Università degli Studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Corso di Laurea in Informatica

Tesi di Laurea

Analisi e implementazione di Euristiche per il Traveling Salesman Problem

Candidato
Brian Riccardi

Relatore
Prof. Giuseppe Lancia

Anno accademico 2017-2018

Contatti dell'Istituto
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
http://www.dimi.uniud.it/

Indice

Introduzione

Il Problema del Commesso Viaggiatore consiste nella ricerca del Circuito Hamiltoniano di costo minimo in un grafo pesato. Formalmente, dato un grafo $G = (V, E)$ e una funzione di peso $w : E \to \mathbb{N}$ siamo interessati ad una permutazione dei vertici del grafo $(v_{i_1}, v_{i_2}, \ldots, v_{i_{|V|}})$ che minimizzi la quantità

$$W = w(v_{i_{|V|}}, v_{i_1}) \; + \sum_{j=1}^{|V|-1} w(v_{i_j}, v_{i_{j+1}})$$

Nell'ambito della Ricerca Operativa e dell'Ottimizzazione Combinatoria, l'importanza del TSP (dall'inglese *Traveling Salesman Problem*) deriva dal fatto di poterlo utilizzare per descrivere un ricco insieme di *real world problems* riguardo, ad esempio, Cristallografia, *Vehicle routing*, *Scheduling*, Robotica, ecc...

Dal punto di vista teorico, la versione decisionale di TSP è notoriamente NP-Completa e ciò lo rende, nella pratica, intrattabile anche per piccole istanze del problema. Dalla necessità di risolvere il TSP e dall'impossibilità (assumendo $P \neq NP$) di poterlo fare efficientemente per una qualsiasi istanza del problema, nasce il concetto di "Euristica", ovvero una tecnica che ha lo scopo di trovare in un tempo ragionevole un'approssimazione sufficientemente buona della soluzione esatta.
Nell'ambito del TSP possiamo dividere le euristiche in tre classi:

- Tour construction: l'algoritmo costruisce il circuito aggiungendo, passo dopo passo, un nuovo vertice a seconda di una strategia più o meno elaborata
- Tour improvement: a partire da un circuito arbitrario, l'algoritmo esegue delle "mosse" che hanno lo scopo di migliorarlo
- Composite: vengono combinate le tecniche di costruzione e di miglioramento

In questa tesi studieremo il Commesso Viaggiatore Simmetrico (STSP), ovvero assumeremo che il grafo sia non diretto, completo e con funzione di peso simmetrica. Presenteremo delle euristiche appartenenti ad ognuna delle classi e ne analizzeremo le performance sia dal punto di vista dell'efficienza in tempo, sia dal punto di vista dell'efficacia nel senso della bontà della soluzione trovata rispetto all'ottimo. Gli algoritmi scelti sono Nearest Neighbor, Nearest Insertion, Christofides, 3-OPT (in tre versioni) e Lin-Kernighan-Helsgaun: per quanto riguarda Christofides e Lin-Kernighan-Helsgaun si è deciso di implementare una versione più semplice degli algoritmi in letteratura; possiamo quindi dire che le euristiche implementate siano solamente ispirate a quelle descritte in [1], [2]; per quanto riguarda 3-OPT, si è scelto di implementare la versione descritta in [5].

I test sono stati effettuati su un insieme di grafi selezionati da TSPLIB[9] e su un insieme di grafi generati casualmente. Le misurazioni sono state effettuate seguendo le procedure descritte in [7]. Tutti i programmi sono stati scritti in `C++`.

1

Il Commesso Viaggiatore

Il Problema del Commesso Viaggiatore può essere formulato, informalmente, in questo modo:

"Un commerciante deve visitare n città in modo tale da visitarle tutte una e una sola volta, ritornando al punto di partenza. Ogni coppia di città (i, j) è collegata da una strada e ogni strada, per essere percorsa, necessita il pagamento di un pedaggio. Il commerciante, essendo molto avido, vuole trovare il percorso che gli assicuri di spendere il meno possibile."

Nonostante la facilità con cui venga espresso, il Commesso Viaggiatore è tutt'altro che semplice da risolvere. È un problema di Ottimizzazione Combinatoria noto da decenni e, date le sue potenzialmente infinite applicazioni pratiche, è ancora oggi uno dei problemi più studiati. Dal punto di vista formale, il TSP (dall'inglese *Traveling Salesman Problem*) consiste nella ricerca del Circuito Hamiltoniano di costo minimo in un grafo pesato. È possibile dare diverse formulazioni equivalenti al problema: di seguito ne diamo due.

Definizione 1.0.1 (**TSP su grafi**). *Sia $G = (V, E)$ un grafo di n vertici completo e pesato con funzione di peso $w : E \to \mathbb{N}$. Vogliamo determinare una permutazione dei vertici $(v_1, v_2, \ldots, v_n)$ tale che*

$$w(v_n, v_1) + \sum_{i=1}^{n-1} w(v_i, v_{i+1})$$

sia minima.

Definizione 1.0.2 (**TSP su matrici**). *Sia $n \in \mathbb{N}_+$ e $C = (c_{i,j})$ una matrice $n \times n$ a valori in $\mathbb{N}$ tale che $c_{i,i} = 0$ per ogni $i \in \{1, 2, \ldots, n\}$. Determinare una permutazione $(i_1, i_2, \ldots, i_n)$ di $(1, 2, \ldots, n)$ tale che*

$$c_{i_n, i_1} + \sum_{j=1}^{n-1} c_{i_j, i_{j+1}}$$

sia minima.

Le due definizioni sono banalmente equivalenti e ne verrà utilizzata una piuttosto che l'altra in base al contesto. Inoltre, diamo alcune definizioni "classiche" e convenzioni.

Definizioni.

(1) Un tour è una permutazione dei vertici.

(2) Costo e peso verranno utilizzati come sinonimi.

(3) Il Commesso Viaggiatore Simmetrico (STSP) consiste nell'ulteriore ipotesi che la matrice C sia simmetrica.

(4) Per istanza geometrica si intende che w soddisfa la disuguaglianza triangolare.

(5) Per istanza euclidea si intende che w è la funzione di distanza euclidea.

La relazione che TSP sembra avere con il problema del Circuito Hamiltoniano (HAM-CYCLE) sottintende una difficoltà computazionale quantomeno simile. Definiamo innanzitutto il linguaggio che stiamo considerando.

Definizione 1.0.3 (**TSP** come linguaggio formale)**.**

$$\begin{aligned} \boldsymbol{TSP} = \{\langle G, w, k\rangle \ : & \ G = (V, E) \text{ } \textit{è un grafo completo}, \\ & \ w : E \to \mathbb{R} \text{ } \textit{è una funzione di costo}, \\ & \ k \in \mathbb{N}, \\ & \ G \text{ } \textit{possiede un tour hamiltoniano di costo} \ \leq k\} \end{aligned}$$

Teorema 1.0.1. ***TSP*** *è NP-Completo.*

Dimostrazione. La dimostrazione viene fatta separando l'appartenenza ad NP dalla hardness.

TSP $\in$ NP: consideriamo un'istanza di **TSP** e una soluzione. La verifica della soluzione può essere svolta in tempo polinomiale: si controlla che ogni vertice appartenga alla soluzione e si controlla che la somma dei pesi degli archi sia al più k. Per *Guess&Verify* **TSP** è in NP.

TSP $\in$ NP-hard: mostriamo che **HAM-CYCLE** $\preceq$ **TSP**.

Sia $G = (V, E)$ un'instanza di **HAM-CYCLE**. Costruiamo G', completo e pesato con funzione

$$w(i,j) = \begin{cases} 0 & \text{se } (i,j) \in E \\ 1 & \text{se } (i,j) \notin E \end{cases}$$

Consideriamo quindi l'istanza $\langle G', w, 0\rangle$ di **TSP**. Si ha quindi che se $\langle G\rangle \in$ **HAM-CYCLE** allora esiste un circuito hamiltoniano in G, ma per costruzione G' ammette un circuito hamiltoniano di costo 0, quindi $\langle G', w, 0\rangle \in$ **TSP**.

Al contrario, se $\langle G', w, 0\rangle \in$ **TSP** allora G' ammette un circuito hamiltoniano di costo 0 e, per w, ogni arco del circuito ha costo 0. Ma allora ogni arco del circuito è in E e quindi anche G ammette un circuito hamiltoniano, quindi $\langle G\rangle \in$ **HAM-CYCLE**.

□

2

Euristiche per il TSP

Dalla *Hardness* di TSP si evince che una sua risoluzione esatta, per ogni istanza del problema, sia computazionalmente intrattabile anche per grafi di modeste dimensioni. L'importanza dei risvolti applicativi, tuttavia, ci costringe a non abbandonare completamente il suo studio e anzi ci spinge ad indagare più a fondo alla ricerca di tecniche per aggirare il vincolo computazionale. Vi sono infatti almeno tre strade che possiamo percorrere: se l'istanza è piccola allora un approccio *bruteforce* è ancora utilizzabile; è possibile tentare di risolvere casi particolari, (*e.g.* un'istanza euclidea che ammette una *collana*[10]); è possibile accettare soluzioni *quasi-ottime*. Quest'ultimo è l'approccio di tipo euristico e, generalmente, soluzioni vicine all'ottimalità sono più che sufficienti. Occorre quindi definire chiaramente quello che intendiamo quando parliamo di soluzioni "quasi-ottime".

Definizione 2.0.1 (**ρ-approssimazione**)**.** *Sia $\mathcal{P}$ un problema di minimizzazione, $\mathcal{A}$ un'euristica per $\mathcal{P}$ e $\rho : \mathbb{N} \to \mathbb{R}$. Diremo che $\mathcal{A}$ è ρ-approssimato se per ogni istanza $x \in \mathcal{P}$ di dimensione n con soluzione esatta $\boldsymbol{OPT}$ si ha $\mathcal{A}(n) \leq \rho(n)\boldsymbol{OPT}$. Diremo inoltre che ρ è il rapporto di approssimazione di $\mathcal{A}$.*

Nel caso del TSP possiamo suddividere le euristiche in tre categorie.

2.1 Tour construction

Nelle euristiche di tipo "tour construction" il circuito viene costruito incrementalmente e l'algoritmo termina appena il tour così creato è valido. Di seguito elenchiamo diversi esempi di euristiche basate sul tour construction.

2.1.1 Strip (`STRIP`)

Questa euristica viene applicata su istanze *geometriche* di STSP. L'idea è quella di cercare il minimo rettangolo che contiene tutti i punti del problema, suddividerlo in $\sqrt{\frac{N}{3}}$ *strips* verticali e costruire il tour immaginando di percorrere ogni strip verticalmente, alternando il verso di percorrenza ad ogni nuova strip. Il tour verrà quindi chiuso collegando l'ultimo punto con il primo. La ricerca del rettangolo minimo può essere svolta linearmente scandendo l'insieme dei punti e memorizzandone le coordinate

estremali (min/max ascissa e min/max ordinata). La suddivisione dei punti nelle strisce può essere ancora svolta linearmente. La scansione ordinata dei punti nella strip può essere simulata ordinando preventivamente i punti per ordinata scegliendo un sorting in senso crescente o decrescente a secondo di quale strip si consideri. I vantaggi principali di questa euristica sono senza dubbio la semplicità concettuale e implementativa, nonché la velocità di esecuzione. Tuttavia, è facile notare come `STRIP` abbia un rapporto di approssimazione $\rho(n) \in \Omega(\sqrt{n})$: si pensi infatti ad un'instanza in cui i punti siano ai limiti delle strip. Per punti uniformemente distribuiti nel quadrato unitario, il tour prodotto è in media non più lungo di $0.93\sqrt{n}$[4].

