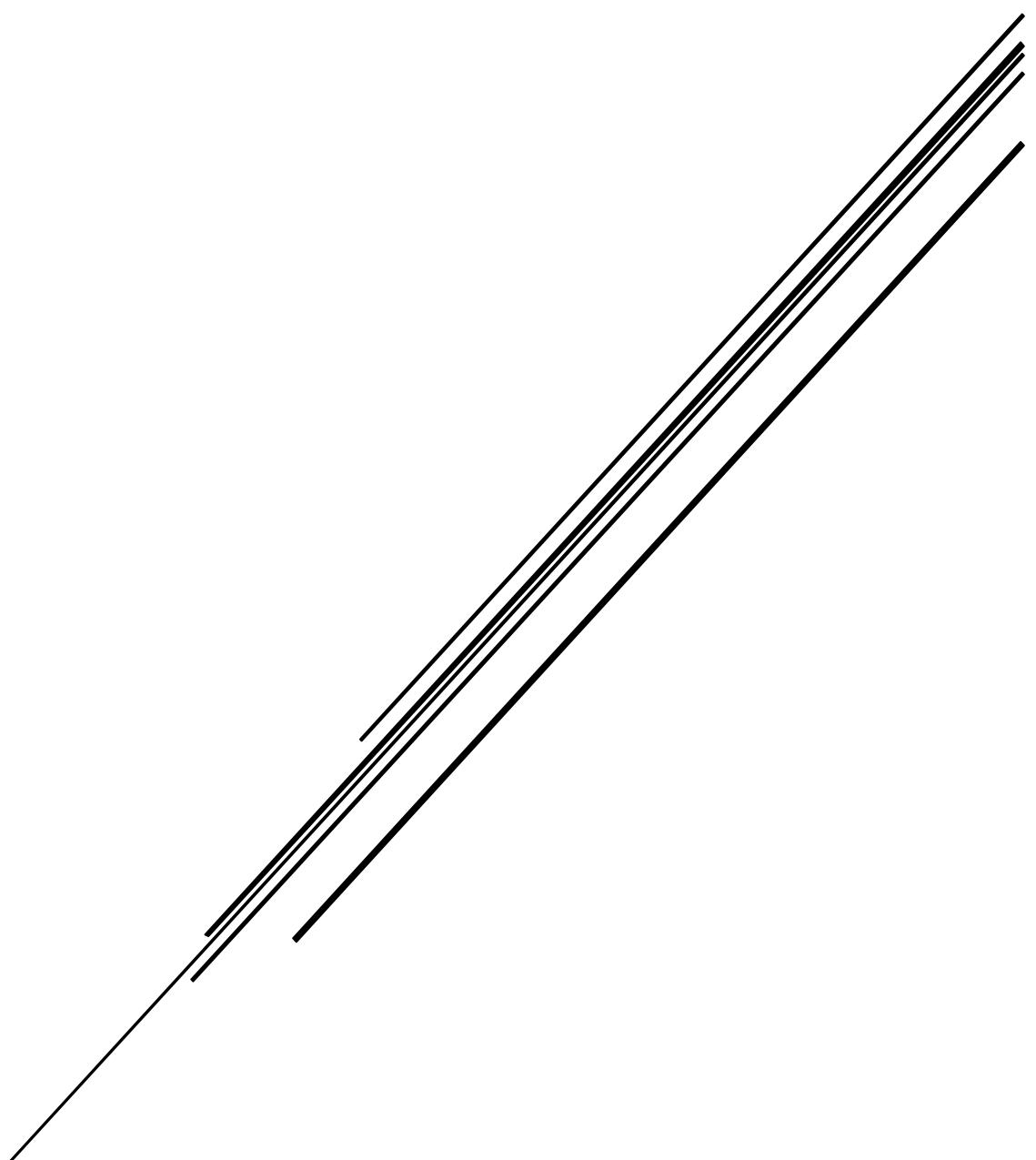


ALGORITMI DI OTTIMIZZAZIONE COMBINATORIA E SU RETE

Dal Problema NP-Hard alla Soluzione Euristica: Studio
del TSP in Ottica Combinatoria



Rivieccio Tommaso M63001845
Imparato Fabrizio M63001758

Sommario

1.	Introduzione	2
2.	Problemi di Ottimizzazione Combinatoria.....	2
2.1	Definizione.....	2
2.2	Travelling Salesman Problem	2
2.2.1	Modellazione	2
3.	Euristiche.....	4
3.1	Algoritmi Greedy	5
3.2	Ricerca locale.....	6
3.3	Simulated Annealing.....	8
4	Algoritmi euristici implementati	9
4.1	Euristica Greedy	9
4.2	Ricerca Locale.....	11
4.3	Simulated Annealing.....	12
5	Confronto con istanze	15
5.1	Att48.....	16
5.2	Ch130	17
5.3	Lin318.....	18
5.4	Pcb442.....	19
5.5	Pr2392	20
6	Conclusioni.....	20

1. Introduzione

Questo elaborato si apre con una definizione generale dei problemi di ottimizzazione combinatoria, concentrandosi in particolare sul Problema del Commesso Viaggiatore (TSP). In seguito, verranno analizzate le principali difficoltà computazionali associate a questi problemi e si discuterà di come sia possibile ottenere soluzioni di buona qualità in tempi ragionevoli tramite algoritmi di approssimazione, in particolare attraverso l'uso di euristiche. Successivamente verrà fatta una rassegna teorica delle euristiche selezionate per l'elaborato. Nella parte pratica, verranno selezionate diverse istanze del TSP. Utilizzando Gurobi e alcuni degli algoritmi euristici presentati nel corso, si procederà a stimare l'errore relativo di ciascuna soluzione e a valutare quale euristica risulti più efficace in termini di complessità computazionale e accuratezza della soluzione.

2. Problemi di Ottimizzazione Combinatoria

2.1 Definizione

Sia B un insieme finito detto insieme di base o ground set:

$$B = \{b_1, \dots, b_n\}$$

Sia invece Σ , una famiglia di sottoinsiemi di B detti subset system:

$$\Sigma = \{S_1, \dots, S_n\}$$

Sia $w: \Sigma \rightarrow R$ una funzione obiettivo che per ogni insieme S di Σ associa un numero reale $w(S)$.

Un problema di ottimizzazione combinatoria si presenta nella forma:

$$\min w(S), \quad S \in \Sigma$$

2.2 Travelling Salesman Problem

Un commesso viaggiatore deve visitare un certo numero di città partendo dalla sua città di residenza e ritornando alla fine del suo giro alla città da cui era partito. Vuole effettuare tutte le visite percorrendo la lunghezza complessiva minima.

Dato un grafo $G(V, A)$, un ciclo hamiltoniano è un ciclo che attraversa tutti i nodi del grafo una ed una sola volta. Se ad ogni arco del grafo associamo un peso positivo d_{ij} , ad ogni ciclo hamiltoniano si può associare un costo dato dalla somma dei pesi degli archi che lo compongono.

Risolvere il problema TSP significa cercare il circuito hamiltoniano di costo minimo. Se il grafo è pieno, il numero di circuiti costruibili è molto grande. Infatti, dati n archi, il numero totale di percorsi sarebbe dato dalle permutazioni di tali archi, ovvero $n!$

A seconda delle caratteristiche del grafo parliamo di TSP simmetrico se il grafo è non orientato, altrimenti asimmetrico se il grafo è orientato.

2.2.1 Modellazione

Partendo dal presupposto che un qualsiasi problema di cammino minimo può essere rappresentato come un problema di ottimizzazione combinatoria, anche il problema TSP può esser visto in questa maniera. Il ground set B corrisponderebbe con A , ovvero l'insieme degli

archi del grafo, necessario per definire le soluzioni del problema. Il sub set system sarebbe invece \mathbf{H} , ovvero l'insieme di tutti i circuiti Hamiltoniani.

Alla fine, la funzione obiettivo sarà:

$$w(H) = \sum_{u,v \in H} d_{uv}$$

Come prima cosa partiamo col definire le variabili decisionali del problema.

Possiamo definire la variabile $x_{ij} \in \{0, 1\}$:

- $x_{ij} = 1$ se l'arco $(i, j) \in A$ appartiene al ciclo hamiltoniano minimo $H_{min} \in \mathbf{H}$;
- $x_{ij} = 0$ altrimenti.

L'obiettivo è chiaramente quello di minimizzare il costo del percorso:

$$\min \sum_{(i,j) \in A} d_{ij} x_{ij}$$

Per quanto ne concerne i vincoli c'è bisogno che in ogni nodo j entri un arco e che in ogni nodo j ne esca uno.

$$\sum_{(i,j) \in A} x_{ij} = 1 \quad j \in V$$

$$\sum_{(j,i) \in A} x_{ji} = 1 \quad j \in V$$

Questi vincoli prendono il nome di vincoli di assegnamento. È fondamentale che il circuito hamiltoniano non ammetta sotto giri. È infatti possibile che i vincoli d'assegnamento vengano rispettati, ma che al contempo non vengano rispettati i vincoli di assenza di sotto giri.

