Implementazione procedura GRASP per istanze di TSP

Filippo Vajana

AA. 2018-2019

1 Introduzione

Il problema del commesso viaggiatore è il più semplice fra i problemi di *routing* e di *scheduling*. Esso viene spesso indicato con il suo nome inglese, Travelling Salesman Problem, da cui la sigla TSP. Il TSP è uno dei casi di studio tipici dell'informatica teorica e della teoria della complessità computazionale. Il nome nasce dalla sua più tipica rappresentazione: dato un insieme di città, e note le distanze tra ciascuna coppia di esse, trovare il tragitto di minima percorrenza che un commesso viaggiatore deve seguire per visitare tutte le città una ed una sola volta.

Modello matematico. Al problema del TSP è associabile un grafo $G = (V, A)$, in cui V è l'insieme degli n nodi e A è l'insieme degli archi che gli uniscono. Si indica con c_{ij} il costo dell'arco per andare dal nodo i al nodo j. Il TSP è *simmetrico* se $c_{ij} = c_{ji} \forall (i, j)$, altrimenti si dice asimmetrico.
A seconda che il problema sia simmetrico o asimmetrico, esso può essere descritto con modelli matematici differenti. Detta x_{ij} la generica variabile binaria tale che $x_{ij} = 1$ se l'arco (i, j) appartiene al circuito e $x_{ij} = 0$ altrimenti, nel caso più generale di problema asimmetrico, una possibile formulazione matematica del problema è:

$$\mathrm{z} = \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$$

s.t.

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, \ldots, n$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad i = 1, \ldots, n$$

$$\sum_{i \in Q} \sum_{j \in V \setminus Q} x_{ij} \geq 1 \qquad \forall Q \subset V, |Q| \geq 1$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, \ldots, n$$

Complessità computazionale. Il TSP è un problema di classe NP-difficile, ne deriva che non siano disponibili algoritmi per la sua risoluzione efficiente. Il

metodo più generale per una risoluzione esatta è l'enumerazione totale, ovvero il calcolo di tutti i possibili cammini sul grafo e la successiva scelta di quello migliore. Tuttavia, la complessità dell'operazione la rende impraticabile per grafi relativi a problemi reali: in un grafo di n nodi, bisognerà calcolare, nel caso peggiore in cui ogni nodo è connesso con tutti gli altri, $n!$ possibili cammini, il che implica una complessità esponenziale.
Limitandosi a particolari istanze è però possibile ottenere soluzioni ottime, in tempi ragionevoli, applicando algoritmi esatti:

- Algoritmi *branch and bound* per grafi con 50-60 nodi.
- Algoritmi a raffinamenti successivi mutuati dalla *programmazione lineare* avendo 150-200 nodi.
- Algoritmi *branch and cut* per grafi fino a 5000 nodi.
- Il metodo *cutting plane*, risalente al 1954, è stato usato con successo su una istanza da 15000 nodi.

1.1 GRASP

GRASP è l'acronimo di *Greedy Randomized Adaptive Search Procedure*, ed identifica una variante più raffinata dell'euristica costruttiva casualizzata Semi-Greedy. Le componenti principali che compongono questo metodo sono espresse chiaramente all'interno del nome:

- *Greedy* indica che viene sfruttata una euristica costruttiva come base di partenza.
- *Randomized* poichè l'euristica base è progettata per eseguire passi casuali.
- *Adaptive* indica che l'euristica usa un criterio di scelta adattivo $\varphi_A(i, x)$, che dipende anche da x.
- *Search* indica che le soluzioni generate dall'euristica costruttiva di base vengono migliorate mediante l'applicazione di una seconda euristica di scambio.

```
1:  procedure GRASP
2:      x* ← ∅
3:      f* ← +∞
    ▷ Euristica Costruttiva
4:      for I = 0; I ≤ ℓ do
5:          while Ext_A(x) ≠ ∅ do
6:              φ ← [φ_A(i, x), ∀i ∈ Ext(x)]
7:              L ← Sort(φ)
8:              π ← Restricted Candidate List(L)
9:              i ← Random Extraction(Ext_A(x), π)
10:             x ← x ∪ {i}
    ▷ Euristica di Scambio
11:     x' ← Search(x)

12:     if x' ∈ X and f(x') ≤ f* then
13:         x* ← x'
14:         f* ← f(x')

    return(x*)
```