2.1.2 Nearest Neighbor (`NN`)

L'idea è quella di avere, ad ogni passo, un tour parziale da estendere aggiungendo un vertice non ancora presente che sia a distanza minima dal vertice aggiunto al passo precedente.

Algorithm 1

```
1:  function NearestNeighbor(G = (V, E), w)
2:      n ← |V|
3:      v ← RandomIn(V)
4:      T ← {v}
5:      F ← V \ T
6:      for i ← 2, i ≤ n, i ← i + 1 do
7:          candidates ← ∅
8:          bestCost ← ∞
9:          for each u ∈ F do
10:             c ← w(v, u)
11:             if c < bestCost then
12:                 candidates ← {u}
13:                 bestCost ← c
14:             else if c = bestCost then
15:                 candidates ← candidates ∪ {u}
16:             end if
17:         end for
18:         v ← RandomIn(candidates)
19:         T ← T ∪ {v}
20:         F ← F \ {v}
21:     end for
22:
23:     return T
24: end function
```

Assumendo la disuguaglianza triangolare, `NN` ha un rapporto di approssimazione $\rho(n) = \frac{1+\lceil \log n \rceil}{2}$ [3].

2.1.3 Nearest Insertion (NI)

Viene scelto un vertice iniziale e ad ogni passo viene aggiunto il vertice libero più vicino al tour nella posizione che minimizzi l'incremento di costo.

Algorithm 2

```
1:  function NearestInsertion(G = (V, E), w)
2:      n ← |V|
3:      v ← RandomIn(V)
4:      d ← new Array(w(v))          ▷ For each vertex u, d(u) is the minimum distance from T
5:      T ← {v}
6:      F ← V \ T
7:      for i ← 2, i ≤ n, i ← i + 1 do
8:          candidates ← ∅
9:          bestCost ← ∞
10:         for each u ∈ F do
11:             c ← d(u)
12:             if c < bestCost then
13:                 candidates ← {u}
14:                 bestCost ← c
15:             else if c = bestCost then
16:                 candidates ← candidates ∪ {u}
17:             end if
18:         end for
19:
20:         v ← RandomIn(candidates)
21:         NearestInsert(v, T, i − 1, w)
22:         F ← F \ {v}
23:
24:         for each u ∈ F do
25:             d(u) ← min{d(u), w(v, u)}
26:         end for
27:     end for
28:
29:     return T
30: end function
```

Algorithm 3

```
1: function NearestInsert(v, T, m, w)          ▷ Insert v in the "best" position with respect to NI
2:     p ← m
3:     bestCost ← costToInsert(t_m, v, t_1, w)
4:     for i ← 1, i < m, i ← i + 1 do
5:         c ← costToInsert(t_i, v, t_{i+1})
6:         if c < bestCost then
7:             bestCost ← c
8:             p ← i
9:         end if
10:    end for
11:
12:    insertIn(v, T, p)
13: end function
14:
15: function costToInsert(x, v, y, w)          ▷ Cost of inserting v between x and y
16:    return w(x, v) + w(v, y) − w(x, y)
17: end function
```

Assumendo la disuguaglianza triangolare, `NI` ha un rapporto di approssimazione $\rho(n) = \left(2 - \frac{2}{n}\right)$[3].

2.1.4 Christofides (`CHR`)

L'algoritmo di Christofides costruisce il tour da un circuito euleriano su un particolare multigrafo costruito da G.

Algorithm 4

```
1: function Christofides(G = (V, E), w)
2:     Tree ← Prim(G, w)
3:     O ← getOdds(Tree, V, n)
4:     M ← minWeightMatching(O, w)
5:     G ← new multigraph(M, Tree)
6:     E ← findEulerTour(G)
7:     T ← shortcut(E)                         ▷ Delete occurrences after the first
8:
9:     return T
10: end function
```

Algorithm 5

```
1: function PRIM(G = (V, E), w)
2:     n ← |V|
3:     v ← RandomIn(V)
4:     d ← new Array(w(v))
5:     p ← new Array(n, v)
6:     T ← {v}
7:     F ← V \ T
8:
9:     for i ← 2, i ≤ n, i ← i + 1 do
10:        bestCost ← ∞
11:        for each u ∈ F do
12:            c ← d(u)
13:            if c < bestCost then
14:                v ← u
15:                bestCost ← c
16:            end if
17:        end for
18:
19:        T ← T ∪ {{p(v), v}}
20:        F ← F \ {v}
21:        for each u ∈ F do
22:            c ← w(v, u)
23:            if c < d(u) then
24:                d(u) ← c
25:                p(u) ← v
26:            end if
27:        end for
28:    end for
29:
30:    return T
31: end function
```

Algorithm 6

```
1: function GETODDS(Tree, V, n)
2:     deg ← new Array(n, 0)
3:     for each {u, v} ∈ Tree do
4:         deg(u) ← deg(u) + 1
5:         deg(v) ← deg(v) + 1
6:     end for
7:
8:     O ← ∅
9:     for each v ∈ V do
10:        if deg(v) ≡_2 1 then
11:            O ← O ∪ {v}
12:        end if
13:    end for
14:
15:    return O
16: end function
```

Il minimum weight perfect matching dei vertici di $\mathcal{O}$ esiste sempre in virtù del seguente

Teorema 2.1.1. *In ogni albero T vi è sempre un numero pari di vertici di grado dispari.*

Dimostrazione. Sia T un albero di n vertici e $n-1$ archi. Denotiamo con p_i il grado dell'i-esimo vertice di grado pari e con d_i il grado dell'i-esimo vertice di grado dispari. È noto che la somma dei gradi di un grafo non diretto qualsiasi sia il doppio del numero degli archi (*handshake lemma*). Supponiamo per assurdo che vi siano $2m+1$ vertici di grado dispari, allora

$$\begin{aligned} 2(n-1) = \sum_{i=1}^{n} deg(v_i) &= p_1 + p_2 + \cdots + p_k + d_1 + d_2 + \cdots + d_{2m+1} \\ &= 2t + (2a_1 + 1) + (2a_2 + 1) + \cdots + (2a_{2m+1} + 1) \\ &= 2t + 2(a_1 + a_2 + \cdots + a_{2m+1}) + 2m + 1 \\ &= 2(t + a_1 + a_2 + \cdots + a_{2m+1} + m) + 1 \\ &= 2q + 1 \end{aligned}$$

Il membro di sinistra è pari, il membro di destra è dispari: assurdo. □

Un circuito euleriano in $\mathcal{G}$ esiste sempre in virtù del seguente

Teorema 2.1.2. *Il multigrafo $\mathcal{G}$ ha ogni vertice di grado pari.*

Dimostrazione. Banalmente, il multigrafo viene costruito aggiungendo gli archi del matching all'MST. Questi sono incidenti a tutti e soli i vertici di grado dispari. Inoltre, ogni vertice di grado dispari ha esattamente un arco del matching incidente. □

Il Teorema 2.1.2 assicura che $\mathcal{G}$ soddisfi la condizione necessaria e sufficiente per l'esistenza di un circuito euleriano.

Assumendo la disuguaglianza triangolare, `CHR` assicura un rapporto di approssimazione **costante** $\rho(n) = 1.5$[1], il migliore tra gli algoritmi polinomiali conosciuti.

2.2 Tour improvement

Un algoritmo di tipo "tour improvement" prende in input un tour iniziale $x_0 \in \mathcal{X}$ e, attraverso degli scambi, produce un tour valido migliore del precedente. Questa tecnica è di tipo *Local Search*: ad ogni passo vengono esaminati un insieme di "vicini" $\mathcal{N}_x \in \mathscr{P}(\mathcal{X})$ (raggiungibili con gli scambi), viene assegnato loro un punteggio tramite $w : \mathcal{X} \to \mathbb{N}$ e ne viene selezionato uno per il passo successivo, fino a raggiungere una condizione di terminazione. Più formalmente, lo schema di Local Search per il TSP può essere formulato nel modo seguente:

Algorithm 7

function LocalSearchTSP($\mathcal{X}$, $w : \mathcal{X} \to \mathbb{N}$, $\mathcal{N} : \mathcal{X} \to \mathscr{P}(\mathcal{X})$)
  $s \leftarrow$ `choose`$(\mathcal{X})$
  $\Delta \leftarrow w(s)$
  while $\Delta > 0$ **do**
    $G \leftarrow \{y \in \mathcal{N}(s) : w(y) < w(s)\}$
    $x \leftarrow$ `selectNext`(G)
    $\Delta \leftarrow w(s) - w(x)$
    $s \leftarrow x$
  end while
  return s
end function

Dove `choose` e `selectNext` dipendono dalle scelte fatte durante la stesura del modello. Una scelta molto comune per `choose` è la scelta causuale; per `selectNext` si possono utilizzare approcci di tipo *best-improvement* (dove viene scelto y che minimizzi $w(\mathcal{N}_s)$), *first-improvement* (dove viene scelto il primo y che abbia costo minore di s) o *ad-hoc*. Al termine della procedura si raggiungerà un minimo locale s dello spazio delle soluzioni $\mathcal{X}$.

Come si evince dallo schema, la parte cruciale del Local Search è la scelta di $\mathcal{N}$. Vedremo, nella prossima sottosezione, alcuni esempi per scegliere il vicinato.

