a cui ad ogni arco è associato un peso/costo/distanza  $c$  e una variabile booleana  $x$  che indica se l'arco appartiene al percorso. La formulazione matematica[2] del problema è la seguente:

$$\min \sum_{(i,j) \in E} c_{ij} \cdot x_{ij} \quad (1)$$

$$S.T. \quad \sum_{(i,j) \in \delta(l)} x_{ij} = 2 \quad \forall l \in V \quad (2)$$

$$\sum_{(i,j) \in E(S)} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \quad (3)$$

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

La relazione (1) è la funzione obiettivo da minimizzare, che rappresenta il costo del cammino. Il vincolo (2) indica che ogni nodo ha grado 2. Il vincolo (3) assicura l'assenza di sotto-cammini.

Riformuliamo il vincolo (2):

$$\sum_{(i,j) \in \delta(l)} x_{ij} = 2 \quad \forall l \in V \setminus \{r\} \quad (5)$$

$$\sum_{(i,j) \in E} x_{ij} = n \quad (6)$$

Il vincolo (5) indica che ogni nodo, tranne il nodo  $r$ , ha grado 2 e il (6) che ci sono  $n$  archi.

Infine riformuliamo il problema così:

$$\min \sum_{(i,j) \in E} c_{ij} \cdot x_{ij} \quad (7)$$

$$S.T. \quad \sum_{(i,j) \in \delta(l)} x_{ij} = 2 \quad \forall l \in V \setminus \{r\} \quad (8)$$

$$\sum_{(i,j) \in E} x_{ij} = n \quad (9)$$

$$\sum_{(i,j) \in E(S)} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \quad (10)$$

$$x_{ij} \in \{0,1\} \quad \forall (i,j) \in E \quad (11)$$

Questa formulazione del problema è importante perché i vincoli (9), (10) e (11) sono molto simili a quelli del problema del MST visto al capitolo precedente. Infatti sono i vincoli del 1–albero di copertura di costo minimo, un albero di copertura di costo minimo con due archi minimi sul vertice 1.

Di seguito il modello matematico del 1–albero di copertura di costo minimo:

$$\min \sum_{(i,j) \in E} c_{ij} \cdot x_{ij} \quad (1)$$

$$S.T. \quad \sum_{(i,j) \in E} x_{ij} = n \quad (2)$$

$$\sum_{(i,j) \in E(S)} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \quad (3)$$

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

Dalla formulazione matematica soprastante si nota che il vincolo (2) del MST si trasforma, per questo tipo di problema, proprio nel vincolo (9) del TSP.

Continueremo questo discorso nel capitolo Algoritmi approssimati: il rilassamento lagrangiano.

### 4 – Algoritmi esatti: l'algoritmo di programmazione dinamica di Held-Karp

Anche se ormai è considerata una curiosità storica di scarso interesse, vale la pena citare l'algoritmo di programmazione dinamica di Held-Karp proposto nel 1962 per risolvere il aTSP<sup>1</sup>. 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. Non potendo co-

1 L'algoritmo risolve il problema del aTSP e ovviamente anche quello del TSP.

noscere quali sotto-problemi risolvere, devono essere risolti tutti. L'algoritmo ha una complessità temporale  $O(2^n \cdot n^2)$  e una complessità spaziale di  $O(2^n \cdot n)$ , portata poi a  $O(2^n \cdot \sqrt{n})$ .

In Figura 2 il cammino minimo su un grafo diretto  $K_4$ . In Figura 3, in verde, è evidenziato il minimo tra le distanze.

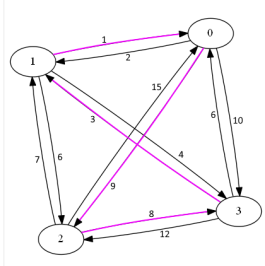


Figura 2: aTSP

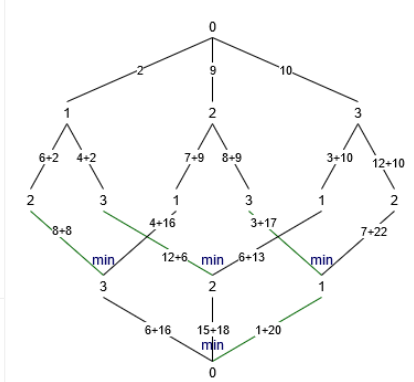


Figura 3: Esecuzione

tice 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 vertici  $V$ , inserendoli nella funzione obiettivo come somma pesata secondo dei moltiplicatori lagrangiani  $\lambda_l$ . Nella tabella successiva è possibile notare che 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.

$$L(\lambda) = \min \sum_{(i,j) \in E} c_{ij} \cdot x_{ij} + \sum_{l \in V \setminus \{r\}} \lambda_l \cdot \left( \sum_{(i,j) \in \delta(l)} x_{ij} - 2 \right) \quad (1)$$

$S.T.$

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

$$\sum_{(i,j) \in E(S)} x_{ij} \leq |S| - 1 \quad \forall S \subset V, S \neq \emptyset \quad (3)$$

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

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.