Dato un grafo orientato, è possibile che questo grafo contenga al suo interno dei cicli separati tra loro, il che porta ad avere soluzioni non ammissibili. Quindi i vincoli di assenza di sotto giri si rendono necessari per tagliare fuori dall'insieme delle soluzioni ammissibili soluzioni che soddisfino i vincoli di assegnamento ma non ammissibili. È necessario imporre che il numero di archi che hanno origine e destinazione sui nodi sia minore o uguale della cardinalità dell'insieme dei nodi, meno 1. Così facendo siamo in grado di eliminare i sotto giri:

$$\sum_{i \in S, j \in S, (i,j) \in A} x_{ij} \leq |S| - 1 \quad \forall S \subset V : 2 \leq |S| \leq |V| - 1$$

È chiaro che questo discorso vada fatto per ogni sottoinsieme di V , escluso ovviamente V .

Perciò se n è la cardinalità di V , allora posso rappresentare i suoi sottoinsiemi con stringhe binarie di n bit, dove ogni bit indica la presenza di un elemento nel sottoinsieme S . Ne consegue che esistono circa 2^n possibili sottoinsiemi, e quindi anche un numero esponenziale di possibili vincoli per l'assenza di sotto-cicli.

Data perciò la complessità esponenziale, una strategia intelligente potrebbe essere quella di determinare dei lower bounds. Un possibile lower bound della soluzione è ottenibile semplicemente considerando una formulazione in cui non vi sia alcun vincolo di sotto giro. In tal caso la soluzione è ammssibile e ottima per il TSP se la soluzione non contiene sotto giri, altrimenti si ottiene solo un lower bound.

Sperimentalmente, si nota che dei 2^n vincoli, non tutti sono necessari, ma è sufficiente aggiungere solo alcuni alla formulazione per ottenere una soluzione ammssibile soddisfacente. Questa osservazione permette di pensare ad un algoritmo esatto per la soluzione del problema TSP asimmetrico, ovvero ATSP. Parliamo di algoritmo a generazioni di vincoli, o row generation.

1. *Si eliminano tutti i vincoli di sotto giro e i vincoli di interezza e si risolve il corrispondente problema di assegnamento*
2. *Si reintroducono nel modello i vincoli di interezza delle variabili*
3. *Se la soluzione corrente non contiene sotto cicli allora abbiamo trovato la soluzione ottima, quindi l'algoritmo termina*
4. *In caso contrario si individuano uno o più sotto cicli nella soluzione corrente*
5. *Si aggiungono al modello i vincoli di eliminazione di almeno uno dei sotto cicli individuati*
6. *Si risolve nuovamente il problema corrente e si ritorna al passo 3.*

Perciò nel caso peggiore sarà necessario introdurre tutti i 2^n vincoli, e risolvere un problema di PLI.

Naturalmente per capire se una soluzione sia ammssibile o meno, il modo più semplice è usare un algoritmo di ispezione di un grafo, che a partire da un nodo cerca di visitare tutti gli altri nodi, come nel caso di DFS.

3. Euristiche

Tutti i problemi possono essere classificati in:

- **P:** classe di problemi che possono essere risolti in tempi polinomiali.
- **NP:** classe di problemi che possono essere verificati in tempi polinomiale.
- **NP-Hard:** classe di problemi che non possono essere risolti in tempi polinomiali.

La quasi totalità dei problemi di ottimizzazione combinatoria è di tipo NP-hard e quindi per aver una soluzione ottima sono richiesti tempi di calcolo esponenziali.

Gli algoritmi euristici sono approssimanti e cercano una buona soluzione che non sia necessariamente ottima, ma abbia tempi di calcolo più contenuti.

La bontà di una soluzione viene quindi valutata in termini di scarto percentuale tra la soluzione ottenuta e quella individuabile attraverso una tecnica esatta.

Perciò l'efficacia di una tecnica euristica viene valutata considerando due criteri in antitesi tra loro, ovvero la qualità della soluzione e i tempi di calcolo necessari per ottenerla.

Dunque, il progetto di una euristica efficiente deve ricercare un giusto compromesso tra la velocità di calcolo e la bontà della soluzione, oltre a dover avere una bassa difficoltà implementativa, ed una buona flessibilità, intesa come la adattabilità dell'euristica all'applicazione in problemi differenti.

Posto I una istanza di un dato problema P , sia $EUR(I)$ il valore della soluzione fornita dall'euristica, e $OPT(I)$ la soluzione ottima determinata da un algoritmo esatto. Come detto in precedenza, è possibile stabilire un errore percentuale relativo, ovvero un gap, calcolabile come lo scostamento percentuale relativo tra la soluzione ottima e quella sub ottima ottenuta con una euristica.

$$gap = \frac{|OPT(I) - EUR(I)|}{|OPT(I)|} 100$$

A questo punto gli algoritmi euristicci possono essere classificati in primo luogo secondo lo scarto.

Algoritmi ad errore massimo garantito sono quegli algoritmi per i quali è possibile fornire un limite massimo all'errore, e hanno sostanzialmente una importanza prevalentemente teorica.

$$gap \leq \varepsilon \quad \forall I$$

Algoritmi euristicci con stima dell'errore sono invece algoritmi che data una istanza del problema, forniscono una soluzione ammissibile ed una stima per eccesso, o upper bound, della distanza della soluzione fornita da quella ottima.

Per le euristiche che non danno alcuna stima dell'errore, la loro valutazione può essere effettuata generando casualmente una serie di istanze del problema di dimensioni diverse e andando a valutare le statistiche dell'errore su queste istanze. In assenza di algoritmi esatti la valutazione dell'errore si può fare utilizzando procedure che forniscono limiti alla soluzione ottima.

Per esempio, in un problema a minimizzare se si indica con $LB(I)$ il valore di lower bound della soluzione ottima, si può ottenere una stima dell'errore per eccesso, come:

$$gap \leq \frac{|LB(I) - EUR(I)|}{|LB(I)|} 100$$

Possiamo a questo punto suddividere gli algoritmi euristicci in due classi principali. Quelli costruttivi che costruiscono gradualmente la soluzione attraverso il passaggio per soluzioni parziali, secondo un approccio iterativo. Abbiamo poi degli algoritmi migliorativi, che sono algoritmi che partendo da una soluzione ammissibile del problema, tentano di modificarla, migliorandola.

3.1 Algoritmi Greedy

Gli Algoritmi Greedy, o algoritmi avidi, sono algoritmi che ad ogni iterazione aggiungono un pezzo alla soluzione, classificandosi quindi come algoritmi costruttivi. Ad ogni iterazione, tra tutte le possibili aggiunte che possono contribuire alla soluzione parziale, viene aggiunto quello di costo inferiore, senza preoccuparsi della struttura complessiva della soluzione. Capita quindi che le ultime iterazioni risultino inefficienti dal momento che le possibilità di scelta si siano notevolmente ridotte.

Dato un problema di ottimizzazione combinatoria, definiamo l'insieme di base, B , il subset System Σ e la funzione obiettivo w .

Una soluzione parziale è un sottoinsieme T dell'insieme di base B ($T \subseteq B$) che può essere ricondotto ad una soluzione ammissibile appartenente all'insieme Σ soltanto aggiungendo elementi di B (*gli elementi appartenenti a $B - T$*).

Ma allora un algoritmo greedy non fa altro che costruire una sequenza di soluzioni parziali fino ad arrivare ad una soluzione ammissibile. Una euristica greedy può quindi essere schematizzata come:

1. *Inizializzazione in cui si pone $T_0 = \emptyset$ e $i = 1$.*
2. *Si sceglie un elemento $e \in B - T_{i-1}$ tale che l'insieme T_{i-1} con l'aggiunta dell'elemento e sia una soluzione parziale e che $w(T_{i-1} \cup \{e\})$ sia minimo*
3. *Si pone $T_i = (T_{i-1} \cup \{e\})$*
4. *Se T_i appena identificato è una soluzione ammissibile, l'algoritmo termina*
5. *Altrimenti, si pone $i = i + 1$ e si ritorna al passo 2.*

È chiaro che al passo 2 sia possibile che esistano più elementi che minimizzino il costo della nuova soluzione parziale, ed in questo caso è necessario una regola nel caso di pareggio, detta Tie Breaking Rule. A volte si sceglie semplicemente a caso, mentre altre volte può essere opportuno definire nuove funzioni di costo.

3.2 Ricerca locale

Le euristiche di Ricerca locale si basano su concetti di intorno di una soluzione ammissibile corrente.

Algoritmo di questo tipo restituisce in output un minimo locale del problema, che non è detto sia l'ottimo. Perciò gli algoritmi di ricerca, data una soluzione corrente $H_0 \in S$, costruiscono un intorno di H_0 tramite soluzioni vicine.