2.2.1 k-OPT e 3-OPT migliorato

Senza perdita di generalità, considereremo sempre un tour $T = (0, 1, \ldots, n-1)$. Inoltre, avendo a che fare con un'entità ciclica, faremo spesso uso di aritmetica modulare; definiamo quindi

$$\oplus : V \times \mathbb{N} \to V \text{ tale che } x \oplus t \ := \ (x + t) \bmod n$$

$$\ominus : V \times \mathbb{N} \to V \text{ tale che } x \ominus t \ := \ (x - t) \bmod n$$

e definiamo le "distanze" tra gli elementi del tour come

$$d^+ : V^2 \to \mathbb{N} \text{ tale che } d^+(a,b) := argmin\{t \in \mathbb{N} : a \oplus t = b\}$$

$$d^- : V^2 \to \mathbb{N} \text{ tale che } d^-(a,b) := argmin\{t \in \mathbb{N} : a \ominus t = b\}$$

$$d : V^2 \to \mathbb{N} \text{ tale che } d(a,b) := min\{d^+(a,b), d^-(a,b)\}$$

Definizione 2.2.1 (**Mossa k-OPT**). *Sia $k \in \mathbb{N}$ e T un tour. Una mossa k-OPT consiste nella rimozione di un insieme R di k archi di T e l'inserimento di un insieme I di k archi in modo tale che $T' := (T \setminus R) \cup I$ sia ancora un tour valido. Scriveremo quindi $T \xrightarrow[k]{I,R} T'$ e identificheremo la mossa con la tripla (R, I, k).*

Definizione 2.2.2 (**Selezione**). *Una **selezione** per una mossa k-OPT è una k-pla $S = (i_1, i_2, \ldots, i_k)$ che identifica $R(S) = \big\{\{i_j, i_j \oplus 1\} : j \in \{1, 2, \ldots, k\}\big\}$. Diremo che la selezione è **completa** se $d(i_j, i_h) \geq 2$ per ogni $j \neq h$.*

Definizione 2.2.3 (**Reinserimento**). *Sia data una selezione S, diremo che $I \subseteq E$ con $|I| = k$ è un **reinserimento** per S se e solo se $(R(S), I, k)$ è una mossa k-OPT. Diremo inoltre che I è **pura** se $I \cap R(S) = \varnothing$, **degenere** altrimenti.*

Un'euristica k-OPT è una Local Search dove, per ogni $x \in \mathcal{X}$, definiamo

$$\mathcal{N}_k(x) = \Big\{y : \exists (I, R)\Big(x \xrightarrow[k]{I,R} y\Big)\Big\} \setminus \{x\}$$

Sebbene la ricerca della mossa k-OPT *best-improving* abbia complessità polinomiale $\Theta(n^k)$, da un punto di vista pratico tale ricerca diventa intrattabile già per valori di $k \geq 3$. In [5] viene descritta una procedura efficiente per l'esplorazione di $\mathcal{N}_3$ nel caso in cui si considerino solo selezioni complete e reinserimenti puri (che vengono chiamate mosse "vere" e saranno le uniche considerate da qui in avanti); tale procedura sembra avere, in media, una complessità inferiore alla cubica. Prima di proseguire, diamo alcune definizioni e fissiamo la notazione. Definiamo

$$\tau^+ : V^2 \to \mathbb{Z} \text{ tale che } \tau^+(a,b) := w(a, a \oplus 1) - w(a, b \oplus 1)$$

$$\tau^- : V^2 \to \mathbb{Z} \text{ tale che } \tau^-(a,b) := w(a, a \ominus 1) - w(a, b \ominus 1)$$

$$\mathcal{S} = \{S : S \text{ è completa } \}$$

$$\mathcal{S}_{ab} = \big\{(x, y) : \exists (v_1, v_2, v_3) \in \mathcal{S} \text{ tale che } v_a = x \wedge v_b = y\big\}$$

Una mossa 3-OPT è caratterizzata dal *removal set* $R(S)$ di 3 archi che vengono rimossi dal tour e dal *reinsertion set* I di 3 archi che vengono aggiunti per costruire un nuovo tour. Vi sono però diversi modi per ristabilire un tour, data la selezione S. Innanzitutto, notiamo che una volta rimossi gli archi della selezione, il tour viene scomposto in tre segmenti che possono essere etichettati con $1, 2, 3$. Supponendo di iniziare sempre il nuovo tour percorrendo il segmento 1 in senso orario, la scelta di I determina univocamente il verso di percorrenza degli altri segmenti. In particolare, possiamo identificare uno **schema di reinserimenti** come una permutazione con segno di $\{2, 3\}$: ad esempio, lo schema $\langle +3, -2\rangle$ indica che il nuovo tour verrà costruito dalla procedura:

Algorithm 8 3-OPT with $\langle +3, -2\rangle$ scheme

```
1: function EXECUTE(T, i_1, i_2, i_3)
2:     S_1 ← T[i_3 ⊕ 1 ... i_1]
3:     S_2 ← T[i_1 ⊕ 1 ... i_2]
4:     S_3 ← T[i_2 ⊕ 1 ... i_3]
5:     S_2 ← reverse(S_2)
6:
7:     T' ← S_1 ∪ {i_1, i_2 ⊕ 1} ∪ S_3 ∪ {i_3, i_2} ∪ S_2 ∪ {i_1 ⊕ 1, i_3}
8:     return T'
9: end function
```

È semplice notare come gli schemi di reinserimenti puri siano solo: $\langle +3, +2\rangle, \langle -2, -3\rangle, \langle +3, -2\rangle, \langle -3, +2\rangle$.

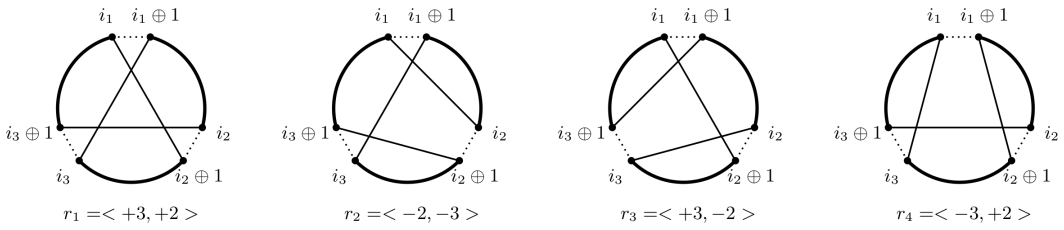


Figura 2.1: Una rappresentazione grafica degli schemi di reinserimenti. *Fonte immagine [5].*

Consideriamo una selezione $S = (i_1, i_2, i_3)$, uno schema di reinserimenti r e i rispettivi insiemi $R(S)$ e $I(r)$. Allora, il guadagno della mossa $(R(S), I(r), 3)$ è

$$\Delta(R, I) = w(R) - w(I)$$

In particolare, è possibile scomporre Δ come somma di tre funzioni di due parametri ciascuna:

$$\Delta(i_1, i_2, i_3) = f_1(i_1, i_2) + f_2(i_2, i_3) + f_3(i_1, i_3)$$

Più precisamente, manipolando Δ per ogni schema di reinserimenti, si ottengono:

$\langle +3, +2\rangle$:	$f_1(x,y) = \tau^+(x,y)$	$f_2(x,y) = \tau^+(x,y)$	$f_3(x,y) = \tau^+(y,x)$
$\langle -2, -3\rangle$:	$f_1(x,y) = \tau^+(x, y \ominus 1)$	$f_2(x,y) = \tau^-(x \oplus 1, y \oplus 2)$	$f_3(x,y) = \tau^+(y,x)$
$\langle +3, -2\rangle$:	$f_1(x,y) = \tau^+(x,y)$	$f_2(x,y) = \tau^+(x, y \ominus 1)$	$f_3(x,y) = \tau^-(y \oplus 1, x \oplus 2)$
$\langle -3, +2\rangle$:	$f_1(x,y) = \tau^-(x \oplus 1, y \oplus 2)$	$f_2(x,y) = \tau^+(x,y)$	$f_3(x,y) = \tau^+(y, x \ominus 1)$

Dove

$$dom(f_1) = \mathcal{S}_{12}, \qquad dom(f_2) = \mathcal{S}_{23}, \qquad dom(f_3) = \mathcal{S}_{13}$$

Fatte queste premesse, ci occuperemo ora di presentare l'idea descritta in [5]. L'assunzione che viene fatta è quella di supporre l'esistenza di un oracolo che, date due etichette (a, b), restituisca in tempo costante una coppia $(x, y) \in \mathcal{S}_{ab}$ che sia in una mossa *best-improving*. In questo modo è necessaria una scansione di costo lineare per trovare il "terzo" vertice della selezione che massimizzi Δ. L'obbiettivo è ora quello di simulare in modo efficiente l'oracolo: in particolare vogliamo restituire coppie che appartengano "molto probabilmente" a una mossa *best-improving*. Per fare ciò, l'osservazione chiave è la seguente:

Osservazione 2.2.1. *Supponiamo di cercare la mossa best-improving, di avere un "campione"* $S^* = (\overline{i_1}, \overline{i_2}, \overline{i_3})$ *di costo* $V = \Delta(S^*)$ *e un candidato* $S = (i_1, i_2, i_3)$. *Allora,* $\Delta(S) > \Delta(S^*)$ *se e solo se*

$$\left(f_1(i_1, i_2) > \frac{V}{3}\right) \vee \left(f_2(i_2, i_3) > \frac{V}{3}\right) \vee \left(f_3(i_1, i_3) > \frac{V}{3}\right)$$

Da questa semplice osservazione nasce l'idea di effettuare una ricerca in tre fasi: ad ogni fase $j \in \{1, 2, 3\}$ vengono enumerate le coppie che soddisfano il j-esimo disgiunto, dalla "più promettente" alla "meno promettente", terminando la fase appena la condizione non è più soddisfatta. La ricerca "ordinata" viene effettuata attraverso delle *MaxHeap* e il valore V viene aggiornato ogniqualvolta venga trovato un nuovo "campione". Ad ogni passo, viene selezionata la coppia in cima alla heap e tramite una scansione lineare si ricerca il terzo indice. Grazie alla combinazione tra l'organizzazione tramite heap e l'aggiornamento online di V si dimostra empiricamente che, in media, le selezioni considerate sono molte meno di $\Theta(n^3)$. La procedura può essere descritta in questo modo:

Algorithm 9

```
1: function MODIFIED3OPT(T, w)
2:     S* ← ∅,   r* ← ∅,   V ← 0
3:     for each r ∈ Schemes do
4:         for j ← 1, j ≤ 3, j ← j + 1 do
5:             H ← makeHeapWithBound(V, j)                ▷ Only pairs with gain > V/3
6:             while |H| > 0 ∧ topPriority(H) > V/3 do
7:                 {x, y} ← popElement(H)
8:                 z ← getZ(x, y, j)
9:                 gain ← getGain(x, y, z, j)
10:                if V < gain then          ▷ Update gain information, best selection and best scheme
11:                    V ← gain
12:                    S* ← {x, y, z}
13:                    r* ← r
14:                end if
15:            end while
16:        end for
17:    end for
18:    apply(T, S*, r*)
19: end function
```

2.2.2 Lin-Kernighan

Abbiamo considerato, nella sezione precedente, un approccio di tipo k-OPT: ad ogni passo, un tour viene migliorato applicando k rimozioni e k reinserimenti (possibilmente non disgiunti). Gli algoritmi di tipo k-OPT si basano sul concetto di:

Definizione 2.2.4. ***k-ottimalità*** *Un tour viene definito k-ottimo se è impossibile ottenere un tour di costo minore applicando una mossa k-OPT.*

Dalla definizione è evidente che un tour k-ottimo sia anche k'-ottimo per $1 \leq k' \leq k$ e che un tour è ottimo se e solo se è n-ottimo. Da queste osservazioni si evince che, generalmente, più k è elevato e migliore sarà la precisione dell'algoritmo. Sfortunatamente, il numero di mosse k-OPT esplode al crescere di k ed è quindi necessario scegliere un k non troppo elevato. Un ulteriore svantaggio è il fatto di dover specificare in anticipo il valore di k anche se non è noto a priori il migliore compromesso tra il tempo di esecuzione e la bontà dell'approssimazione per un fissato k su un preciso grafo.

L'idea di Lin e Kernighan è stata quella di effettuare "online" la scelta di k: ad ogni step, l'algoritmo esamina una mossa k-OPT per valori crescenti di k finché non viene raggiunta una condizione di terminazione. Più precisamente, ad ogni step l'algoritmo cerca di trovare due insiemi di archi $X = \{x_1, \ldots, x_k\}$ e $Y = \{y_1, \ldots, y_k\}$ tali che il tour ottenuto eliminando gli archi di X e aggiungendo gli archi di Y sia ancora valido. I due insiemi vengono costruiti elemento per elemento. Inizialmente X e Y sono vuoti; al passo i vengono aggiunti x_i e y_i rispettivamente a X e Y.