Funzione probabilità. Sebbene l'algoritmo esegua passi casuali, tali passi devono essere coerenti con i valori forniti dal criterio di selezione $\varphi_A(i,x)$. In particolare la funzione di assegnazione della probabilità $\pi_A(i,x)$ deve rispettare la monotonia della funzione di selezione:

$$\varphi_A(i,x) \leq \varphi_A(j,x) \iff \pi_A(i,x) \geq \pi_A(j,x)$$

Le principali strategie per l'assegnamento della probabilità di estrazione π ai nodi in $Ext_A(x)$ sono:

- Probabilità uniforme.
- Heuristic-Biased Stochastic Sampling.
- Restricted Candidate List.

Per il progetto in questione è stata utilizzata la strategia basata su RCL con inserimento basato sul valore del nodo candidato. Il valore $\varphi_A(i,x)$ di un nodo è calcolato come il costo del suo inserimento all'interno del circuito corrente rappresentato dalla soluzione parziale x. La RCL contiene tutti i nodi candidati i con valore:

$$\varphi_A(i,x) \in [\varphi_{min}, \varphi_{min} + \mu(\varphi_{max} - \varphi_{min})]$$

dove $\mu \in [0,1]$ è un parametro libero:

- con $\mu = 0$ si *"annulla"* il concetto di RCL e si ottiene l'euristica base.
- con $\mu = 1$ vengono aggiunti indiscriminatamente tutti i nodi e si degenera in una *random walk*.

2 Pseudocodice

Algorithm 1

1: **procedure** MAIN($distanceMatrix, runs$)
2: $results \leftarrow \emptyset$
3: **for** $i = 1; i \leq runs$ **do**
4: $(circuit, cost) \leftarrow$ `Run`$(distanceMatrix)$
5: $results \leftarrow results \cup \{(circuit, cost)\}$
6: **Return** min$(results)$

Main. La procedura `Main()` serve come entrypoint per l'esecuzione dell'algoritmo GRASP sulla istanza corrente. Riceve come input la matrice delle distanze tra i nodi del grafo $G = (V, A)$ e il numero di volte che l'algoritmo dovrà essere eseguito. Siccome le istanze di prova sono composte da grafi totalmente connessi la matrice delle distanze contiene tutte le informazioni necessarie all'esecuzione dell'algoritmo.

Algorithm 2

1: **procedure** RUN($distanceMatrix$)
2: $dm \leftarrow distanceMatrix$
3: $rowsCount \leftarrow |V|$
4: $circuitList \leftarrow \emptyset$

▷ **Seleziono nodo iniziale**
5: $startNode \leftarrow$ `RandomUniform`$([0, |V|))$
6: $circuitList.AddFirst(startNode)$

▷ **Inizializzo frontiera**
7: $frontierList \leftarrow [0, \ldots, |V| - 1]$
8: $frontierList \leftarrow frontierList.Remove(startNode)$

▷ **Applico euristica costruttiva**
9: **for** $i = 1; i \leq |V|$ **do**
10: $(cNode, fNode) \leftarrow$ `SelectNextNode`$(dm, circuitList, frontierList)$
11: $circuitList \leftarrow circuitList.AddAfter(cNode, fNode)$
12: $frontierList \leftarrow frontierList.Remove(fNode)$

▷ **Applico euritica di scambio 2-opt**
13: $circuitList \leftarrow$ `Opt2Swap(circuitList)`

14: **return** $(circuitList,$ `Cost`$(dm, circuitList))$

Run. La funzione `Run()` implementa la struttura generale del metodo GRASP. Come primo passo inizializza il circuito andando a selezionare con probabilità uniforme uno qualsiasi dei nodi del grafo G, tale scelta progettuale è stata fatta nel tentativo di aumentare la diversificazione delle soluzioni. Il secondo passo genera la frontiera dell'esplorazione inserendo in un vettore tutti i nodi V ad

eccezione di *startNode*. Successivamente viene applicata l'euristica costruttiva base che ad ogni iterazione seleziona sia il nuovo nodo da aggiungere sia la posizione nella quale inserirlo nel circuito corrente. L'ultimo passo della funzione `Run()` è rappresentato dalla applicazione della euristica di scambio *2-opt* per migliorare il circuito ottenute precedentemente.