Si ispeziona quindi un intorno di H_0 e si calcola la migliore soluzione H_1 appartenente all'intorno.

Se H_1 coincide con H_0 l'euristica termina, altrimenti si itera il procedimento ripartendo da H_1 .

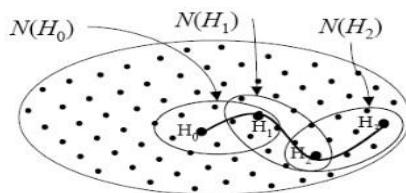


Figura 1: Rappresentazione Algoritmo di Ricerca Locale

In particolare, il punto di partenza è fondamentale poiché potrebbe influire sul punto di convergenza dell'euristica. Al contempo la scelta della grandezza dell'intorno è determinante, in quanto consentirebbe di identificare un set di punti maggiore, e quindi eventualmente punti migliori. Tuttavia, allargare l'intorno si traduce nell'ispezionare più soluzioni ammissibili, e quindi la singola iterazione diventa computazionalmente più onerosa. Poniamo due definizioni fondamentali interamente ad una euristica di ricerca locale.

Una mossa è una operazione che a partire dalla soluzione corrente consente di generare altre soluzioni ammissibili per il problema di partenza con alcune caratteristiche simili alla

soluzione di partenza. Un intorno di una soluzione H_0 è invece l'insieme costituito da tutte le soluzioni ottenibili applicando una determinata mossa alla soluzione H_0 .

Possiamo strutturare una euristica di ricerca locale come:

1. *Si sceglie una soluzione iniziale H_0 detta di innesco, per il processo di ricerca*
2. *Si definisce un intorno $N(H_0)$ della soluzione corrente ottenuto applicando una data mossa alla soluzione corrente*
3. *Si individua una nuova soluzione $H_1 \in S$ cui corrisponde il minimo della funzione obiettivo: $H_1: w(H_1) = \min w(H) \forall H \in \Sigma$*
4. *Se $w(H_1) \leq w(H_0)$, in un problema a minimizzare, si sostituisce H_0 con H_1 , per poi ritornare al passo 2, altrimenti si assume la soluzione H_0 come soluzione finale.*

Per migliorare la qualità delle soluzioni di una euristica di ricerca locale è quindi possibile applicare più volte l'algoritmo a partire da diverse soluzioni iniziali, oppure, come già anticipato, aumentare la dimensione degli intorni. Nel primo caso parliamo di Approccio Multi-start, dove le soluzioni iniziali possono essere costruite in modo casuale, oppure in modo guidato, al fine di distribuirle su tutto lo spazio delle soluzioni. Se il problema presenta numerosi punti di minimo locale, ed un unico minimo globale (nascosto dai molti minimi locali vicini tra loro), diventa difficile raggiungere questo picco, a meno di non scegliere una opportuna soluzione iniziale.

In generale all'aumentare della dimensione degli intorni considerati migliora quindi la qualità delle soluzioni, ma peggiora il tempo di calcolo. Nel caso del TSP, una 2-opt creerà una dimensione di ogni intorno del tipo $O(n^2)$, mentre nel caso di 3-opt avremo intorni di dimensione dell'ordine di $O(n^3)$.

Ciò significa banalmente che per un grafo di 100 nodi, tipicamente piccolo, ogni esplorazione dell'intorno 3-opt richiederà 1000000 di valutazioni della funzione obiettivo.

Naturalmente sarebbe possibile generalizzare al caso k -opt, fino ad un massimo $k = n$. In tal caso l'intorno contiene tutti i possibili circuiti Hamiltoniani, risultando in un intorno di grandezza $O(n!)$, che è assolutamente inaccettabile.

Bisogna quindi stabilire come ottenere un trade-off tra accuratezza e tempo di calcolo. Si può pensare di diversificare, esplorando regioni diverse tra loro, come accade ad esempio nell'approccio multi-start, o intensificare, aumentando la dimensione degli intorni ed esplorando a fondo una unica regione.

Naturalmente la scelta tra l'una e l'altra dipende dalle performance dell'algoritmo per il problema affrontato, il che ci porta a dire che non esiste una regola assoluta per effettuare una decisione, e che quindi ci saranno lunghe fasi di tuning dei parametri.

Dunque, per poter scegliere la soluzione corrente in maniera efficiente, si possono adottare due politiche diverse.

La prima politica, detta di First Improvement prevede di effettuare lo scambio tra la soluzione corrente e la prima soluzione migliorativa trovata. Ciò significa naturalmente che solo l'ultima iterazione ispezionerà per intero l'intorno.

In alternativa si può adottare una politica di Best Improvement, che prevede l'ispezione di un intorno della soluzione corrente, valutando funzione obiettivo in tutti i punti di questo intorno, per poi prendere il punto in cui corrisponde il valore minimo di funzione obiettivo.

3.3 Simulated Annealing

Il Simulated Annealing è un algoritmo euristico basato su un fenomeno fisico: il processo di tempra di un solido. Tale processo viene simulato da un algoritmo, detto algoritmo di Metropolis.

A partire dalla posizione delle molecole, definiamo lo *stato corrente* S alla *temperatura* T con *energia* E . Applicando una perturbazione, una molecola si sposterà in una nuova posizione. Questo porta il sistema a raggiungere un *nuovo stato* S' , che avrà un'*energia* E' diversa da quella precedente. Il nuovo stato viene accettato come stato corrente con una probabilità data da:

$$P(\text{accettazione}) = \begin{cases} 1, & \Delta E \leq 0 \\ e^{-\frac{\Delta E}{KT}}, & \Delta E > 0 \end{cases}$$

Dove ΔE è la *differenza tra E' ed E* e K è la *costante di Boltzmann*. La temperatura T viene abbassata gradualmente e l'algoritmo si ferma quando il suo valore è tale da risultare in una probabilità di accettazione prossima allo zero.

Questo algoritmo può essere usato per la risoluzione di problemi di ottimizzazione combinatoria, con le opportune sostituzioni:

- L'*energia* corrisponde alla *soluzione obiettivo*;
- Lo *stato corrente* corrisponde alla *soluzione corrente* del problema;
- Lo *spostamento di una molecola* corrisponde ad una *mossa* che porta ad una nuova soluzione;
- La *temperatura* corrisponde ad un *parametro di controllo* dell'algoritmo.

Facendo queste sostituzioni la nuova formula che si ottiene riguardo la *probabilità di accettazione* di una nuova soluzione è:

$$P(\text{accettazione di } x') = \begin{cases} 1, & f(x') \leq f(x) \\ e^{-\frac{f(x')-f(x)}{T_k}}, & f(x') > f(x) \end{cases}$$

Il parametro T_k viene abbassato nel corso dell'algoritmo, abbassando la probabilità di accettare soluzioni peggiorative.

Prima di avviare l'algoritmo c'è bisogno di definire alcuni valori:

- La *temperatura iniziale* T_0 ;
- La *temperatura finale* T_f ;
- Il numero di *transizioni* L_k da effettuare per ogni valore T_k della *temperatura*;
- La *legge di decremento* di T .

Il valore di T_0 deve essere scelto in modo tale che nella fase iniziale dell'algoritmo si scelgano tutte le transizioni, e cioè che:

$$e^{-\frac{\Delta f}{T_0}} \approx 1, \quad \forall \Delta f > 0$$

Per esempio, considerando il valore della *funzione obiettivo* f_0 alla temperatura T_0 consideriamo la massima variazione della funzione obiettivo, cioè Δf_{max} . Per essere sicuri di accettare queste variazioni potremmo porre $T_0 = 10\Delta f_{max}$.

Il valore di T_f deve essere scelto in modo tale che la probabilità di accettazione di soluzioni peggiorative deve essere nulla. Cioè:

$$e^{\frac{\Delta f}{T_f}} \approx 0, \quad \forall \Delta f > 0$$

Sempre considerando il valore della *funzione obiettivo* f_0 consideriamo la minima variazione della funzione obiettivo Δf_{min} . Potremmo porre $T_f = 10^{-1} \Delta f_{min}$.

La legge più frequente per decrementare la temperatura dalla generica iterazione k all'iterazione $k + 1$ è:

$$T_{k+1} = \alpha T_k, \quad \alpha < 1$$

I valori di L_k si scelgono in base al valore di α :

- Se α è elevato, il decremento è più lento, e posso fare meno iterazioni per temperatura;
- Se α è basso, il decremento è più veloce ed è più opportuno fare più iterazioni per temperatura.

Un possibile valore di L_k è quello di un L costante correlato alla dimensione del problema, come $L = n$. Altrimenti si può far variare in base al numero μ_{min} di nuove soluzioni accettate.

Un altro criterio di arresto che può essere usato è il *criterio di arresto basato sulla stagnazione*, cioè l'algoritmo viene bloccato quando per un numero m di volte non viene trovata una soluzione che migliora la migliore soluzione trovata nelle iterazioni precedenti.