Prima di proseguire dimostriamo questo semplice fatto:

Teorema 2.2.1. *Sia $(a_1, a_2, \ldots, a_m)$ una sequenza di interi tale che $a_1 + a_2 + \cdots + a_m > 0$. Allora, esiste una permutazione ciclica σ tale che, per ogni $1 \leq i \leq m$, $a_{\sigma(1)} + \cdots + a_{\sigma(i)} > 0$.*

Dimostrazione. (Per induzione su $m \geq 1$)
Base: se $m = 1$, σ è la permutazione identica id_1.
Step: sia $m > 1$ e si considerino i seguenti casi:

- se $a_1 + \cdots + a_{m-1} > 0$ allora, per ipotesi induttiva, esiste una permutazione ciclica τ tale che $a_{\tau(1)} + \cdots + a_{\tau(i)} > 0$ per ogni $1 \leq i \leq m-1$. Ma allora $\sigma = \tau \circ id_m$.
- altrimenti, sappiamo che $|a_1 + \cdots + a_{m-1}| < a_m$, sia quindi $\tau = (2, 3, \ldots, m, 1)$, ovvero la sequenza $(a_m, a_1, a_2, \ldots, a_{m-1})$. Ma allora tale sequenza ricade nel caso precedente, per cui esiste una permutazione σ' che soddisfa il teorema. Da cui $\sigma = \sigma' \circ \tau$.

□

Per rendere efficiente la costruzione di X e Y, gli archi che vengono scelti devono rispettare alcuni criteri:

1. **Scambio sequenziale:** x_i, y_i, x_{i+1} devono condividere un vertice; più precisamente, vale $x_i = (t_{2i-1}, t_{2i})$, $y_i = (t_{2i}, t_{2i+1})$, $x_{i+1} = (t_{2i+1}, t_{2i+2})$ dove t_i indica l'i-esimo vertice del tour dopo che è stato effettuato lo scambio X, Y.

2. **Fattibilità:** viene richiesto che $x_i = (t_{2i-1}, t_{2i})$ venga scelto in modo tale che se viene scelto $y_i = (t_{2i}, t_1)$, il tour risultante sia valido. In questo modo viene garantita la possibilità di chiudere il tour in ogni momento, riducendo così il tempo di esecuzione e semplificando il codice.

3. **Guadagno positivo:** detti $g_i = w(x_i) - w(y_i)$ e $G_i = \sum_{i=1}^{i} g_i$, viene richiesto $G_i > 0$ per ogni i. Questo criterio pone una condizione di terminazione molto efficace nella pratica e, sebbene sembri troppo restrittivo, non lo è affatto in virtù del Teorema 2.2.1.

4. **Disgiuntività:** viene richiesto che, ad ogni passo, $X \cap Y \neq \varnothing$; il criterio semplifica ulteriormente il codice e fornisce un'altra condizione di terminazione.

L'algoritmo (semplificato) viene quindi schematizzato come segue:

Algorithm 10

1. Si generi un tour T casualmente.
2. Si inizializzi $i := 1$. Si Scelga t_1.
3. Si scelga $x_1 = (t_1, t_2) \in T$.
4. Si scelga $y_1 = (t_2, t_3) \notin T$ tale che $G_1 > 0$. Se ciò non è possibile, si proceda al passo 12.
5. Sia $i := i + 1$.
6. Si scelga $x_i = (t_{2i-1}, t_{2i}) \in T$ tale che:
   (a) Se t_{2i} viene connesso a t_1, il risultato deve essere un tour valido T';
   (b) $x_i \neq y_s$ per ogni $s < i$;
   Se $w(T') < w(T)$ si salvi $T := T'$ e si ritorni al passo 2.
7. Si scelga $y_i = (t_{2i}, t_{2i+1}) \notin T$ tale che:
   (a) $G_i > 0$;
   (b) $y_i \neq x_s$ per ogni $s \leq i$;
   (c) x_{i+1} esiste;
   Se tale y_i esiste, si ritorni al passo 5.
8. Se c'è un'alternativa non provata per y_2, si imposti $i := 2$ e si ritorni al passo 7.
9. Se c'è un'alternativa non provata per x_2, si imposti $i := 2$ e si ritorni al passo 6.
10. Se c'è un'alternativa non provata per y_1, si imposti $i := 1$ e si ritorni al passo 4.
11. Se c'è un'alternativa non provata per x_1, si imposti $i := 1$ e si ritorni al passo 3.
12. Se c'è un'alternativa non provata per t_1, si ritorni al passo 2.
13. Stop.

Ai passi 6 e 7, l'algoritmo controlla se le scelte per x_i e y_i soddisfino i criteri; l'algoritmo originale, tuttavia, permette di indagare più a fondo le scelte che possono essere fatte per $i = 2$: per alcuni casi è infatti possibile non rispettare, temporaneamente, il criterio di **fattibilità** per ottenere successivamente un tour valido. Lo schema descritto è più semplice anche per quanto riguarda il passo 6: T viene rimpiazzato da T' appena quest'ultimo ha un costo inferiore; l'algoritmo originale, invece, continua ad aggiungere potenziali scambi con lo scopo di trovare un tour ancora più vantaggioso, rimpiazzando T con il tour con costo migliore trovato. Questa scelta, tuttavia, non produce tour migliori, non migliora il tempo di esecuzione e complica notevolmente il codice[2].

Per migliorare le prestazioni dell'algoritmo sono stati aggiunte altre regole con lo scopo di *limitare* e *indirizzare* la scelta dei possibili archi da rimuovere o aggiungere:

5. La ricerca di un arco $y_i = (t_{2i}, t_{2i+1})$ viene limitata ai 5 archi di costo minore adiacen ti a t_{2i}.

6. Per $i \geq 4$ non possono essere rimossi archi che appartengono a tutti i $2-5$ tour migliori.

7. La ricerca viene interrotta se il tour corrente è lo stesso di quello trovato all'interazione precedente.

8. Se vi sono più scelte per $y_i (i \geq 2)$, viene data una priorità $w(x_{i+1}) - w(y_i)$.

9. Se vi sono due alternative per x_4, viene data priorità a quella con costo maggiore.

Come ultima euristica, al termine degli step "sequenziali", vengono effettuate mosse 4-OPT non sequenziali con lo scopo di ottenere ulteriori miglioramenti.

2.3 Euristiche composite e la variante di Helsgaun dell'algoritmo Lin-Kernighan

Come ultima classe di euristiche vengono presentate le cosiddette *composite* che cercano di "combinare" i vantaggi di entrambe le altre classi. A titolo di esempio verrà descritto l'algoritmo Lin-Kernighan-Helsgaun.

In [2] viene presentata un'implementazione efficace ispirata all'algoritmo di Lin e Kernighan. Sebbene la procedura si basi sempre sull'idea di applicare mosse k-OPT determinando k "al volo", la variante presentata da Helsgaun applica alcune modifiche riguardo alla scelta dei "vicini" da esaminare: se in [6] vengono esaminati i cinque archi incidenti meno costosi (regola 5), nel nuovo algoritmo è centrale il concetto di *insieme dei candidati* che migliora notevolmente la regola 5. Inoltre, se in [6] le "mosse base" (ovvero la *profondità* con cui l'algoritmo esamina le scelte che non rispettano il criterio di **fattibilità**) sono 2/3-OPT, nel nuovo algoritmo vengono estese a mosse 5-OPT.

2.3.1 Insieme dei candidati e α-nearness

Prima di proseguire, diamo le seguenti definizioni.

Definizione 2.3.1 (**1-Tree**). *Sia G un grafo e T uno Spanning Tree su $G \setminus \{1\}$. Siano inoltre e_1, e_2 due archi di G incidenti in 1. Diremo allora che $T_1 := T \cup \{e_1, e_2\}$ è un 1-Tree per G.*
Diremo inoltre che T_1 è un **1-Tree Minimo** *(Minimum 1-Tree oppure M1T) se non esiste un altro 1-Tree di peso minore.*

Osservazione 2.3.1. *Un tour ottimo è un 1-Tree Minimo tra quelli in cui ogni vertice ha grado 2.*

Osservazione 2.3.2. *Se un 1-Tree Minimo è un tour, allora è un tour ottimo.*

È quindi possibile riformulare il TSP come:

Definizione 2.3.2 (**TSP su 1-Tree**). *Sia G un grafo pesato e completo. Determinare un 1-Tree Minimo di G tra quelli che hanno ogni vertice di grado 2.*

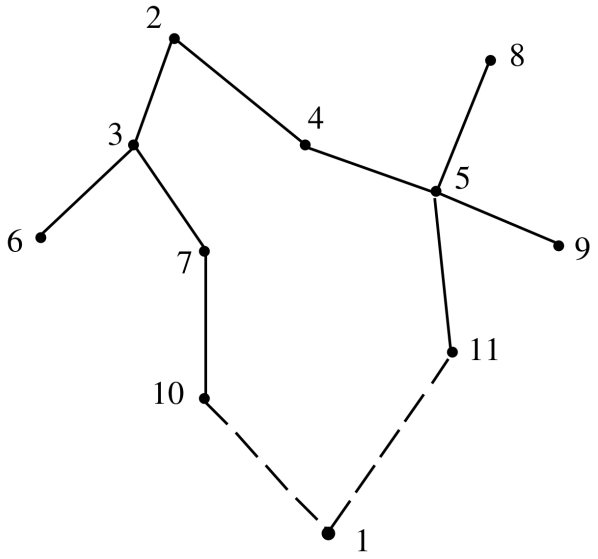


Figura 2.2: Un 1-Tree. *Fonte immagine [2]*

Come già accennato, nel nuovo algoritmo è fondamentale il ruolo di *insieme dei candidati* che estende e migliora la regola 5 di [6]. Tale regola si basa sull'idea (troppo restrittiva) che la probabilità che un arco appartenga al tour ottimo cresca al diminuire del costo dello stesso. Una misura migliore sembra essere, invece, l'appartenenza o meno a un Minimum 1-Tree: risulta infatti che, in media, un arco appartente ad un M1T abbia probabilità $0.7 \sim 0.8$ di appartenere a un tour ottimo [2]. L'idea è quindi quella di costruire l'insieme dei candidati a partire da una misura di "vicinanza" a un M1T.

Definizione 2.3.3 (α**-nearness**). *Sia T_1 un Minimum 1-Tree di peso $w(T_1)$ e sia $T_1^+(i,j)$ il Minimum 1-Tree di G che contiene l'arco (i,j). Definiamo quindi la α-nearness di (i, j) come*

$$\alpha(i,j) := w(T_1^+(i,j)) - w(T_1)$$

Esaminiamo due semplici proprietà di α.

Teorema 2.3.1.
(1) $\alpha(i,j) \geq 0$ per ogni arco (i,j).
(2) Se (i,j) appartiene a qualche M1T, allora $\alpha(i,j) = 0$.