Algorithm 3

1: **procedure** SELECTNEXTNODE($distanceMatrix, circuit, frontier, \mu$)
2: $dm \leftarrow distanceMatrix$

▷ **Costruisco vettore dei costi**
3: $costs \leftarrow \emptyset$
4: **for** $circuitNode \in circuit$ **do**
5: **for** $frontierNode \in frontier$ **do**
6: $\varphi \leftarrow$ `InsertionCost`$(circuitNode, frontierNode, dm)$
7: $costs \leftarrow costs \cup \{(circuitNode, frontierNode, \varphi)\}$

▷ **Costruisco RCL**
8: $\varphi_{min} \leftarrow \min_{\varphi}(costs)$
9: $\varphi_{max} \leftarrow \max_{\varphi}(costs)$
10: $rcl \leftarrow \emptyset$
11: **for** $candidate \in costs$ **do**
12: **if** $\varphi(candidate) \leq \varphi_{min} + \mu(\varphi_{max} - \varphi_{min})$ **then**
13: $rcl \leftarrow rcl \cup \{(candidate.circuitNode, candidate.frontierNode)\}$

▷ **Estraggo casualmente nodo da inserire**
14: $candidateIdx \leftarrow$ `RandomUniform([0,|rcl|))`

15: **return** $rcl[candidateIdx]$

SelectNextNode. La funzione `SelectNextNode()` implementa la strategia di esplorazione della frontiera del metodo GRASP. Per prima cosa calcola i costi di ogni possibile inserimento di un nodo nella frontiera di esplorazione (*frontier*) all'interno del circuito corrente (*circuit*). I costi sono poi salvati all'interno del vettore *costs*. Il costo di un inserimento è calcolato come:

$$\texttt{Costo}(cn, fn) + \texttt{Costo}(fn, cn.next) - \texttt{Costo}(cn, cn.next)$$

Successivamente utilizzando il vettore *costs* ed il valore μ fornito in input viene costruito il vettore *rcl* andando ad inserire i nodi di *frontier* sulla base del loro costo $\varphi_A(i, x)$. L'ultimo passo della funzione estrae con probabilità uniforme un nodo dalla RCL, tale nodo verrà poi inserito all'interno del circuito corrente.

3 Risultati

Dati. Le istanze TSP usate per i benchmark fanno parte di *TSPLIB*[1], una libreria che raccoglie un gran numero di istanze di problemi nell'ambito del TSP. I dettagli delle istanze utilizzate per testare l'implementazione di GRASP sono riassunti in Tabella 1.

	Nodi	Ottimo
att48	48	10628
bayg29	29	1610
bays29	29	2020
burma14	14	3323
fri26	26	937
gr21	21	2707
gr24	24	1272
pr76	76	108159
st70	70	675

Tabella 1: Dettagli istanze benchmark

Soluzioni. Per ogni istanza di test vengono effettuate 1000 esecuzioni indipendenti e per ogni esecuzione vengono registrati il circuito finale ed il relativo costo. I dati raccolti verrano poi usati per generare le statistiche mostrate più avanti.
In Tabella 2 vengono mostrate le prestazione dell'algoritmo GRASP per le singole istanze di test. Come metrica è stato scelto l'indice *Relative Solution Quality* che quantifica la qualità di una soluzione sulla base della differenza relativa tra la soluzione ottenuta e l'ottimo del problema:

$$\delta_A(I) = \frac{|f_A(I) - f_A^*(I)|}{f_A^*(I)} \geq 0$$

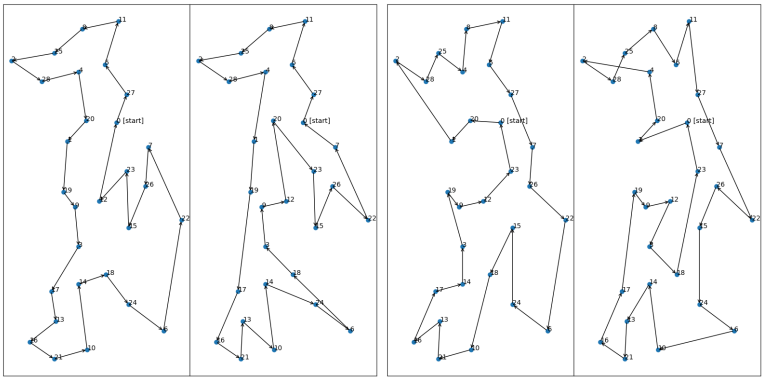
	Ottimo	$P_{0.25}$	$P_{0.50}$	$P_{0.75}$
att48	217.55	231.88	237.62	244.24
bayg29	2.55	5.53	6.55	7.89
bays29	1.98	5.45	6.58	9.41
burma14	0.87	2.68	3.91	5.15
fri26	4.38	9.82	10.25	11.02
gr21	1.88	10.82	12.63	14.59
gr24	4.48	7.47	8.49	9.69
pr76	6.71	15.68	18.48	20.14
st70	1.93	9.33	11.78	13.78

Tabella 2: Valori indice RSQ

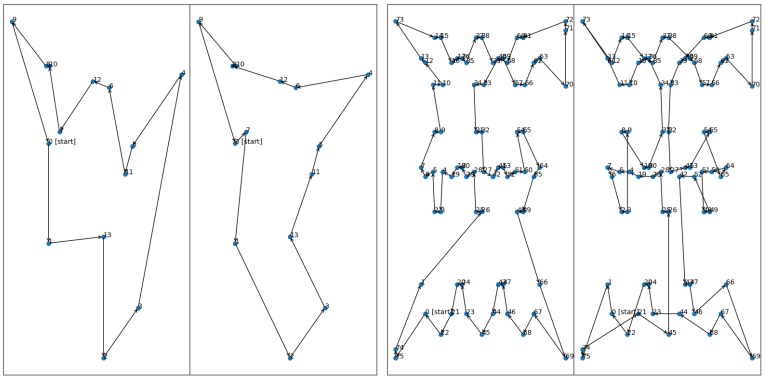
Circuiti. Al fine di mostrare chiaramente le differenze tra le differenti soluzioni ottenute per una particolare istanza, sono stati messi a confronto il circuito

[1] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

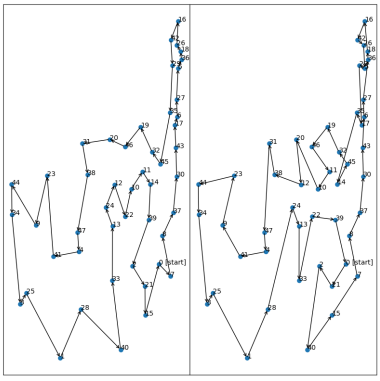
corrispondente alla soluzione migliore con quello della soluzione peggiore. Salta immediatamente all'occhio come le soluzioni migliori formino un cammino più regolare con la prevalenza di transizioni tra nodi vicini. Le soluzioni peggiori viceversa evidenziano un comportamento più frenetico con spostamenti più ampi tra un nodo e l'altro.