4 Algoritmi euristicici implementati

In questo capitolo verranno presentate in dettaglio le implementazioni delle tre heuristiche selezionate per la risoluzione del problema.

4.1 Euristica Greedy

Per trovare una soluzione iniziale, alla quale poi applicare euristiche migliorative, è stata utilizzata l'**euristica greedy**.

Il programma costruisce un dizionario che contiene i nodi e le loro coordinate spaziali a partire da un file “.tsp” che segue questa struttura:

$$\text{nodo}: (x, y)$$

- *nodo* è la chiave;
- (x, y) è la coppia di coordinate che indica la posizione del nodo.

Gli archi vengono poi calcolati come distanza tra i vari nodi. È importante indicare anche la struttura dati usata per la loro memorizzazione; infatti, questo ha inficiato molto sulle prestazioni dell'algoritmo. Usando un dizionario così strutturato:

$$(i, j): d_{ij}$$

- (i, j) è la chiave ed indica l'arco che va dal nodo i al nodo j ;
- d_{ij} è il costo di quell'arco, cioè la distanza tra il nodo i e il nodo j .

così come viene fatto negli esempi visti a lezione con Gurobi, si ottengono prestazioni estremamente più basse con l'aumentare della dimensione del grafo, rispetto all'uso di matrici.

Per questo si è optato per una soluzione che utilizzi le matrici:

```

1.     distancesMatrix: List[List[float]] = [[0.0 for _ in range(dimension)] for _ in range(dimension)]
2.
3.     for i in range(0, dimension):
4.         for j in range(0, dimension):
5.             if i != j:
6.                 xDistance = nodes.get(i+1)[0] - nodes.get(j+1)[0]
7.                 yDistance = nodes.get(i+1)[1] - nodes.get(j+1)[1]
8.                 distancesMatrix[i][j] = sqrt(xDistance**2 + yDistance**2)
9.             else:
10.                distancesMatrix[i][j] = inf

```

Infatti, considerando l'istanza analizzata *pr2392*, contenente 2392 nodi, i tempi di esecuzione erano intorno ai 5-6 minuti utilizzando un dizionario, e sono stati ridotti a soli 0.25 secondi usando le matrici. Questo è dovuto ad un'operazione di filtraggio degli archi necessaria e molto lenta nel caso dei dizionari, ma molto veloce con le matrici.

L'algoritmo trova una soluzione seguendo questa funzione:

```

1.     def calculateNearestNeighbour(edges: List[List[int]], dimension: int, start: int = 1):
2.
3.         solution: Dict[Tuple[int, int], float] = {}
4.         distance: float = 0.0
5.
6.         currentNode: int = start - 1
7.         visitedNodes = set()
8.         visitedNodes.add(currentNode)
9.
10.        for _ in range(0, dimension - 1):
11.            minimum: float = inf
12.            minNode: int = -1
13.
14.            for j in range(dimension):
15.                if (j not in visitedNodes) and (edges[currentNode][j] < minimum):
16.                    minimum = edges[currentNode][j]
17.                    minNode = j
18.
19.            solution[(currentNode+1, minNode+1)] = minimum
20.            distance += minimum
21.
22.            visitedNodes.add(minNode)
23.            currentNode = minNode
24.
25.        # Ultima mossa
26.        solution[(currentNode+1, start)] = edges[currentNode][start-1]
27.        distance += edges[currentNode][start-1]
28.
29.        return solution, distance

```

A partire da una soluzione vuota, una lista di nodi visitati vuota e un nodo di partenza, compio $n - 2$ mosse greedy che minimizzano il costo. Per ogni nodo i sul quale mi trovo, verifico se il nodo destinazione j non è stato già visitato e se è il nodo più conveniente verso cui muovermi. La mossa $n - 1$ viene fatta al di fuori del ciclo e chiude il ciclo hamiltoniano, generando una soluzione ammissibile.

L'unico problema è che questo algoritmo greedy è limitato ad un nodo di partenza; quindi, per ottenere la miglior greedy è stato pensato di progettare un algoritmo **greedy multi-start**. In questo modo viene applicato l'algoritmo greedy con partenza su ogni nodo, e la soluzione ammissibile migliore viene memorizzata ed usata per gli approcci migliorativi successivi.

```

1. def calculateBestNearestNeighbour(edges: Dict[Tuple[int, int], float], dimension: int):
2.
3.     solution: Dict[Tuple[int, int], float] = {}
4.     distance: float = inf
5.     bestStart: int = 0
6.
7.     for i in range(1, dimension + 1):
8.
9.         tempSolution, tempDistance = calculateNearestNeighbour(edges, dimension, i)
10.
11.        if tempDistance < distance:
12.            distance = tempDistance
13.            solution = tempSolution
14.            bestStart = i
15.
16.    return solution, distance, bestStart

```

Le prestazioni e i parametri sia della singola greedy, che dell'approccio multi-start, sono stati misurati e salvati in memoria per poterne permettere un più facile uso nei programmi che implementano le euristiche migliorative analizzate nel progetto.

4.2 Ricerca Locale

Per la fase di ricerca locale, è stata adottata come tecnica di innesco una soluzione fornita dall'algoritmo Greedy Multi Start. La ricerca locale è stata implementata in due varianti:

- **First Improvement**
- **Best Improvement**

Entrambe si basano sulla mossa 2-opt, utilizzata per generare l'intorno delle soluzioni.

La soluzione iniziale viene fornita sotto forma di una lista di archi. Tuttavia, per semplificare l'elaborazione, la ricerca locale converte tale rappresentazione in una lista ordinata di nodi che descrive il tour orientato. Questo formato consente una generazione più diretta delle mosse 2-opt, calcolando tutte le combinazioni di coppie di indici (i, k) con $i < k$.

Generazione dello spazio delle mosse 2-opt

```

1. def generateTwoOptSpace(tour: List[int]) -> List[Tuple[int,int]]:
2.     n = len(tour)
3.     swaps = list(combinations(range(1,n-2), 2))
4.     return swaps

```

Lo spazio viene generato come tutte le possibili combinazioni di indici (i, k) con $i < k$. In questo caso, lavorando con gli indici, non c'è bisogno di generare ulteriormente questa lista.

Valutazione dell'accettabilità di una nuova soluzione

Questa funzione valuta se una soluzione candidata ha una funzione obiettivo migliore rispetto alla soluzione corrente, basandosi su una rappresentazione del grafo come dizionario delle distanze.

```

1. def isAcceptable(startTour:List[int],endTour:List[int],distances:Dict[Tuple[int,int],float])-> bool:
2.     return valueObj(endTour, distances)<valueObj(startTour, distances)

```

Applicazione della mossa 2-opt

```

1. def twoOptSwap(tour: List[int], i: int, k: int) -> List:
2.     return tour[:i] + list(reversed(tour[i:k+1])) + tour[k+1:]

```

La mossa viene applicata preservando la lista prima di i e dopo k ed invertendo tutta la sequenza intermedia. In questo modo quando si andrà a ricalcolare la funzione obiettivo come somma degli archi, si terrà in considerazione degli swap tra gli archi non adiacenti.

Strategie di Ricerca

First Improvement

La strategia First Improvement esegue una scansione delle mosse 2-opt e applica la prima mossa migliorativa trovata. Dopo ogni miglioramento, l'elenco delle mosse viene rimescolato casualmente (shuffle) per aumentare la variabilità nella ricerca e migliorare i tempi di convergenza. Il processo continua finché non vengono trovati ulteriori miglioramenti.

```

1. def localSearchFirstImprovement(tour: List, distances: Dict[Tuple[int,int], float]) -> List:
2.     moveSpace = generateTwoOptSpace(tour)
3.     noImprovement = False
4.     while not noImprovement:
5.         improvement_found = False
6.         for (i, k) in moveSpace:
7.             newTour = twoOptSwap(tour, i, k)
8.             if isAcceptable(tour, newTour, distances):
9.                 tour = newTour
10.                random.shuffle(moveSpace)
11.                improvement_found = True
12.                break
13.            if not improvement_found:
14.                noImprovement = True
15.    return tour

```

Best Improvement

La strategia Best Improvement, invece, esplora sistematicamente l'intero spazio delle mosse 2-opt a ogni iterazione, selezionando la migliore tra tutte le soluzioni maggiorative disponibili. In questo caso non viene eseguito lo shuffle poiché l'obiettivo è garantire la selezione del miglior miglioramento possibile. Se nessuna mossa produce un miglioramento, l'algoritmo si arresta.

```

1. def localSearchBestImprovement(tour: List, distances: Dict[Tuple[int,int], float]) -> List:
2.     moveSpace = generateTwoOptSpace(tour)
3.     noImprovement = False
4.     while not noImprovement:
5.         improvement_found = False
6.         for (i, k) in moveSpace:
7.             newTour = twoOptSwap(tour, i, k)
8.             if isAcceptable(tour, newTour, distances):
9.                 tour = newTour
10.                improvement_found = True
11.            if not improvement_found:
12.                noImprovement = True
13.    return tour

```

4.3 Simulated Annealing

L'algoritmo di Simulated Annealing (SA) è stato adottato come tecnica di ottimizzazione per affrontare il problema trattato, grazie alla sua capacità di esplorare efficacemente lo spazio delle soluzioni ed evitare minimi locali. Per l'implementazione, sono stati presi in considerazione tre diversi approcci di gestione degli iperparametri, con l'obiettivo di migliorare progressivamente la qualità delle soluzioni ottenute:

- Impostazione con **parametri di default**;
- Ottimizzazione tramite **ricerca randomica** (random search);
- Ottimizzazione tramite **ricerca a griglia** (grid search).