Dimostrazione.
(1) $\alpha(i,j) = w(T_1^+(i,j)) - w(T_1) \geq w(T_1) - w(T_1) = 0.$
(2) $\alpha(i,j) = w(T_1^+(i,j)) - w(T_1) = w(T_1) - w(T_1) = 0.$ □

La prima difficoltà si presenta nel calcolo della α-nearness per ogni arco del grafo: dalla definizione sembra necessario, ogni volta, aggiungere l'arco (i,j) a un M1T ed eliminare l'ipotetico ciclo che si verrebbe a formare nel caso in cui $(i.j)$ non sia già presente in T_1. Tale operazione ha costo $\mathcal{O}(n)$ per ogni arco del grafo, per un costo totale di $\mathcal{O}(n^3)$ per computare tutte le α. L'osservazione chiave è la seguente.

Osservazione 2.3.3. *Sia $\beta(i,j)$ il costo dell'arco da rimuovere (i.e. il più costoso) se, all'inserimento di (i,j), si forma un ciclo. Ne segue che $\alpha(i,j) = w(i,j) - \beta(i,j)$. Si considerino quindi i e (j_1, j_2) tale che j_1 si trovi nel ciclo formatosi all'aggiunta dell'arco (i,j_2). Allora*

$$\beta(i,j_2) = \max\{\beta(i,j_1), w(j_1,j_2)\}$$

Da questa osservazione è possibile calcolare, in tempo ottimale $\Theta(n^2)$, le α di tutti gli archi del grafo. L'algoritmo sfrutta la programmazione dinamica e assume che i vertici siano ordinati topologicamente rispetto alla relazione "discendente" venutasi a creare durante la costruzione dell'M1T (ad esempio con Prim).

Algorithm 11

Require: $dad(i) = j \implies i < j$

```
function ComputeBeta()
    β ← new Matrix(n, n)
    for i ← 2, i < n, i ← i + 1 do
        β[i][i] ← −∞
        for j ← i + 1, j ≤ n, j ← j + 1 do
            β[i][j] ← max{β[i][dad(j)], w(j, dad(j))}
            β[j][i] ← β[i][j]
        end for
    end for
end function
```

Sebbene l'algoritmo necessiti di spazio quadratico per la matrice β, è possibile modificarlo in modo da richiedere solo spazio lineare e riuscire in ogni caso a calcolare tutti gli α [2]. Tale pseudocodice viene omesso.

Nonostante gli α siano migliori rispetto al semplice costo per definire la regola 5, è possibile migliorare ulteriormente il loro effetto applicando delle trasformazioni ai pesi degli archi del grafo.

Osservazione 2.3.4. *Ogni tour è un 1-Tree, quindi $w(T_1)$ è una limitazione inferiore al costo del tour ottimo.*

Osservazione 2.3.5. *Se il peso di tutti gli archi adiacenti a v_i viene incrementato dello stesso valore π_i, un tour che era precedentemente ottimo lo è ancora. Inoltre, se T_π è un $M1T$ sul grafo perturbato G_π, vale*

$$w(T_\pi) - 2\sum_{i=1}^{n} \pi_i \leq w(T_{OPT})$$

.

Osservazione 2.3.6. *Dalle osservazioni precedenti si ottiene che il valore $f(\pi) = w(T_\pi) - 2\sum_{i=1}^{n} \pi_i$ è ancora una limitazione inferiore al tour ottimo per G.*

L'attenzione viene quindi spostata sulla ricerca della perturbazione π che massimizzi $f(\pi)$. Tale massimizzazione è nota come limitazione di Held-Karp.

La strategia per stimare la perturbazione ottima è nota come "ottimizzazione del subgradiente" e l'idea è quella di, iterativamente, considerare il subgradiente di f ed effettuare un "passo" nel suo verso. Si può dimostrare che, in questo caso, un subgradiente per f è $v := deg - \mathbf{2}$, dove deg è il vettore dei gradi dei vertici in T_π e $\mathbf{2}$ è un vettore di soli 2. L'idea è quella di rendere più corti gli archi incidenti a vertici di grado 1, più lunghi quelli incidenti a vertici di grado maggiore di 2 e di non cambiare quelli incidenti a vertici di grado esattamente 2: in questo modo, si costringe T_π ad "alleggerire" i vertici con troppi archi adiacenti e ad "appesantire" i vertici di grado 1, nella speranza di ottenere un tour. Infatti, in virtù dell'osservazione 2.3.2, se T_π è un tour allora è un tour ottimo.

La procedura iterativa viene così schematizzata:

Algorithm 12

```
1: function ASCENT()
2:     π ← 0
3:     F ← −∞
4:     k ← 0
5:     repeat
6:         T_π ← getMin1Tree(π)
7:         f ← w(T_π)
8:         for i ← 1, i ≤ n, i ← k + 1 do
9:             f ← f − 2π[i]
10:        end for
11:        F ← max{F, f}
12:        v ← deg(T_π) − 2
13:        if ‖v‖ = 0 then
14:            return F
15:        end if
16:        t ← getStepSize(k)
17:        π ← π + tv
18:        k ← k + 1
19:    until exitCondition = True
20:    return F
21: end function
```

La convergenza del metodo alla limitazione di Held-Karp è assicurata se $\lim_{k\to\infty} t_k = 0$ e $\sum_{k=0}^{\infty} t_k = \infty$ [8]. Tuttavia, anche se la convergenza è assicurata (*ad esempio per* $t_k := \frac{t_0}{k}$), è necessario che sia veloce. Vi sono molteplici strategie che sono efficaci nella pratica, tuttavia dipendono dal grafo che si sta considerando. Helsgaun propone la seguente:

- La dimensione del passo rimane costante per ogni "periodo".
- Quando un periodo termina, viene dimezzato.
- La lunghezza del primo periodo viene impostata a $\frac{n}{2}$.
- $t_0 := 1$ e, durante il primo periodo, viene raddoppiato ad ogni step fintantoché F non viene incrementato.
- Se l'ultimo step di un periodo incrementa F, allora viene raddoppiato.
- L'algoritmo termina se $v_k = \mathbf{0}$ oppure il periodo o il passo diventano 0.

Una volta trovata la perturbazione π viene calcolato il nuovo valore di "vicinanza migliorata" che denoteremo con α_π.

2.3.2 La scelta del tour iniziale

L'algoritmo Lin-Kernighan applica molteplici scambi sullo stesso problema, utilizzando ad ogni iterazione un tour generato casualmente. Helsgaun, invece, sfrutta un'euristica di tipo "tour construction":

Algorithm 13

1. Si scelga un vertice i casualmente.
2. Si scelga un vertice j, non ancora scelto, nel modo seguente:
   (a) Se possibile, si scelga j tale che:
      i. (i, j) è un "candidato"
      ii. $\alpha(i, j) = 0$
      iii. (i, j) appartiene al migliore tour trovato fin'ora
   (b) Altrimenti, si scelga j tale che (i, j) sia un "candidato".
   (c) Altrimenti, si scelga j casualmente.
3. Sia $i := j$, se vi sono ancora vertici da scegliere, si ritorni al passo 2.

3

Implementazione e testing

A seguito di una trattazione teorica è seguita un'attività sperimentale atta a confrontare le euristiche studiate. La sperimentazione si è ispirata alla competizione "DIMACS Implementation Challenge" svoltasi nel 2000, durante la quale sono state messe a confronto diverse euristiche sviluppate e implementate da ricercatori di ogni parte del mondo [3]. Gli algoritmi sono stati testati sia su una serie di grafi generati casualmente, sia su una serie di grafi presenti in TSPLIB [9].

3.1 Scelte implementative

L'implementazione delle euristiche e dei programmi di misurazione è stata svolta nel linguaggio `C++`: la scelta è dovuta alla flessibilità con cui il linguaggio permette l'astrazione pur mantenendo elevate prestazioni. Di seguito verranno descritte, per ogni euristica, le strutture dati utilizzate e le scelte effettuate.

3.1.1 Strutture dati comuni e ridefinizioni di tipo

Le entità principali del problema sono il "Tour" e il "Grafo". Per quanto riguarda il Tour è stato deciso di utilizzare la classe `std::vector`, fornito dalla libreria standard del linguaggio. Il vantaggio principale di `std::vector` è il fatto di rappresentare un array ridimensionabile che fornisce sia l'accesso diretto ad un elemento (tramite indice), sia il "push" di un elemento in coda ad esso, entrambi con complessità $\mathcal{O}(1)$[1]. Queste performance sono quindi adatte per la gestione dei tour negli algoritmi "tour construction". Il Grafo in input, invece, è stato rappresentato come matrice dei costi.

È stato fatto uso estensivo della keyword `typedef` per rendere più espressivo il codice: in questo modo, semplici `double`, `int` o `std::vector<int>` vengono utilizzati come `Cost`, `Vertex` o `Tour`.

3.1.2 Nearest Neighbor e Nearest Insertion

È stato deciso di mantenere la semplicità concettuale delle due euristiche anche nell'implementazione: ad ogni passo vengono scansionati tutti i vertici alla ricerca del "migliore" libero, per poi aggiungerlo

[1] Più precisamente, il `push_back` ha complessità costante *ammortizzata*: `std::vector` possiede una "capacità" iniziale (un buffer interno allocato dinamicamente) e ogniqualvolta che un elemento deve essere "pushato" ma il buffer è pieno, `std::vector` alloca un nuovo buffer di dimensione doppia; è facile notare come l'inserimento di n elementi abbia quindi complessità $\mathcal{O}(n)$.

al tour parziale a seconda della politica di inserimento. Solo per quanto riguarda Nearest Insertion si è deciso di utilizzare una struttura dati di supporto, adatta agli inserimenti in posizioni arbitrarie della stessa: `std::list<>`. Anche in questo caso la classe viene offerta dalla libreria standard e rappresenta una lista concatenata. Il `typedef` utilizzato è `TourList`.

3.1.3 Christofides

L'algoritmo di Christofides richiede la costruzione di un Minimum Spanning Tree e di un Minimum Weight Matching. Per quanto riguarda l'MST si è deciso di utilizzare l'algoritmo di Prim, che è stato implementato senza l'uso di strutture dati particolari: la scelta è dovuta al fatto che, dovendo considerare solo grafi completi, l'utilizzo di Code con Priorità o algoritmi alternativi non avrebbe garantito prestazioni migliori. La costruzione del Minimum Weight Matching è stata invece semplificata: la strategia utilizzata è di tipo *greedy* nel senso che vengono scansionati gli archi in ordine crescente di peso e viene scelto il primo incidente a due vertici non ancora aggiunti al matching. Infine, la ricerca del circuito euleriano è stata svolta ricorsivamente, modificando il classico algoritmo *Depth First Search*: l'esistenza del circuito è assicurata dal Teorema 2.1.2 quindi è sufficiente visitare in profondità ogni arco non ancora visitato; dovendo effettuare inserimenti di vertici in posizioni arbitrarie del circuito che si sta costruendo, si è deciso di sfruttare `TourList`. Inoltre, avendo rappresentato gli archi non diretti del grafo come una coppia di archi diretti nei due versi opposti, sono state associate delle etichette in modo da riuscire a ricavarne una dall'altra. In questo modo è possibile "visitare" entrambi i versi di un arco che si sta per percorrere.

```
void eulerDFS(TourList *tourList, TourList::iterator it, Vertex u, GraphLabeled & graph, BoolVec & visited)
{
    size_t nOfEdges = visited.size() / 2;

    // Insert u at position it
    it = tourList->insert(it, u);

    // For each incident edge
    for(auto & edge : graph[u]) {
        if(not visited[edge.second]) {
            // Visiting an edge and its "reverse"
            visited[edge.second] = visited[(edge.second+nOfEdges) % (2*nOfEdges)] = true;
            eulerDFS(tourList, it, edge.first, graph, visited);
        }
    }
}
```

Figura 3.1: Il codice per la ricerca del circuito euleriano.