(a) bayg29 (b) bays29



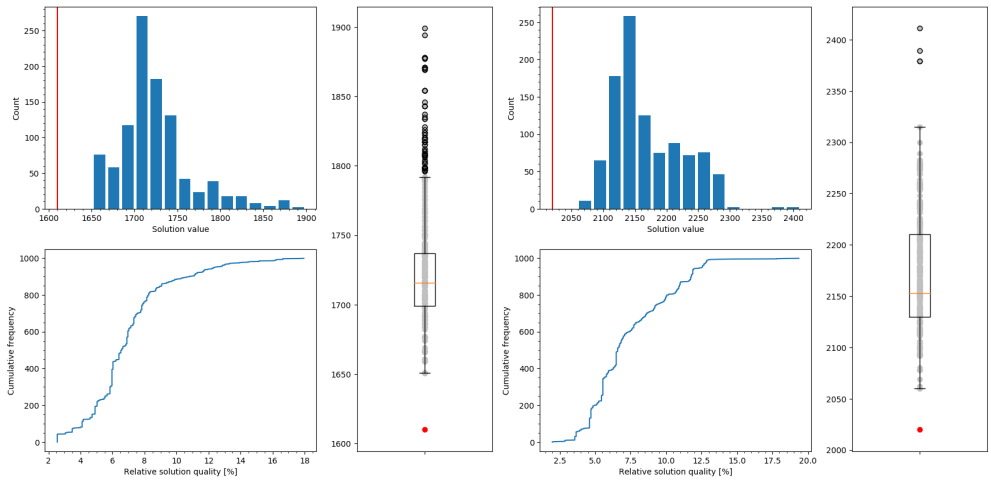
(c) burma14 (d) pr76



(e) att48

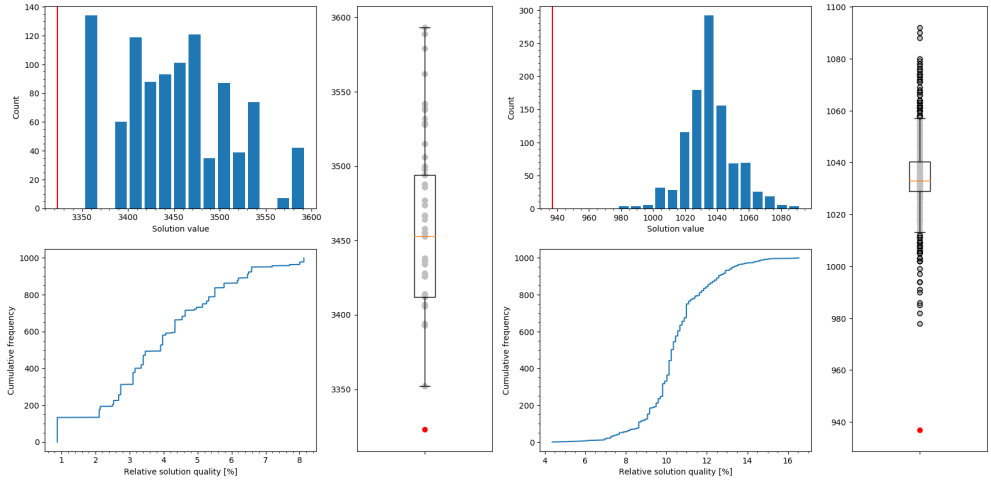
Figura 1: Soluzione migliore (*sinistra*) e soluzione peggiore (*destra*).

Distribuzione soluzioni. A seguire una raccolta di grafici che mostrano la qualità delle soluzioni trovate per le varie istanze di prova. Per ogni istanza vengono mostrati: un istogramma delle frequenze dei valori delle soluzioni, la funzione cumulativa per la qualità relativa delle soluzioni ed infine un boxplot che mostra la distribuzione dei valori delle soluzioni. In rosso è evidenziato il valore ottimo dell'istanza.