I principali iperparametri coinvolti nell'algoritmo SA sono i seguenti:

- **Temperatura iniziale (tK)**: controlla la probabilità iniziale di accettare soluzioni peggiorative;
- **Temperatura finale (tF)**: determina il termine del processo di raffreddamento;
- **Fattore di raffreddamento (α)**: coefficiente che regola la decrescita della temperatura ad ogni ciclo;
- **Numero massimo di iterazioni senza miglioramenti** consecutivi ($nNoImprovement$);
- **Numero di iterazioni per ciascun livello di temperatura** ($nIter$).

L'algoritmo utilizza la mossa 2-opt come operatore di generazione dell'intorno, che consente di creare una nuova soluzione candidata invertendo un sottoinsieme di archi nel tour corrente.

Inizializzazione dei parametri (Default)

Nel primo approccio, i parametri sono calcolati tramite formule euristiche, generalizzabili su diverse istanze del problema:

```
1. tK = 10*abs(solutionDistance/2) # Temperatura iniziale
2. tF = 1e-4*abs(solutionDistance) # Temperatura finale
3. nIter = n # Numero iterazioni per ogni temperatura
4. alfa = 0.995 # Coefficiente di raffredamento
5. nNoImprovement = 5*n # Numero di mosse peggiorative consecutive
```

Criterio di accettazione

La funzione di accettazione implementa il criterio di Metropolis, che consente l'accettazione di soluzioni peggiorative con una probabilità decrescente nel tempo, favorendo l'esplorazione nella fase iniziale e la convergenza nella fase finale:

```
1. def
isAcceptable(startTour: List[int], endTour: List[int], T_k: float, distances: Dict[Tuple[int,int], float]) ->
bool:
2.     delta = valueObj(endTour, distances) - valueObj(startTour, distances)
3.     if delta < 0:
4.         return True
5.     else:
6.         p = math.exp(-delta / T_k)
7.         return random.random() < p
```

Una soluzione viene accettata incondizionatamente se migliora l'obiettivo. In caso contrario, è accettata con una probabilità che dipende negativamente dal peggioramento (δ) e positivamente dalla temperatura corrente (T_k).

Strategie di Ricerca

L'algoritmo principale di SA è il seguente:

```
1. def simulatedAnnealing(T_k: float, alfa: float, tour: List[int], distances: Dict[Tuple[int,int], float], nIter: int, nNoImprovement: int, tF: float, seed: int = int(time.time())) -> List[int]:
2.     if seed is not None:
3.         random.seed(seed)
4.     noImprovement: int = 0
5.     bestTour: List[int] = copy.deepcopy(tour)
6.     while T_k > tF and noImprovement < nNoImprovement:
```

```

7.     for _ in range(nIter):
8.         i, k = sorted(random.sample(range(1, len(tour) - 2), 2))
9.         newTour = twoOptSwap(tour, i, k)
10.        if isAcceptable(tour, newTour, T_k, distances):
11.            tour = newTour
12.            if valueObj(tour, distances) < valueObj(bestTour, distances):
13.                bestTour = copy.deepcopy(tour)
14.                noImprovement = 0
15.            else:
16.                noImprovement += 1
17.            else:
18.                noImprovement += 1
19.        T_k *= alfa
20.    return bestTour

```

Il ciclo esterno regola il processo di raffreddamento, mentre il ciclo interno genera iterazioni locali tramite la mossa 2-opt.

Ottimizzazione degli Iperparametri

Ricerca Randomica

Il metodo di ricerca randomica esplora lo spazio dei parametri generando combinazioni casuali entro range predefiniti. Viene testata ogni configurazione eseguendo l'algoritmo SA e registrando la soluzione migliore trovata:

```

1. def randomSearchSimulatedAnnealing(startTour: List[int], distances: Dict[Tuple[int, int], float], tK:
   float, tF: float, nIter: int, nNoImprovement: int, alfa: float, nTest: int):
2.     best_score = float("inf")
3.     best_params = None
4.     best_seed = None
5.     best_tour: List = []
6.     print(f"Testing {nTest} combinations...")
7.     startTime = time.perf_counter()
8.     base_seed = int(time.time())
9.     for idx in range(nTest):
10.         seed = base_seed + idx # seed unico per ogni combinazione, riproducibile
11.         alfaValue = random.uniform(0.8, alfa)
12.         nIterValue = random.randint(10, int(1.5*nIter))
13.         tKValue = random.uniform(0.25*tK, 1.5*tK)
14.         tFValue = random.uniform(0.25*tF, 1.5*tF)
15.         nNoImprovementValue = random.randint(10, int(1.5*nNoImprovement))
16.         print(f"\nTesting: tK={tKValue}, alfa={alfaValue}, nIter={nIterValue},
   nNoImprovement={nNoImprovementValue}, tf={tFValue} ,seed={seed}")
17.         tour_copy = copy.deepcopy(startTour)
18.         resultTour = simulatedAnnealing(tKValue, alfaValue, tour_copy, distances, nIterValue,
   nNoImprovementValue, tFValue, seed=seed)
19.         score = valueObj(resultTour, distances)
20.         print(f"→ Score: {score:.2f}")
21.         if score < best_score:
22.             best_score = score
23.             best_params = (tKValue, alfaValue, nIterValue, nNoImprovementValue, tFValue)
24.             best_seed = seed
25.             best_tour = resultTour
26.     endTime = time.perf_counter()
27.     print("\n☒ Best combination:")
28.     print(f"tK={best_params[0]}, alfa={best_params[1]}, nIter={best_params[2]},
   nNoImprovement={best_params[3]}, tf={best_params[4]}, seed={best_seed}")
29.     print(f"Best score: {best_score:.2f}. Time: {(endTime-startTime):.6f}s")
30.     return best_tour

```

Ricerca a Griglia

Nel caso della ricerca a griglia, i parametri sono esplorati su un insieme discreto e predefinito di combinazioni. Questo approccio garantisce una copertura sistematica ma meno esplorativa rispetto alla random search:

```

1. def gridSearchSimulatedAnnealing(startTour: List[int], distances: Dict[Tuple[int, int],
2.     float], paramGrid: List[Tuple[float, float, float, float]]):
3.     best_score = float("inf")
4.     best_params = None
5.     best_seed = None
6.     best_tour: List[] = []
7.     print(f"Testing {len(paramGrid)} combinations...")
8.     startTime = time.perf_counter()
9.     base_seed = int(time.time())
10.    for idx, (tK, alfa, nIter, nNoImprovement, tF) in enumerate(paramGrid):
11.        seed = base_seed + idx # seed unico per ogni combinazione, riproducibile
12.        print(f"\nTesting: tK={tK}, alfa={alfa}, nIter={int(nIter)},"
13.            f"nNoImprovement={int(nNoImprovement)}, tF={tF}, seed={seed}")
14.        tour_copy = copy.deepcopy(startTour)
15.        resultTour = simulatedAnnealing(tK, alfa, tour_copy, distances, int(nIter),
16.            int(nNoImprovement), tF, seed=seed)
17.        score = valueObj(resultTour, distances)
18.        print(f"→ Score: {score:.2f}")
19.        if score < best_score:
20.            best_score = score
21.            best_params = (tK, alfa, nIter, nNoImprovement)
22.            best_seed = seed
23.            best_tour = resultTour
24.    endTime = time.perf_counter()
25.    print("\n☒ Best combination:")
26.    print(f"tK={best_params[0]}, alfa={best_params[1]}, nIter={best_params[2]},"
27.        f"nNoImprovement={best_params[3]}, tF={tF}, seed={best_seed}")
28.    print(f"Best score: {best_score:.2f}. Time: {(endTime-startTime):.6f}s")
29.    return best_tour