3.1.4 3-OPT

L'implementazione scelta è quella descritta in [5]. Si è mantenuta la scelta di utilizzare `std::vector` per rappresentare un Tour. Le MaxHeap, invece, sono state implementate sfruttando la classe `std::priority_queue` fornita dalla libreria standard: è stato, tuttavia, definito il tipo di dato `HeapElement` dotato di operatore di confronto, così da rendere agevole l'utilizzo della coda. L'applicazione della mossa 3-OPT trovata viene fatta seguendo la costruzione a "segmenti" definita dallo *schemda di reinserimento*: per evitare situazioni limite (segmenti che sono a cavallo tra inizio e fine del tour) è stato deciso di raddoppiare il circuito, così da "simulare", attraverso il `vector`, la ciclicità. È stata utilizzata la funzione `std::reverse()` per capovolgere i segmenti che vengono percorsi in senso antiorario: il tour viene quindi

costruito accodando i diversi segmenti con l'orientamento definito dallo schema.

```
struct HeapElement {
    Position x, y;
    Cost fValue;

    HeapElement() { }
    HeapElement(Position xx, Position yy, Cost ff)
    : x(xx), y(yy), fValue(ff) { }

    bool operator<(const HeapElement & other) const {
        return fValue < other.fValue;
    }
};
```

Figura 3.2: Il tipo di dato HeapElement.

Sono state sviluppate tre varianti di 3-OPT:

`random3opt:` il tour iniziale viene generato casualmente.

`christofides3opt:` il tour iniziale viene generato con l'algoritmo `CHR`.

`reiterated3opt:` inizialmente il tour viene generato con `CHR`; quando viene trovato un minimo locale, il tour viene "perturbato" scambiando casualmente alcuni vertici. La procedura quindi viene eseguita a partire da questo tour perturbato; la scelta di quanti vertici scambiare e di quante volte reiterare viene decisa parametricamente: in particolare, viene mantenuto un parametro `maxIteration` che viene incrementato ogni volta che viene prodotto un minimo locale migliore dei precedenti e la procedura termina quando vengono eseguite `maxIteration` iterazioni.

3.1.5 Lin-Kernighan-Helsgaun (`LKH`)

A causa della complessità implementativa di questo algoritmo è stato deciso di seguire l'approccio descritto in [2]: il tour viene rappresentato come una lista concatenata di elementi di tipo `LKHNode`. Come nel caso di Christofides, si è deciso di semplificare l'algoritmo rispetto a quello ufficiale: il tour iniziale non viene generato con l'euristica costruttiva di Helsgaun ma con `CHR`; questa scelta viene giustificata in [2]. Un'altra scelta discordante con l'algoritmo di Helsgaun è nelle "mosse base": in [2] vengono utilizzate mosse sequenziali 5-OPT; si è deciso invece di utilizzare mosse sequenziali 3-OPT per favorire la leggibilità del codice pur mantenendo la struttura originale dell'algoritmo.

La procedura, come descritta da Helsgaun, costruisce gli *insiemi dei candidati* basandosi sulla nozione di α-nearness migliorata. Questa viene calcolata da un M1T del grafo perturbato, massimizzando π con il metodo del subgradiente (nel codice, `ascent`).

```
struct LKHNode {
    int nCandidates;        // How many candidates are in this->candidateSet
    bool isLeaf;            // Check if this node is a leaf of the MST
    Vertex id;              // Vertex label
    Position pos;           // Position of this node in the tour
    int v;                  // degree - 2 (in MST)
    int lastV;              // v at the previous step of ascent
    Cost cost;              // Cost of the edge {Dad, this} in MST
    Cost specialCost;       // Used when computing M1T
    Cost currentPi, bestPi; // Perturbation of ascent
    Cost beta;              // To save beta when computing candidate sets
    LKHNode* suc;           // Successor in the tour
    LKHNode* pre;           // Predecessor in the tour
    LKHNode* dad;           // Dad in MST (with Prim)
    LKHNode* specialDad;    // Used when computing M1T
    LKHNode* mark;          // Used when computing candidate sets

    Candidate* candidateSet;// Candidates of this
};

struct Candidate {
    Cost alpha;             // Alpha-nearness to reach this candidate
    LKHNode* adj;           // Which node represents this candidate
    Candidate* suc;         // Next candidate in the set
};
```

Figura 3.3: Tipi di dato `LKHNode` e `Candidate`.

Una volta generati i candidati vengono effettuate `maxRuns` iterazioni: ad ogni iterazione viene generato un tour con `CHR` e per `maxTrials` iterazioni viene eseguito `linKernighan` su tale tour. Al termine di ogni "trial", se si è trovato un tour migliore dei precedenti, i candidati vengono modificati inserendo gli archi del tour migliorato. Al termine di ogni "run", viene salvato il migliore dei tour rispetto alle run precedenti.

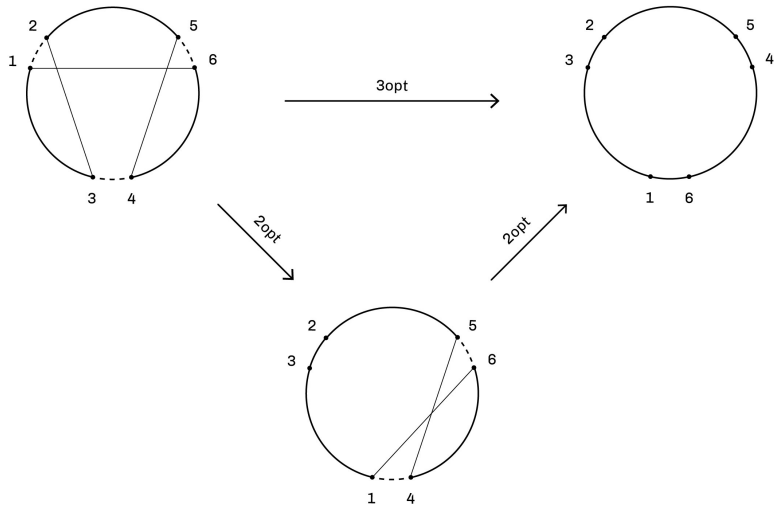
```
void LinKernighanHelsgaun()
{
    createCandidateSet();

    Cost bestCost = oo_cost;
    for(size_t run=0; run<maxRuns; ++run) {
        Cost cost = findTour();
        if(cost < bestCost) {
            recordBestTour();
            bestCost = cost;
        }

    }
}

Cost findTour()
{
    size_t trial;
    Cost bestCost = oo_cost, cost;

    for(trial=0; trial<maxTrials; ++trial) {
        chooseInitialTour();
        cost = linKernighan();
        if(cost < bestCost) {
            bestCost = cost;
            recordBetterTour();
            adjustCandidateSet();
        }
    }

    resetCandidateSet();
    return bestCost;
}
```

Figura 3.4: Lo "scheletro" della procedura.

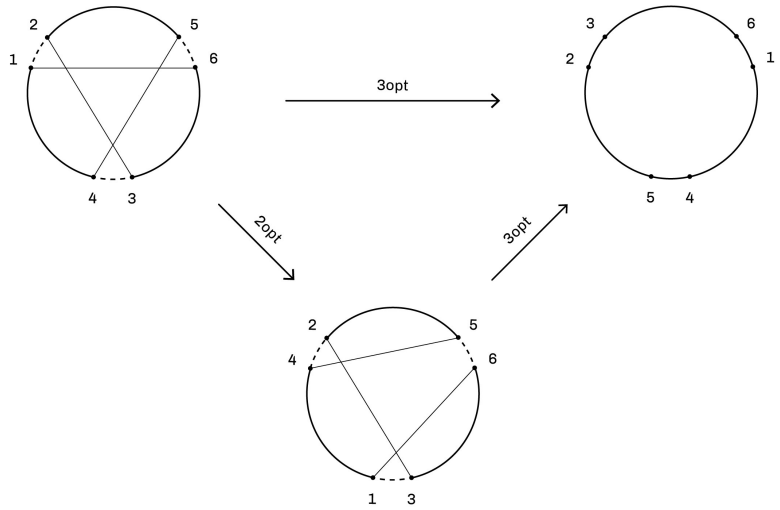
Ogni volta che viene trovata una mossa 2-OPT o 3-OPT che porti ad un tour meno pesante, questa

viene applicata. Se, invece, non esistono tali mosse, viene applicata la mossa 3-OPT più vantaggiosa e la procedura si ripete cercando di "rompere" l'ultimo arco aggiunto dalla 3-OPT della mossa precedente; viene inoltre mantenuta una pila delle mosse effettuate: se si ottiene un miglioramento del tour, tale pila viene svuotata; altrimenti, la pila viene utilizzata per ripristinare il tour precedente (quello che potremmo definire un "undo").

Sono state poi analizzate le mosse 3-OPT sequenziali e si è ricavata una procedura per scomporre tali mosse in al più tre 2-OPT sequenziali, così da semplificare ulteriormente il codice.



(a)



(b)

Figura 3.5: Vengono descritte le trasformazioni che permettono di effettuare una 3-OPT come combinazione di 2-OPT. Vi sono quattro tipi di 3-OPT sequenziali che emergono dall'algoritmo. Tre di questi sono simmetrici per rotazione al caso (a): questo può essere simulato con due 2-OPT. Notiamo poi che il quarto tipo, il caso (b), è riconducibile al caso (a) mediante un'applicazione di 2-OPT.

3.2 Soggetto della misura

Durante la fase sperimentale ci siamo focalizzati su due aspetti fondamentali di un'euristica: tempo di esecuzione e rapporto (empirico) di approssimazione. Sono stati utilizzati due tipi di di campione:

- Un insieme di grafi completi di $n \in \{100, 200, 300, 500, 1000, 2000, 3000, 5000\}$ nodi, con pesi distribuiti uniformemente nell'intervallo $\left[1, n^2\right] \cap \mathbb{N}$.
- Un insieme di grafi presenti in `TSPLIB`; sono stati scelti `a280`, `pr439`, `u1060`, `vm1748`, `pr2392`.

Inoltre, per `3-OPT` con reiterazione sono stati studiati tempo di esecuzione e qualità dell'approssimazione al variare di `maxExchanges`, mentre per `LKH` sono stati studiati tempo di esecuzione e qualità dell'approssimazione al variare di `maxCandidates`. Entrambi gli studi sono stati svolti su un campione di cinque grafi generati casualmente di $n = 1000$ nodi.