### 5.1 – Metodo del sub-gradiente

Sia Held-Karp che Volgenant-Jonker propongono una serie di equazioni per il calcolo di una successione di  $\lambda_l (\forall l \in V)$  secondo le seguenti regole euristiche, tra loro differenti che cambiano leggermente i valori restituiti ma non l'algoritmo. Il parametro  $\alpha=2$  decresce durante l'esecuzione, mentre  $M$  è il numero massimo di iterazioni, UB è l'upper bound calcolato con un algoritmo veloce. Ho usato un algoritmo 2-approssimato eseguito in  $\Theta(V^2)$ .

Tabella 3: Held-Karp

$$\lambda_i^{k+1} = \lambda_i^k + t^k \cdot (d_i^k - 2) \quad (1)$$

$$t^k = \frac{\alpha^k [UB - L(\lambda^k)]}{\sum_{i \in V} (d_i(x^k) - 2)^2} \quad (2)$$

Tabella 4: Volgenant-Jonker

$$\lambda_i^{k+1} = \lambda_i^k + 0.6 \cdot t^k \cdot (d_i^k - 2) + 0.4 \cdot t^k \cdot (d_i^{k-1} - 2) \quad (1)$$

$$t^k = t^1 \cdot \frac{k^2 - 3 \cdot (M-1) \cdot k + M \cdot (2 \cdot M - 3)}{2 \cdot (M-1) \cdot (M-2)} \quad (2)$$

$$M = \frac{n^2}{50} + n + 16 \quad (3)$$

$$t^1 = \frac{L(\lambda^1)}{n} \quad (4)$$

## 4.1 – Ottimizzazione

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  $s - 2$ , 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, riducendo del 15% la memoria utilizzata.

Nella seguente tabella di esempio di 5 nodi, ho evidenziato con colori diversi i set di nodi che dipendono tra di loro. Resta il fatto che la dipendenza è relativa solo alla 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}

## 5 – Algoritmi approssimati: il rilassamento lagrangiano

Gli algoritmi con rilassamento lagrangiano di Held-Karp[4] e Volgenant-Jonker[5], risalgono rispettivamente al 1969 e al 1980. Si basano sul concetto che un ciclo hamiltoniano è un 1-albero in cui ogni ver-

## 6 – Branch and bound

L'algoritmo di branch and bound[2] costituisce un metodo enumerativo per la risoluzione di problemi di ottimizzazione lineare, basato sull'esplorazione parzialmente implicita della regione ammissibile, in quanto una parete delle soluzioni ammissibili vengono escluse dalla ricerca della soluzione ottimale senza essere esplicitamente analizzate.

Nel TSP la ramificazione[5] ha luogo impostando certi archi come *richiesti* o *vietati*, cosicché ogni sottoinsieme è caratterizzato da un'insieme  $R$  di archi *richiesti* ed un'insieme  $F$  di archi *vietati*.

Durante la computazione due semplici regole sono usate per gli archi:

1. Se 2 archi incidenti su un nodo sono *richiesti*, tutti gli altri archi incidenti a quel nodo possono essere *vietati*;
2. Se non ci sono più di 2 archi non *vietati* incidenti in un nodo, allora sono *richiesti* entrambi gli archi. Infatti ogni arco deve avere grado 2.

La ramificazione inizia determinando il punto  $p$  con grado maggiore di 2 sul sotto-cammino dell'attuale miglior 1-albero. Per la regola (1) ci sono 2 archi non *richiesti* sul nodo  $p$ :  $e_1, e_2$ . Detto  $S(R, F)$  l'attuale insieme delle soluzioni fattibili allora la sua ramificazione sarà  $\{S1, S2, S3\}$ :

$S1(R \cup \{e_1, e_2\}, F)$

$S2(R \cup \{e_1\}, F \cup \{e_2\})$

$S3(R, F \cup \{e_1\})$

Per quanto riguarda la scelta di  $p, e_1, e_2$ , ci sono 3 regole:

1. Se possibile il nodo  $p$  incidente ad un arco *richiesto*;
2. Se ci sono più nodi  $p$  a disposizione allora quello con il minor numero di archi fattibili incidenti ad esso;
3.  $e_1$  non deve essere parte di un sotto-cammino.