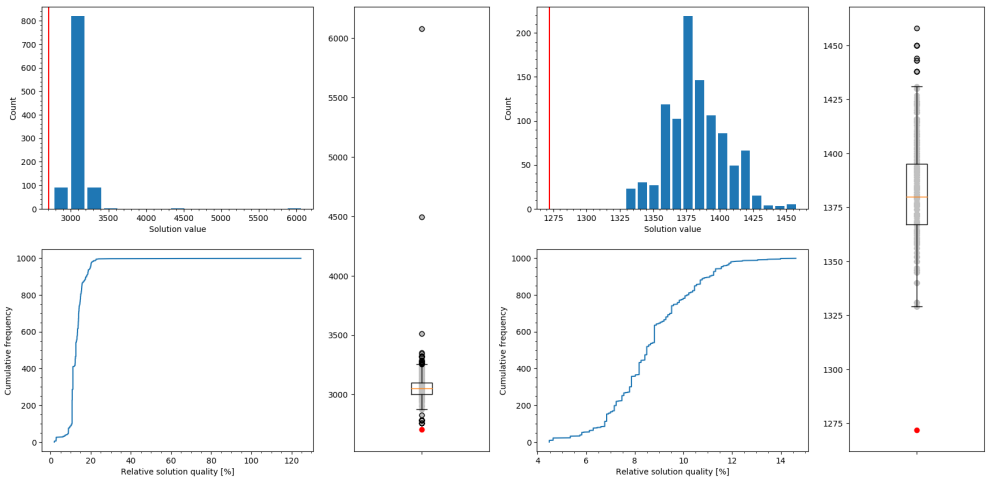
(a) bayg29

(b) bays29



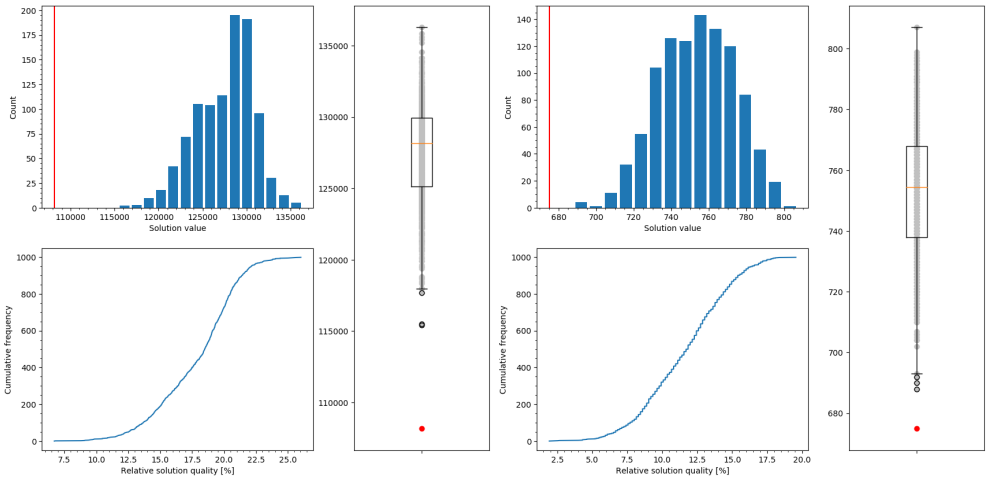
(c) burma14

(d) fri26



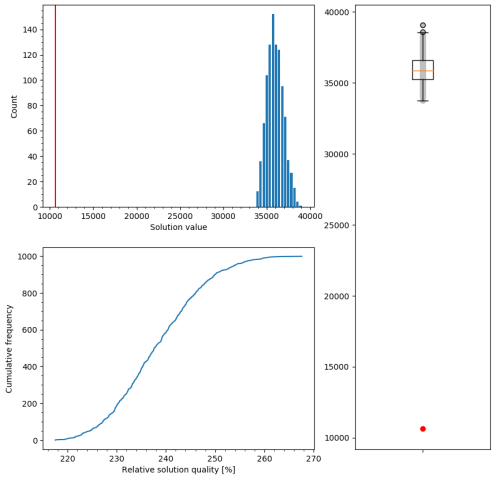
(e) gr21

(f) gr24



(g) pr76

(h) st70



(i) att48

3.1 Contributo euristica scambio 2-opt

Obiettivo di questa sezione è mostrare come l'utilizzo congiunto di euristiche costruttive e di scambio all'interno del metodo GRASP porti ad ottenere risultati migliori rispetto ad approcci più semplici basati sull'impiego della sola euristica costruttiva. Per simulare un approccio esclusivamente costruttivo è stata modificata l'implementazione di GRASP descritta in precedenza in modo da escludere la fase di applicazione dell'euristica 2-opt. I dati sperimentali sono stati raccolti sulle medesime istanze di benchmark usate per l'analisi di GRASP. La Figura 1 mette a confronto la distribuzione statistica delle soluzioni ottenute con i due metodi, mentre la Tabella 3 sottostante usa l'indice RSQ nel caso medio per evidenziare la migliore qualità delle soluzioni ottenute da GRASP.