La qualità dell'approssimazione è l'errore relativo del costo del tour trovato dall'euristica rispetto al costo ottimo,

$$\epsilon = \frac{C_{eur} - C_{opt}}{C_{opt}}$$

Per le istanze scelte di `TSPLIB` l'ottimo è noto; per i grafi generati casualmente si è cercato di approssimare l'ottimo con la bound di Held-Karp calcolata attraverso `ascent`. Sfortunatamente, la differenza percentuale tra le bound approssimate e i costi dei tour superavano, in alcuni casi, i 5000 punti: l'ipotesi è che la distribuzione uniforme dei costi degli archi abbia sensibilmente abbassato la bound di Held-Karp rispetto al tour ottimo. In mancanza di evidenze sperimentali che ci permettessero di sfruttare efficacemente il valore calcolato da `ascent` si è deciso di non assegnare un errore ai costi dei tour nei grafi casuali.

3.3 Modalità di misurazione del tempo di esecuzione

La macchina su cui sono stati svolti i test è un Dell XPS-13 9370 con cpu Intel Core i5-82500U, 4 core (8 thread) da 1.60GHz (3.40GHz in Turbo Boost) e 8GB di memoria Ram.

Per evitare di incorrere in risultati non affidabili la misurazione del tempo di esecuzione è stata svolta seguendo la procedura descritta in [7]. È stato definito un'intervallo di confidenza all'$(1 - \alpha)$ per $\alpha = 0.05$ e, per ogni dimensione n, è stato generato un campione di cinque grafi di n nodi: per quanto riguarda i grafi di `TSPLIB` tale campione è stato costruito con cinque copie dello stesso grafo.

3.4 Randomness

La casualità è uno strumento fondamentale nell'Informatica: essa permette di ottenere algoritmi più efficienti, strutture dati più robuste e simulazioni più realistiche. Ognuna delle euristiche descritte utilizza, in misura diversa, la *randomness*. Non solo, anche la procedura di misurazione descritta precedentemente utilizza la randomness per la generazione dei campioni. È quindi necessario un algoritmo che approssimi sufficientemente bene la casualità: a differenza del linguaggio `C` e altri, il `C++` mette a disposizione la libreria `<random>` per gestire distribuzioni e generatori di numeri (pseudo)casuali. Tra i generatori lineari congruenziali è presente un'implementazione del prng Park-Miller (che viene utilizzato

anche in [7]). È stato quindi scelto tale generatore per effettuare ogni operazione che coinvolgesse la randomness.

3.5 Risultati e confronti

3.5.1 Tabelle dei risultati

Vengono di seguito presentati i risultati degli esperimenti effettuati. I tempi sono espressi in secondi, "Errore (%)" è $\epsilon \cdot 100$, "Diff T (%)" e "Diff C (%)" sono le differenze percentuali rispetto al valore di riferimenti: nel caso di `RL3O` è di 10 scambi, nel caso di `LKH` è di 30 candidati. In questa sottosezione ci limitiamo a fornire le tabelle dei dati; questi verranno confrontati e commentati nella successiva.

Grafo	Tempo	Costo
rand_100	0.002	37100
rand_200	0.003	215400
rand_300	0.005	494100
rand_500	0.012	1511000
rand_1000	0.047	6964000
rand_2000	0.188	32077995
rand_3000	0.439	75575990
rand_5000	1.175	204449976

Su grafi generati casualmente.

Grafo	Tempo	Costo	Errore (%)
a280	0.004	3410	32
pr439	0.009	134706	26
u1060	0.053	294537	31
vm1748	0.139	414024	23
pr2392	0.231	484614	28

Su grafi di `TSPLIB`.

Figura 3.6: Risultati `NN`.

Grafo	Tempo	Costo
rand_100	0.0014	64000
rand_200	0.0061	362200
rand_300	0.0110	1082100
rand_500	0.0250	3638500
rand_1000	0.1913	20945000
rand_2000	0.8406	116840000
rand_3000	1.9702	326238000
rand_5000	5.7974	1157599975

Su grafi generati casualmente.

Grafo	Tempo	Costo	Errore (%)
a280	0.007	3024	17
pr439	0.019	131497	23
u1060	0.107	277467	24
vm1748	0.503	408179	21
pr2392	0.557	472147	25

Su grafi di `TSPLIB`.

Figura 3.7: Risultati `NI`.

Grafo	Tempo	Costo
rand_100	0.002	104300
rand_200	0.008	795400
rand_300	0.017	3172500
rand_500	0.042	12952000
rand_1000	0.175	115443000
rand_2000	0.923	835615690
rand_3000	2.547	2957684425
rand_5000	7.886	13201053989

Su grafi generati casualmente.

Grafo	Tempo	Costo	Errore (%)
a280	0.007	3256	26
pr439	0.014	133596	25
u1060	0.115	266853	19
vm1748	0.299	403090	20
pr2392	0.608	448250	19

Su grafi di `TSPLIB`.

Figura 3.8: Risultati `CHR`.

Grafo	Tempo	Passi	Tempo medio passo	Costo
rand_100	0.3	68	0.005	20200
rand_200	3.5	144	0.024	77800
rand_300	12.8	237	0.054	192000
rand_500	75.7	407	0.186	556000
rand_1000	844.4	922	0.916	2651000
rand_2000	8186.1	1863	4.394	12434000
rand_3000	29437.4	2884	10.207	31245000
rand_5000	158043.7	4886	32.346	63764102

Su grafi generati casualmente.

Grafo	Tempo	Passi	Tempo medio passo	Costo	Errore (%)
a280	8	208	0.04	2735	6
pr439	37	330	0.11	115028	7
u1060	1201	869	1.38	240504	7
vm1748	4958	1484	3.34	369548	10
pr2392	14445	2095	6.89	409235	8

Su grafi di TSPLIB.

Figura 3.9: Risultati L3OR.

Grafo	Tempo	Passi	Tempo medio passo	Costo
rand_100	0.1	27	0.005	19600
rand_200	1.2	55	0.021	70000
rand_300	4.1	79	0.052	175500
rand_500	19.0	127	0.149	501000
rand_1000	247.5	290	0.853	2357000
rand_2000	2614.1	584	4.476	10214000
rand_3000	7926.3	892	8.886	25479000
rand_5000	40948.9	1508	27.154	76525000

Su grafi generati casualmente.

Grafo	Tempo	Passi	Tempo medio passo	Costo	Errore (%)
a280	1.1	35	0.03	2755	7
pr439	4.5	50	0.09	112478	5
u1060	97.5	138	0.71	233601	4
vm1748	434.9	188	2.31	349744	4
pr2392	917.1	281	3.26	393285	4

Su grafi di TSPLIB.

Figura 3.10: Risultati L3OC.

Grafo	Tempo	Iterazioni	Tempo medio iterazione	Costo	Migliori
rand_100	2.7	24	0.11	18400	4
rand_200	16.7	21	0.79	69400	1
rand_300	54.5	23	2.37	167400	3
rand_500	178.9	22	8.13	474500	2
rand_1000	1369.0	23	59.52	2365000	3
rand_2000	6196.6	21	295.08	9992000	1
rand_3000	17527.3	22	796.69	25833000	2
rand_5000	60088.7	23	2612.55	74329000	3

Su grafi generati casualmente.

Grafo	Tempo	Iterazioni	Tempo medio iterazione	Costo	Migliori	Errore (%)
a280	19.2	24	0.8	2650	4	2.7
pr439	131.9	33	4.0	108837	13	1.5
u1060	923.2	25	36.9	232786	5	3.9
vm1748	3185.6	27	118.0	348560	7	3.6
pr2392	7466.1	44	169.7	389969	24	3.2

Su grafi di `TSPLIB`.

Scambi	Tempo	Iterazioni	Tempo medio iterazione	Costo	Diff T (%)	Diff C (%)
1	359	22	16.3	2256000	-67	-3.4
2	452	27	16.8	2386000	-59	2.1
4	518	22	23.5	2266000	-53	-3.0
6	733	25	29.3	2312000	-33	-1.0
8	953	21	45.4	2329000	-13	-0.3
10	1093	24	45.5	2336000	0	0.0

Su un campione di grafi causali di $n = 1000$ nodi, al variare di `maxExchanges`.

Figura 3.11: Risultati `RL3O`.

Grafo	Tempo	Costo
rand_100	4.5	28100
rand_200	20.0	119400
rand_300	44.8	331500
rand_500	100.3	1028000
rand_1000	234.7	6232000
rand_2000	1210.2	36998000
rand_3000	4316.5	116297983
rand_5000	10140.1	464524914

Su grafi generati casualmente.

Grafo	Tempo	Costo	Errore (%)
a280	11.1	2824	10
pr439	18.2	122035	14
u1060	144.7	261742	17
vm1748	351.6	387958	15
pr2392	797.4	421233	11

Su grafi di `TSPLIB`.

Candidati	Tempo	Costo	Diff T (%)	Diff C (%)
5	179	10442000	-23.6	67.6
10	123	7824000	-47.4	25.5
13	130	7289000	-44.7	17.0
15	125	7267000	-46.9	16.6
18	184	6849000	-21.8	9.9
20	182	6567001	-22.3	5.4
25	136	7053000	-42.1	13.2
30	235	6232000	0.0	0.0

Su un campione di grafi causali di $n = 1000$ nodi, al variare di `maxCandidates`.

Figura 3.12: Risultati `LKH`.

3.5.2 Tabelle e grafici dei confronti

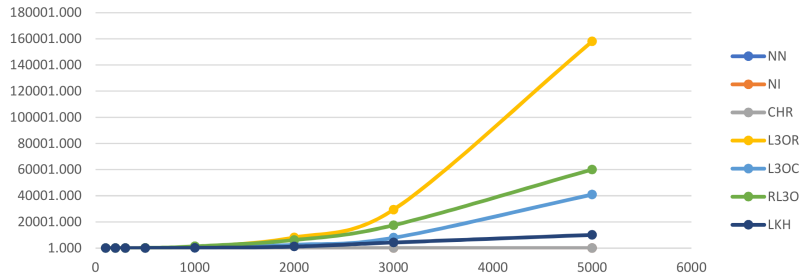
Vengono ora confrontati e commentati i risultati ottenuti: tempi e costi tra diverse euristiche, confronto nel numero di scambi in RL3O e confronto nel numero di candidati in LKH.

Grafo	NN	NI	CHR	L3OR	L3OC	RL3O	LKH
100	0.002	0.0014	0.002	0.3	0.1	2.7	4.5
200	0.003	0.0061	0.008	3.5	1.2	16.7	20.0
300	0.005	0.0110	0.017	12.8	4.1	54.5	44.8
500	0.012	0.0250	0.042	75.7	19.0	178.9	100.3
1000	0.047	0.1913	0.175	844.4	247.5	1369.0	234.7
2000	0.188	0.8406	0.923	8186.1	2614.1	6196.6	1210.2
3000	0.439	1.9702	2.547	29437.4	7926.3	17527.3	4316.5
5000	1.175	5.7974	7.886	158043.7	40948.9	60088.7	10140.1

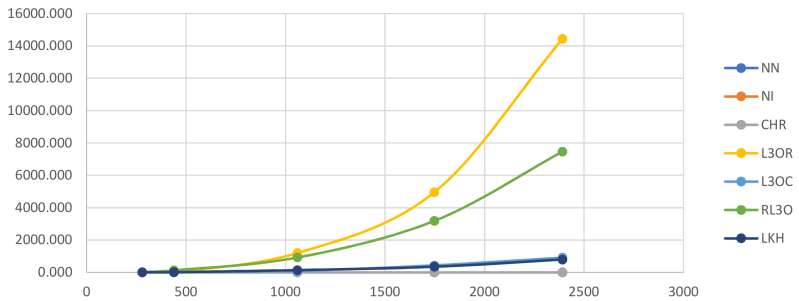
Su grafi generati casualmente.