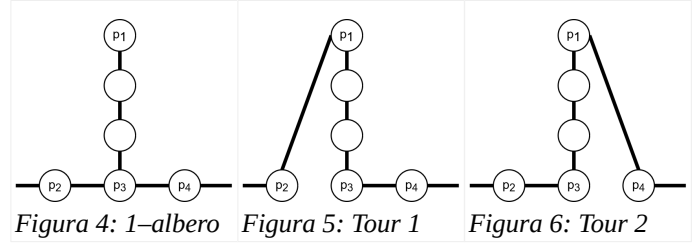
### 6.1 – Euristiche per l'upper bound

Volgenant–Jonker[5] suggeriscono di utilizzare l euristica per l'upper bound solamente sugli 1-alberi che contengono un ciclo e ramificazioni con grado non maggiore di 2 come in Figura 4.

Per ogni ramificazione troviamo il nodo  $p_1$  con grado 1, il nodo  $p_3$  sul ciclo, e identifichiamo i nodi  $p_2$  e  $p_4$  connessi a  $p_3$ . Ci sono 2 possibili cammini che possono essere creati e sono quelli della Figura 5 e Figura 6. Bisogna scegliere l'arco con il peso minore. A questo punto se il costo attuale del cammino è

maggiore dell'upper bound corrente bisogna terminare la ricerca in questa ramificazione.

Tabella 5: 2 possibili modifiche di 1-albero in un cammino



### Dettagli implementativi

Per l'implementazione del branch and bound ho seguito Volgenant–Jonker[5], le strutture dati utilizzate sono quelle in Tabella 6, gli MST sono calcolati utilizzando l'algoritmo di Prim, e la coda utilizzata per la ramificazione è stata implementata con un vettore che viene ordinato solo all'aggiunta di un nuovo elemento in tempo  $O(N \cdot \log N)$  sull'attributo *bound*.

Tabella 6: Strutture dati utilizzate per Branch and bound

```
struct Node
{
    float bound = 0;
    vector<float> λ;
    vector<Edge*> R, F;
    Tree *oneTree;
};

struct Edge
{
    Node *from, *to;
};

struct Tree
{
    vector<Edge*> T;
};

bool comparator(Node *l, Node *r)
{
    return l->bound > r->bound;
};

vector<Node*> queue;
```

## 7 – Casi speciali: il TSP metrico e l'algoritmo di Christofides

Il TSP metrico è un caso particolare del TSP, in cui si introducono delle restrizioni:

- È simmetrico;
- Le distanze sono tutte non negative;
- Le distanze soddisfano la disuguaglianza triangolare.

L'algoritmo di Christofides[6] è stato progettato nel 1976 per trovare soluzioni approssimative al proble-



ma del TSP metrico. Garantisce un'approssimazione di 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.  $S$  = sotto-grafo indotto di  $G$  usando i nodi in  $O$ ;
4.  $M$  = accoppiamento perfetto di peso minimo su  $S$ ;
5.  $H$  = multi-grafo connesso formato dagli archi di  $T$  e  $M$ ;
6.  $C$  = cammino euleriano su  $H$ ;
7.  $Z$  = cammino hamiltoniano da  $C$ , saltando i vertici ripetuti.

$Z$  sarà la soluzione del TSP.

## 7.1 – Implementazione

Gli algoritmi interni utilizzati sono:

- Albero ricoprente minimo: algoritmo di Prim  $O(E \cdot \log V)$ ;
- Accoppiamento perfetto di peso minimo: algoritmo dei cammini, alberi e fiori di Jack Edmonds[7]  $O(V^2 \cdot E)$ .

## 7.2 – Algoritmo dei cammini, alberi e fiori (Blossom algorithm)

Ideato da Jack Edmonds nel 1965 serve a trovare l'abbinamento massimo in un grafo. Inizia con un abbinamento vuoto e lo migliora iterativamente aggiungendo archi, uno alla volta, per costruire cammini aumentanti nell'abbinamento  $M$ . L'aggiunta a un cammino aumentante può far crescere un abbinamento poiché ogni altro arco in un cammino aumentante è un arco nell'abbinamento; man mano che vengono aggiunti archi al cammino aumentante, vengono scoperti altri archi dell'abbinamento. L'algoritmo ha tre possibili risultati dopo ogni iterazione:

1. Non trova cammini aumentanti, nel qual caso ha trovato un abbinamento massimo;
2. Trova un cammino aumentante che migliorerà il risultato;
3. Trova un fiore per manipolarlo e scoprire un nuovo cammino aumentante.