```

5 Confronto con istanze

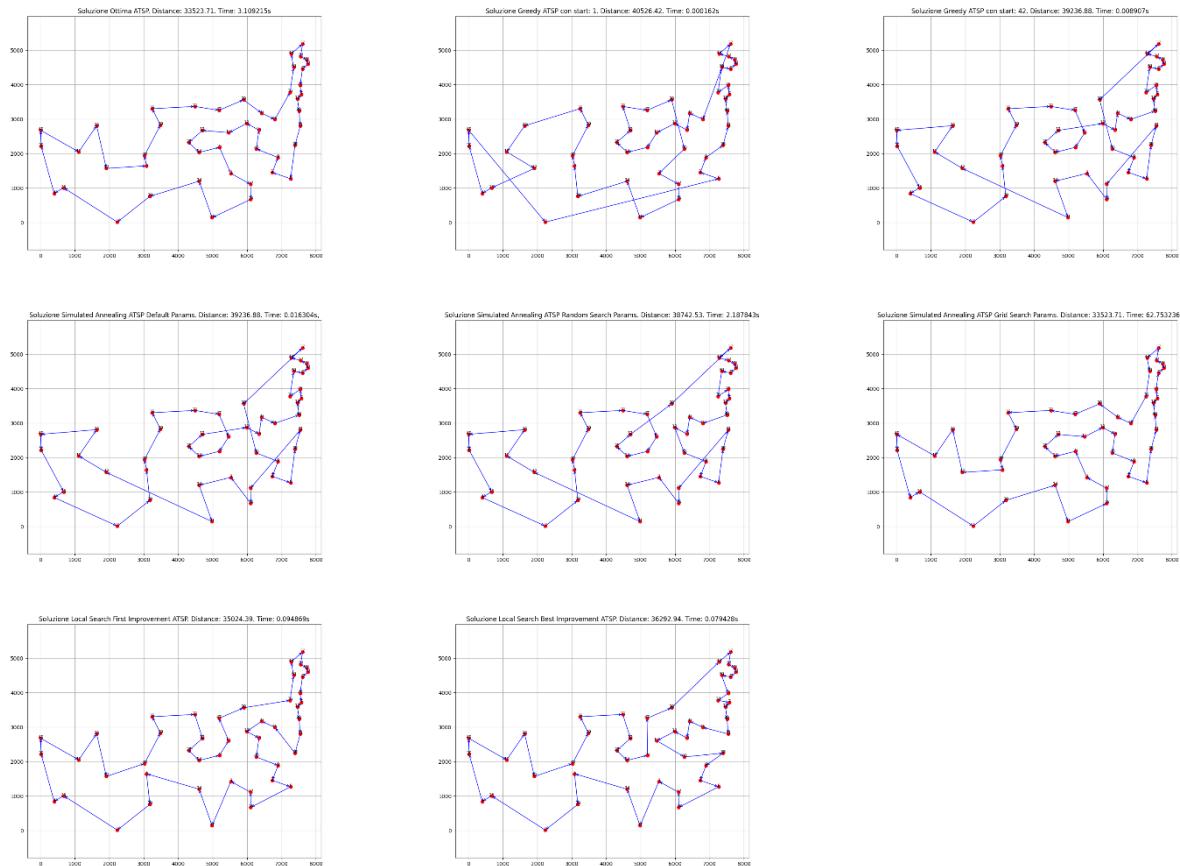
In questo capitolo verranno analizzati i dati sperimentali raccolti su un insieme di istanze del problema del Traveling Salesman Problem (TSP), selezionate con complessità crescente in termini di numero di nodi. L'analisi si concentrerà sia sui risultati quantitativi prodotti dai diversi algoritmi, sia sulle topologie dei tour generati, con l'obiettivo di valutarne l'efficacia e il comportamento al variare della dimensione del problema.

I due parametri principali considerati sono:

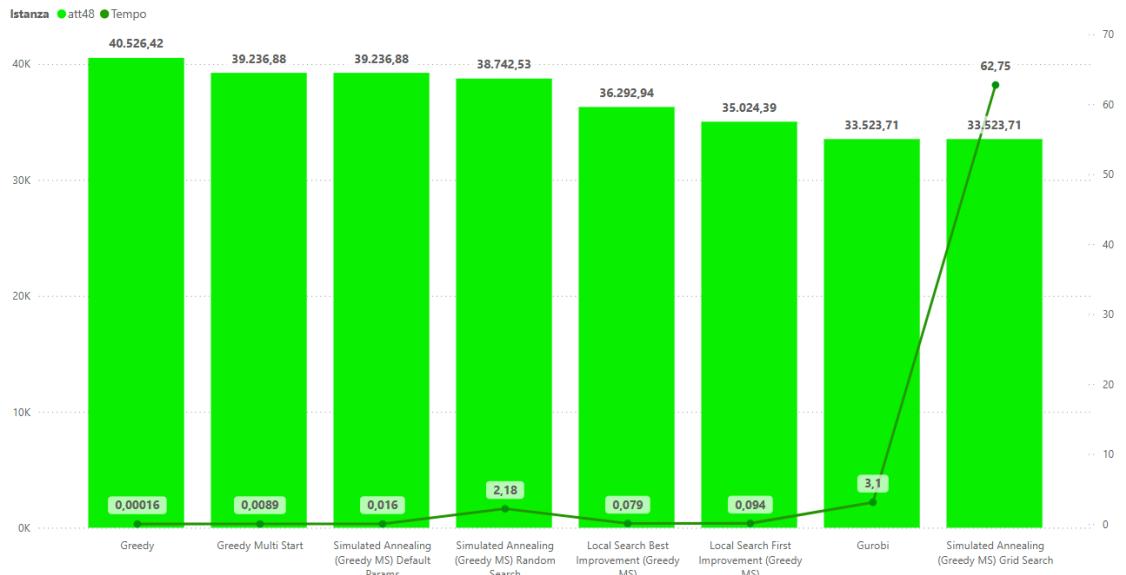
- **Il valore della funzione obiettivo**, che rappresenta la qualità della soluzione in termini di costo del tour;
- **Il tempo di esecuzione**, inteso come indicatore dell'efficienza computazionale.

È importante sottolineare che i tempi di esecuzione costituiscono una stima approssimativa. Le euristiche implementate sono state sviluppate come semplici script Python e non sfruttano parallelismo né ottimizzazioni particolari. Per quanto riguarda Gurobi, non è possibile conoscere nel dettaglio il comportamento interno del risolutore, che potrebbe impiegare meccanismi di parallelizzazione o ottimizzazioni a basso livello. Tuttavia, tutte le esecuzioni sono state effettuate sulla stessa macchina, garantendo che le risorse hardware a disposizione fossero equivalenti in ogni test, così da assicurare una base di confronto coerente. Inoltre, nella valutazione dei tempi di esecuzione delle euristiche di miglioramento locale, è stato incluso anche il tempo necessario alla generazione della soluzione di innesco. Questo perché tale soluzione rappresenta un prerequisito indispensabile per l'avvio della ricerca locale e incide direttamente sul tempo totale richiesto dal processo risolutivo.