Grafo	NN	NI	CHR	L3OR	L3OC	RL3O	LKH
280	0.004	0.007	0.007	8	1.1	19.2	11.1
439	0.009	0.019	0.014	37	4.5	131.9	18.2
1060	0.053	0.107	0.115	1201	97.5	923.2	144.7
1748	0.139	0.503	0.299	4958	434.9	3185.6	351.6
2392	0.231	0.557	0.608	14445	917.1	7466.1	797.4

Su grafi di TSPLIB.



Su grafi generati casualmente.



Su grafi di TSPLIB.

Figura 3.13: Confronto diretto dei tempi di esecuzione delle diverse euristiche.

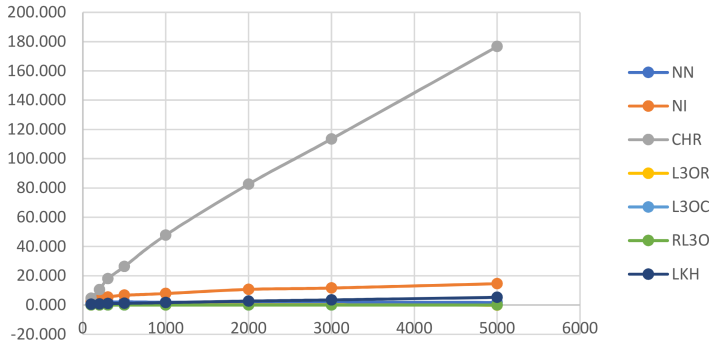
Da come si evince dai grafici, le euristiche "tour construction" sono le vincitrici per quanto riguarda la velocità di esecuzione; il più lento è LR3O con un notevole distacco (più del doppio del tempo rispetto al penultimo RL3O); LKH si dimostra sorprendemente veloce. Il trend si mantiene sia sui grafi casuali, sia sui grafi di TSPLIB.

Grafo	NN	NI	CHR	L3OR	L3OC	RL3O	LKH
100	1.016	2.478	4.668	0.098	0.065	0.000	0.527
200	2.104	4.219	10.461	0.121	0.009	0.000	0.720
300	1.952	5.464	17.952	0.147	0.048	0.000	0.980
500	2.184	6.668	26.296	0.172	0.056	0.000	1.166
1000	1.945	7.856	47.813	0.121	-0.003	0.000	1.635
2000	2.210	10.693	82.628	0.244	0.022	0.000	2.703
3000	1.926	11.629	113.492	0.209	-0.014	0.000	3.502
5000	1.751	14.574	176.603	-0.142	0.030	0.000	5.250

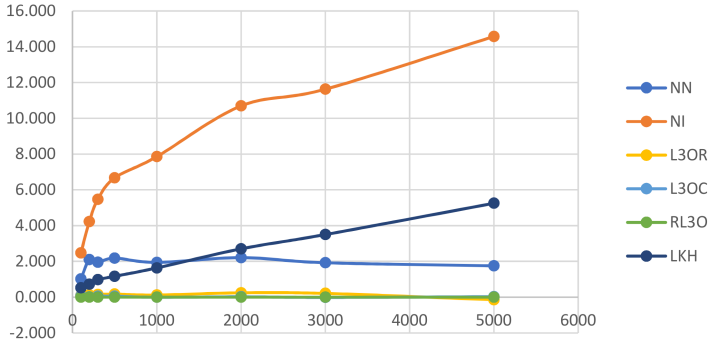
Su grafi generati casualmente.

Grafo	NN	NI	CHR	L3OR	L3OC	RL3O	LKH
280	0.29	0.14	0.23	0.03	0.04	0.00	0.07
439	0.24	0.21	0.23	0.06	0.03	0.00	0.12
1060	0.27	0.19	0.15	0.03	0.00	0.00	0.12
1748	0.19	0.17	0.16	0.06	0.00	0.00	0.11
2392	0.24	0.21	0.15	0.05	0.01	0.00	0.08

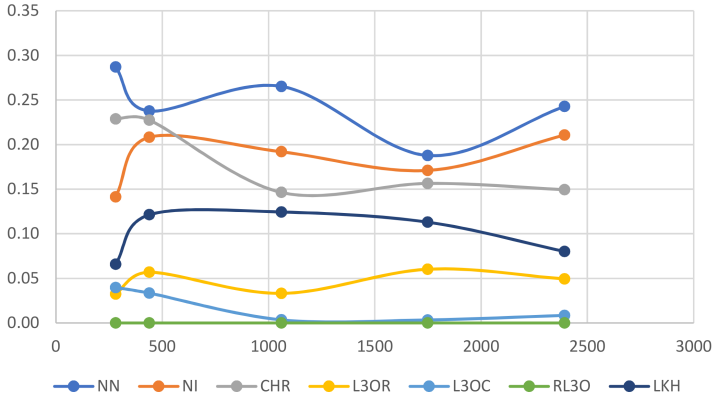
Su grafi di `TSPLIB`.



Su grafi generati casualmente, con la presenza di `CHR`.



Su grafi generati casualmente, senza `CHR` (che ha trovato tour molto più costosi).



Su grafi di `TSPLIB`.

Figura 3.14: Confronto diretto dei costi dei tour trovati dalle diverse euristiche.

Situazione diametralmente opposta per quanto riguarda i costi: `L3OR` e `RL3O` si dimostrano i più precisi; `CHR` decisamente impreciso per quanto riguarda i grafi casuali, tanto da rendere necessaria la visualizzazione grafica dei dati senza la sua presenza. Per quanto riguarda i grafi di `TSPLIB` la situazione è più equilibrata. I confronti sui costi, non essendo noti gli ottimi di una delle classi di grafi, sono stati fatti in differenza percentuale rispetto a `RL3O`.

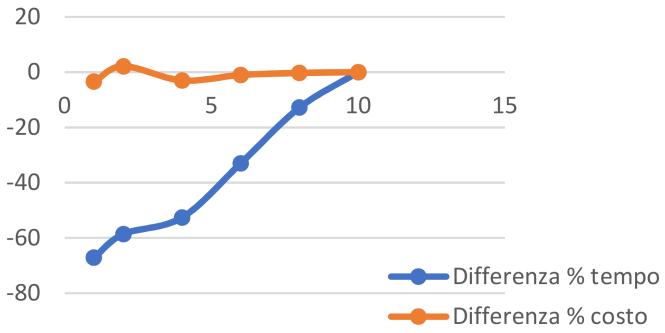


Figura 3.15: Confronto tra efficienza ed efficacia di `RL3O` al variare del numero massimo di *exchanges*.

Per quanto riguarda la scelta del numero massimo di scambi sembra non esserci una differenza sostanziale nei costi dei tour trovati quando tali scambi sono contenuti. Prevedibile, invece, il guadagno in tempo se riduciamo il numero di scambi: meno scambi significa, intuitivamente, che il tour è molto simile a prima e quindi richiede meno passi *best-improving* per riottenere il minimo locale.

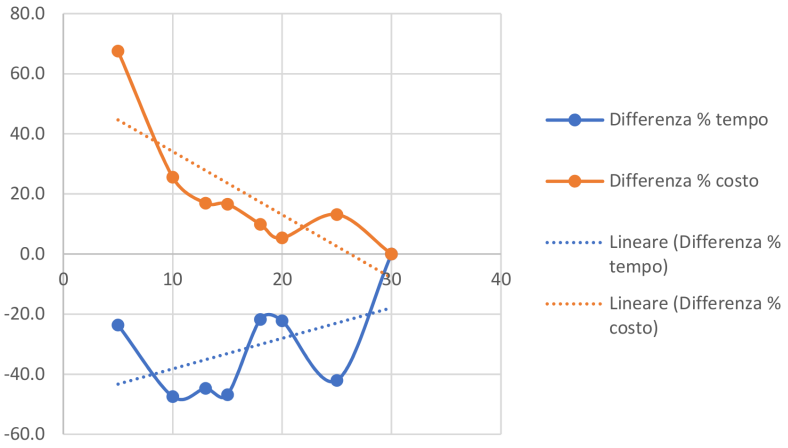


Figura 3.16: Confronto tra efficienza ed efficacia di `LKH` al variare del numero massimo di candidati per ogni *candidate set*.

Prevedibile anche il risultato del test sul numero di candidati: la tendenza è quella di trovare tour più costosi e in meno tempo se i candidati sono pochi e tour meno costosi ma in più tempo se i candidati sono di più. Questo è anche il comportamento che ci si aspetterebbe da un'euristica, ovvero il tradeoff qualità/tempo.

4

Conclusioni

È stato presentato il Commesso Viaggiatore Simmetrico come uno dei problemi fondamentali dell'Ottimizzazione Combinatoria. La *difficoltà* che richiede la ricerca della soluzione ottima e l'importanza che tale soluzione ha in svariati ambiti applicativi ci ha spinto a trattare il problema dal punto di vista degli algoritmi approssimati. Abbiamo presentato alcune delle più famose euristiche di tipo "tour construction", "tour improvement" e "composite". La famiglia "costruttiva" si compone di algoritmi che definiscono un tour valido vertice dopo vertice. La famiglia "migliorativa" comprende euristiche che analizzano un tour valido e cercano di modificarlo per ottenere un tour valido migliore. L'ultima classe si compone di algoritmi che tentano di combinare i vantaggi di entrambe. Abbiamo poi implementato gli algoritmi descritti e trattato, dal punto di vista empirico, le caratteristiche che riteniamo fondamentali in un algoritmo approssimato: velocità di esecuzione e errore rispetto all'ottimo; dai risultati sperimentali è emersa una correlazione tra queste due proprietà. Come ci si aspettava, euristiche meno elaborate impiegano meno tempo per trovare il tour "quasi"-ottimo, al costo di avere un elevato errore; viceversa, euristiche più elaborate impiegano un tempo notevolmente maggiore per trovare una soluzione, che sarà di qualità migliore. La scelta dell'euristica da utilizzare, tuttavia, dipende strettamente da quale caratteristica si voglia prediligere.

Bibliografia

[1] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. 1976.

[2] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[3] David Johnson e Lyle A. Mcgeoch. Experimental analysis of heuristics for the stsp. 2001.

[4] R. M. Karp e J. M. Steele. *Probabilistic analysis of heuristics*. 1985.

[5] Giuseppe Lancia e Marcello Dalpasso. Speeding-up the exploration of the 3-opt neighborhood for the tsp. 2018.

[6] Shen Lin e Brian W. Kernighan. An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, 21:498–516, 1973.

[7] Alberto Policriti. Appunti per il laboratorio di algoritmi e strutture dati. 2018.

[8] Boris Polyak. A general method for solving extremum problems. *Soviet Mathematics. Doklady*, 8:593–597, 1967.

[9] Gerhard Reinelt. TSPLIB–a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, 1991.

[10] Günter Rothe. Two solvable cases of the traveling salesman problem. 1988.