L'algoritmo viene eseguito in  $O(V^2 \cdot E)$  ed è quello che influisce sul tempo d'esecuzione dell'algoritmo di Christofides.

### Manipolazione dei fiori

Nel caso in cui il fiore sia dispari viene contratto come in Figura 7. L'algoritmo continua e trova un cammino aumentante tra 5 e 7, allora espande in ordine inverso i fiori, e ricostruisce i cammini aumentanti correttamente come in Figura 8.

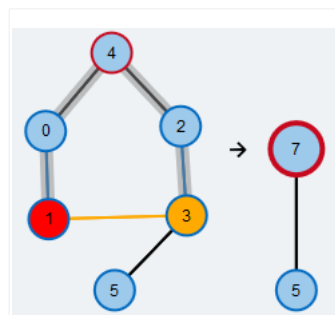


Figura 7: Contrazione

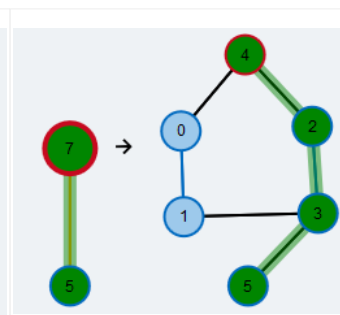


Figura 8: Espansione

## 8 – Un algoritmo 2-approssimato per la risoluzione del TSP metrico (Thomas Cormen)

Questo è un algoritmo di approssimazione a fattore costante 2 che viene eseguito in  $\Theta(V^2)$ . 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 vertici, ordinata in base a quando un vertice viene visitato per primo in un attraversamento anticipato di  $T$ ;
4. Ritorna il ciclo hamiltoniano da  $H$ .

## 9 – Risultati ottenuti

Ho testato su varie istanze 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-appros.
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 che l'algoritmo di Christofides calcolare un tour peggiore.

L'algoritmo di Christofides ha complessità temporale  $O(V^2 \cdot E)$  mentre il 2-approssimato  $\Theta(V^2)$ . Di seguito i risultati ottenuti:

Nodi	Tempo d'esecuzione		
	Christofides	Held-Karp	2-appros.
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

## 10 – Commenti conclusivi

### D.P. Held–Karp

La soluzione di programmazione dinamica risulta, ad oggi, l'unica in grado di trovare la soluzione ottima, ma essendo esponenziale il suo utilizzo è limitato a topologie di una ventina di nodi.

### Branch and bound basato sul rilassamento lagrangiano

Questi algoritmi sono stati quelli che hanno dato i risultati migliori, ma sono molto più lenti rispetto all'algoritmo di Christofides, sono anche soggetti ad una giusta scelta di parametri euristici.

### 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, è facile da implementare e il fatto che utilizzi altri algoritmi può avere un margine di miglioramento per quanto riguarda la velocità d'esecuzione utilizzando algoritmi computazionalmente migliori per i suoi sotto-problemi.

### 2–Approssimato

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

## 11 – Gioco icosiano

A quanto pare il problema del commesso viaggiatore fu formulato matematicamente nel '800 dal matematico irlandese William Rowan Hamilton e dal matematico britannico Thomas Kirkman basandosi sul gioco icosiano di Hamilton che era un puzzle ricreativo basato sulla ricerca di un ciclo hamiltoniano.



Figura 9: Gioco icosiano di Hamilton (1857)

## 12 – Allegati

### Codice sorgente

Tutto il progetto è disponibile con licenza MIT qui: <http://github.com/mikymaione/Held-Karp-algorithm>

## 13 – Riferimenti bibliografici

1. Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C., 2010. Introduzione agli algoritmi e strutture dati. McGraw-Hill.
2. Vercellis, C., 2008. Ottimizzazione. Teoria, metodi, applicazioni. McGraw-Hill.
3. Held, M. and Karp, R., 1962. A dynamic programming approach to sequencing problems. Journal for the Society for Industrial and Applied Mathematics, 1:10.
4. Held, M. and Karp, R., 1970. The Traveling Salesman Problem and Minimum Spanning Trees. Operations Research, 18, 1138-1162.
5. 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.
6. Christofides, N., 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Report 388, Graduate School of Industrial Administration, CMU.
7. Edmonds, J., 1965. Paths, trees, and flowers. Can. J. Math, 17: 449–467.
8. Lippman, S. and Lajoie, J. and Moo, B., 2012. C++ Primer. Addison-Wesley Professional.