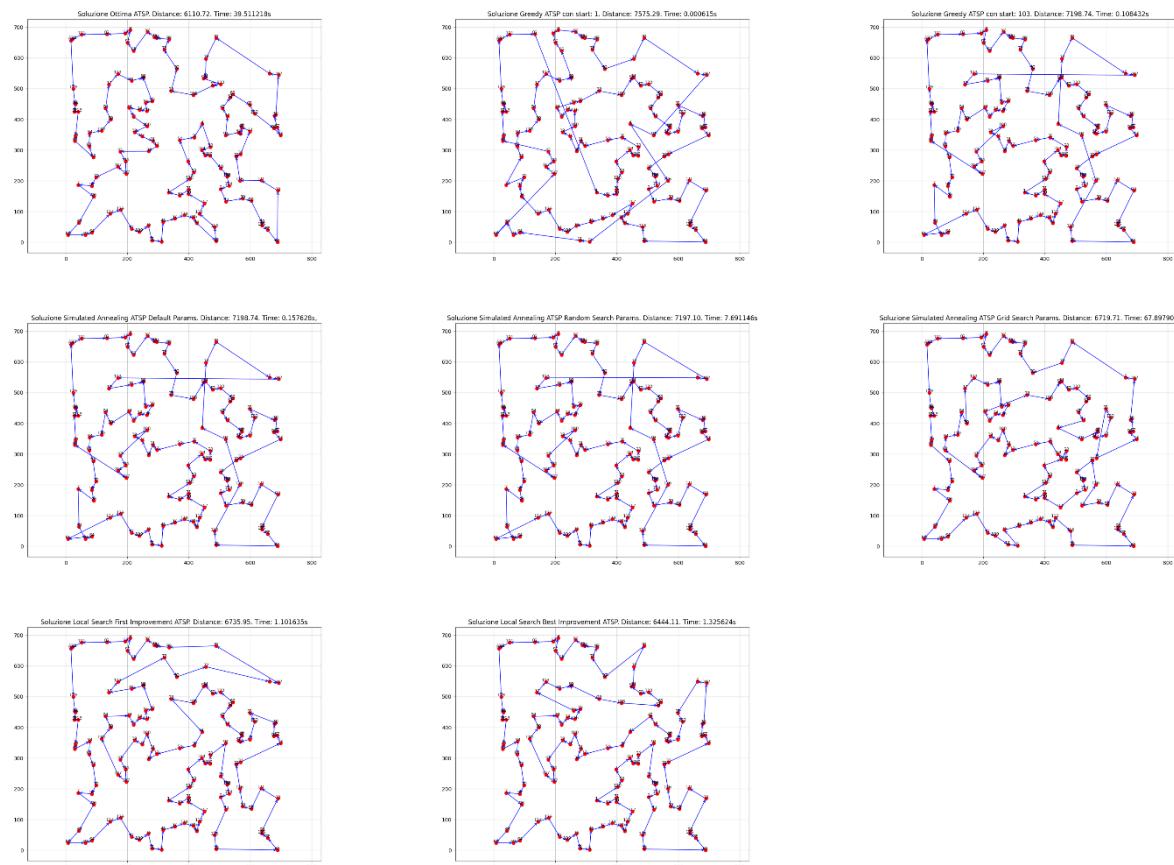
5.1 Att48



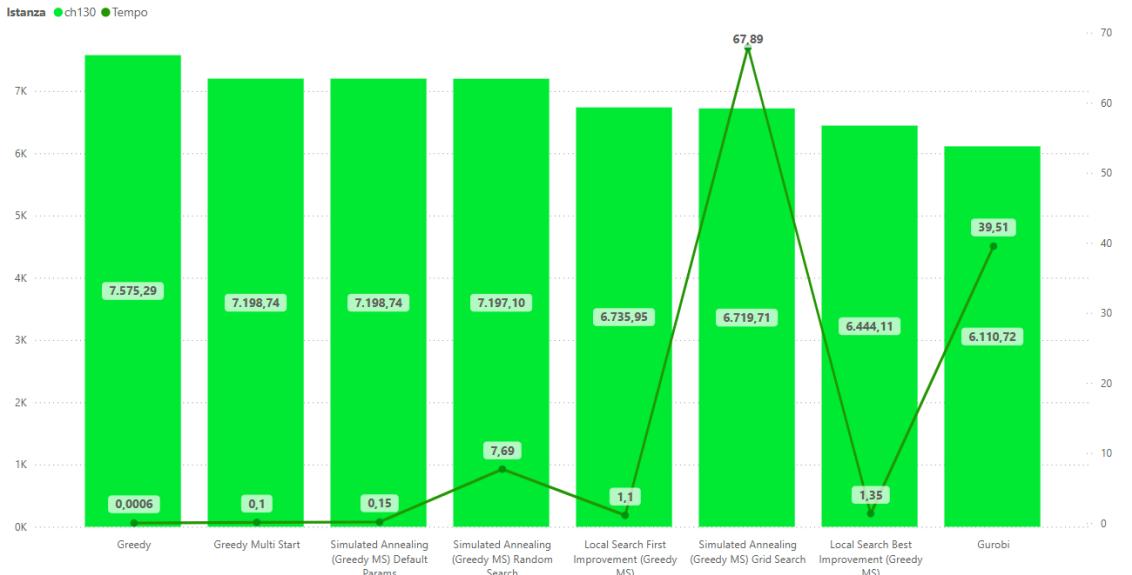
Trend Funzione Obiettivo att48



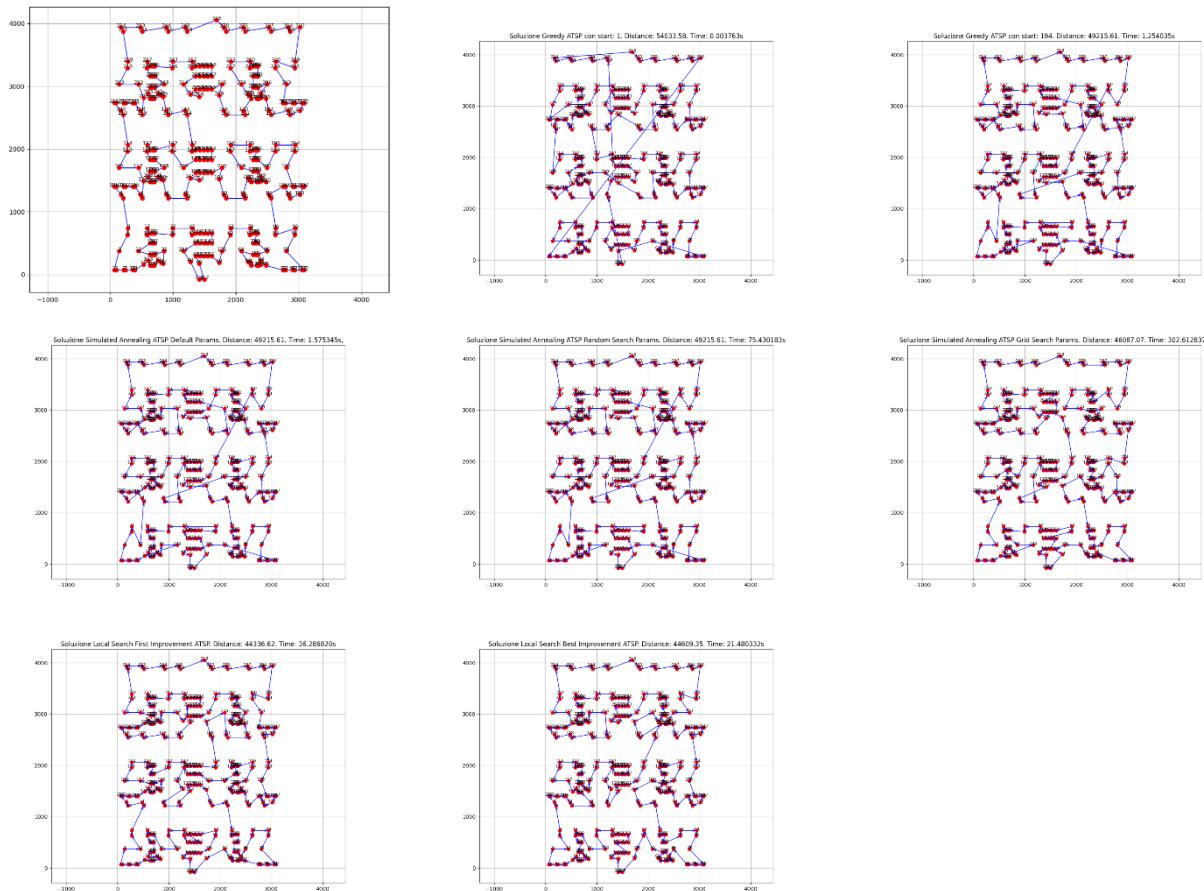
5.2 Ch130



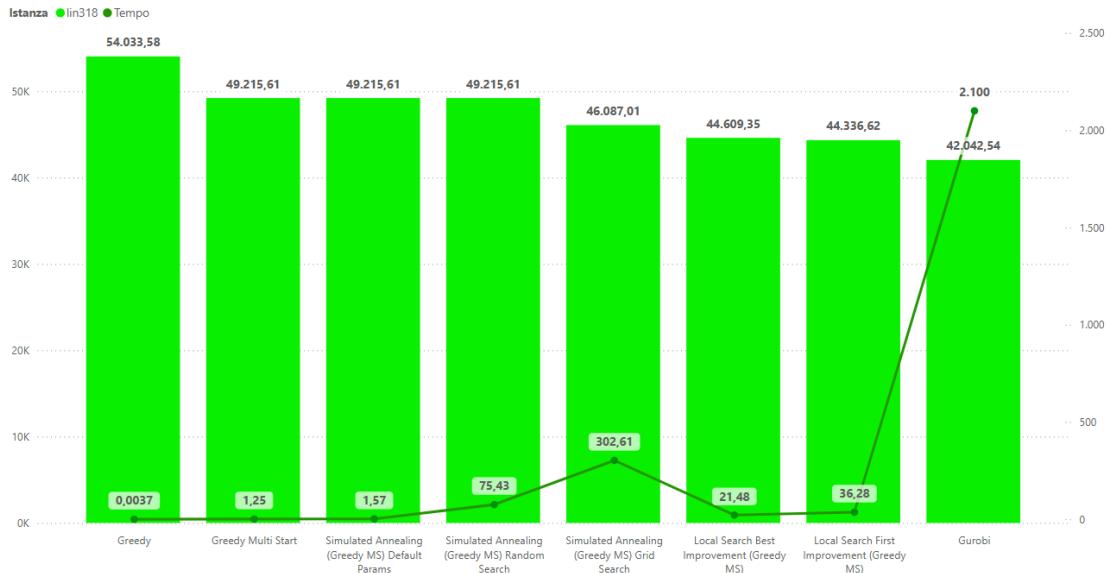
Trend Funzione Obiettivo ch130



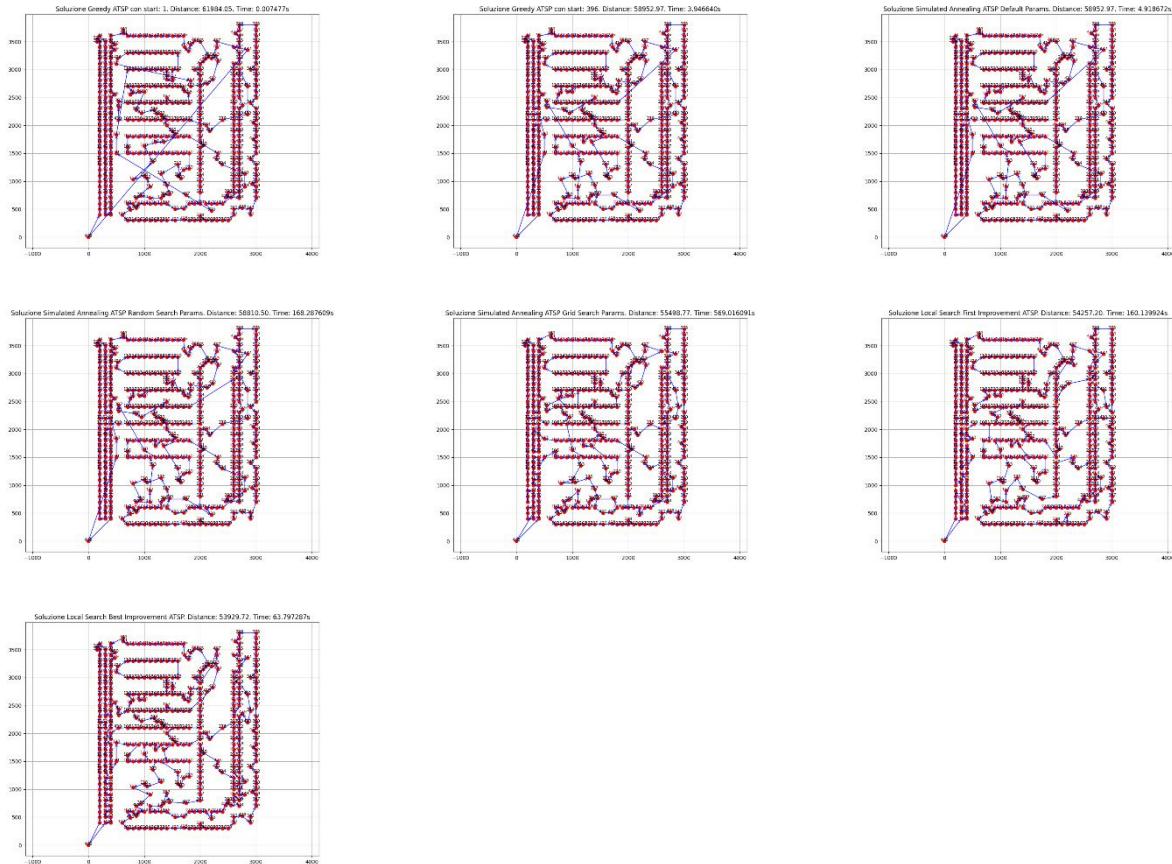
5.3 Lin318



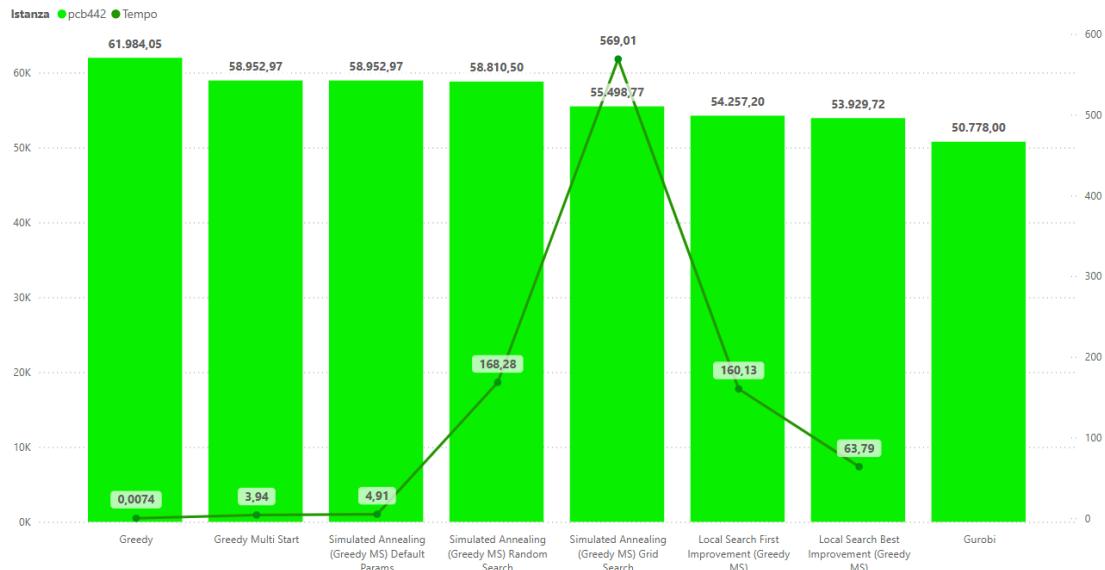
Trend Funzione Obiettivo lin318