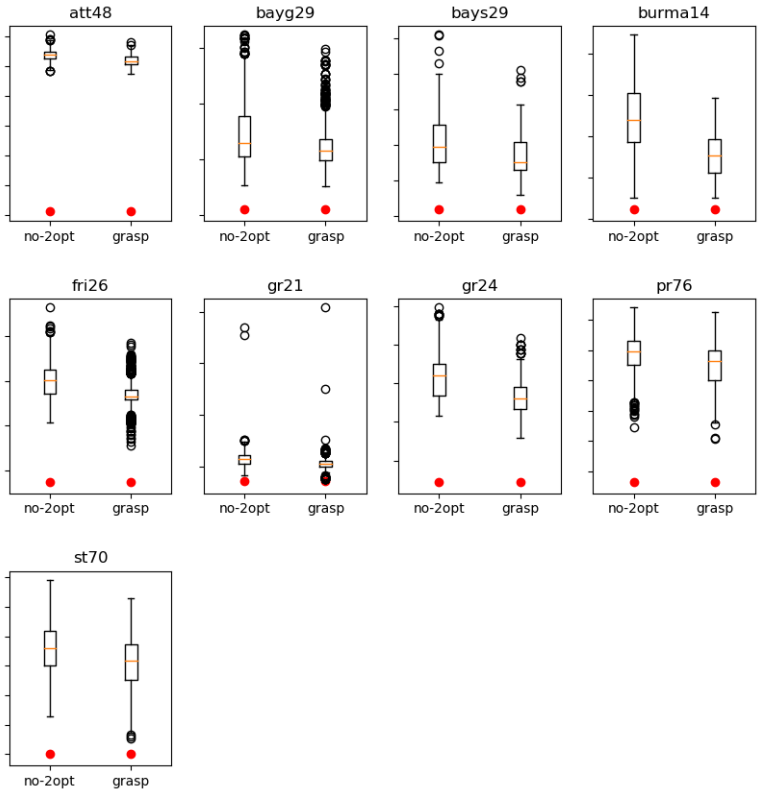


Figura 1: Valore delle soluzioni. No-2Opt (*sinistra*) e GRASP (*destra*)

Il principale vantaggio che l'utilizzo di 2-Opt comporta è un deciso miglioramento nella qualità media delle soluzioni ottenute: i boxplot di Figura 1 mostrano come le soluzioni prodotte da GRASP risultino tendenzialmente più concentrate intorno al valore medio con una minor influenza degli outliers.
Osservando le soluzioni migliori ottenute dai due metodi si osserva come l'applicazione della euristica costruttiva solo in alcuni casi migliori quanto ottenuto dal metodo esclusivamente costruttivo. Questo sembrerebbe suggerire come il criterio di scelta utilizzato all'interno della euristica costruttiva, ovvero il co-

sto di inserimento nel circuito corrente, sia il un fattore limitante per quanto riguarda la ricerca della soluzione ottima.

	No-2Opt	GRASP	ΔRSQ
att48	247.06	238.31	-3.54%
bayg29	8.25	6.97	-15.53%
bays29	9.23	7.41	-19.67%
burma14	6.71	3.84	-42.87%
fri26	12.2	10.48	-14.12%
gr21	16.52	13.09	-20.78%
gr24	10.72	8.64	-19.42%
pr76	19.68	17.9	-9.08%
st70	13.25	11.6	-12.48%

Tabella 3: Confronto valori medi di RSQ per No-2Opt e GRASP.