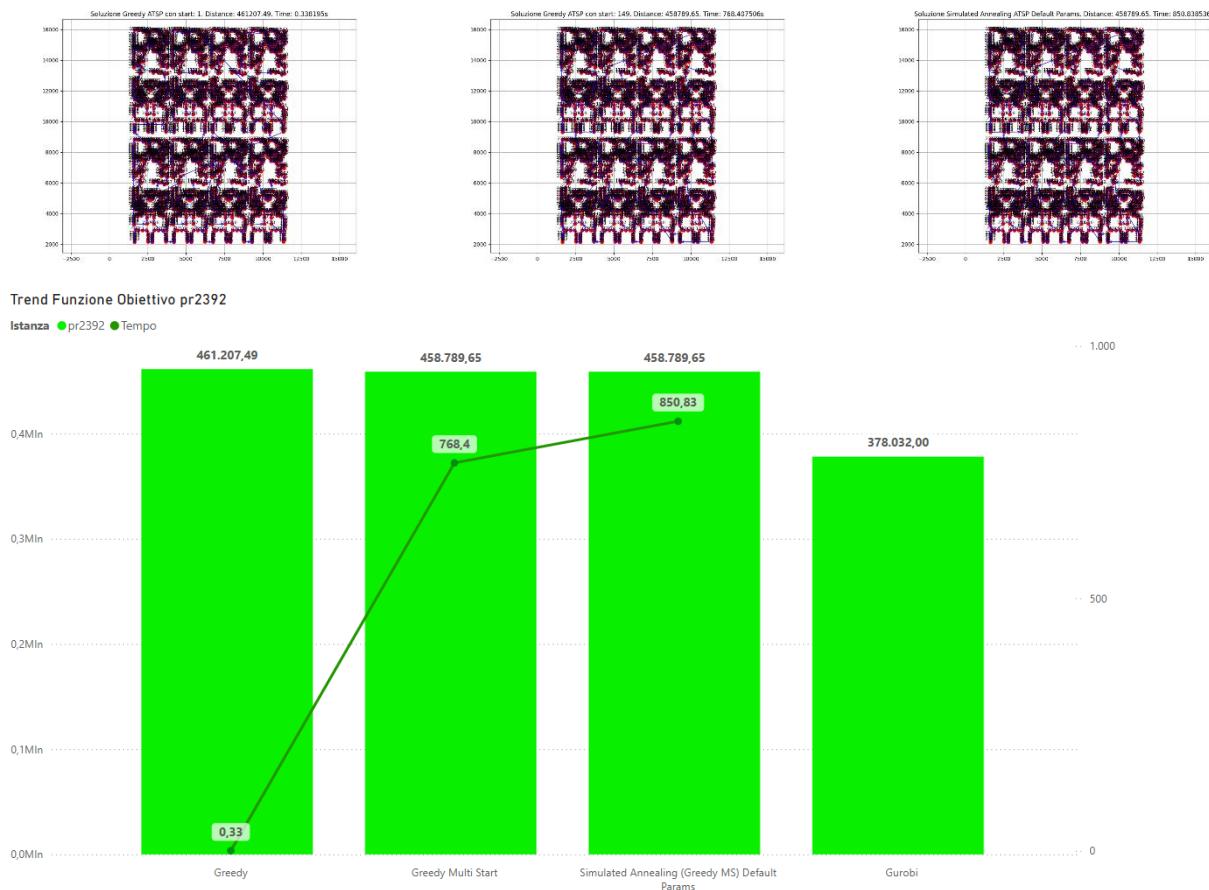
5.4 Pcb442



Trend Funzione Obiettivo pcb442



5.5 Pr2392



6 Conclusioni

In questo capitolo conclusivo vengono aggregati e analizzati i dati sperimentali raccolti su tutte le istanze considerate, ad eccezione dell'ultima, per la quale non è stato possibile ottenere un numero sufficiente di campioni affidabili. L'obiettivo è identificare l'euristica complessivamente più efficace.

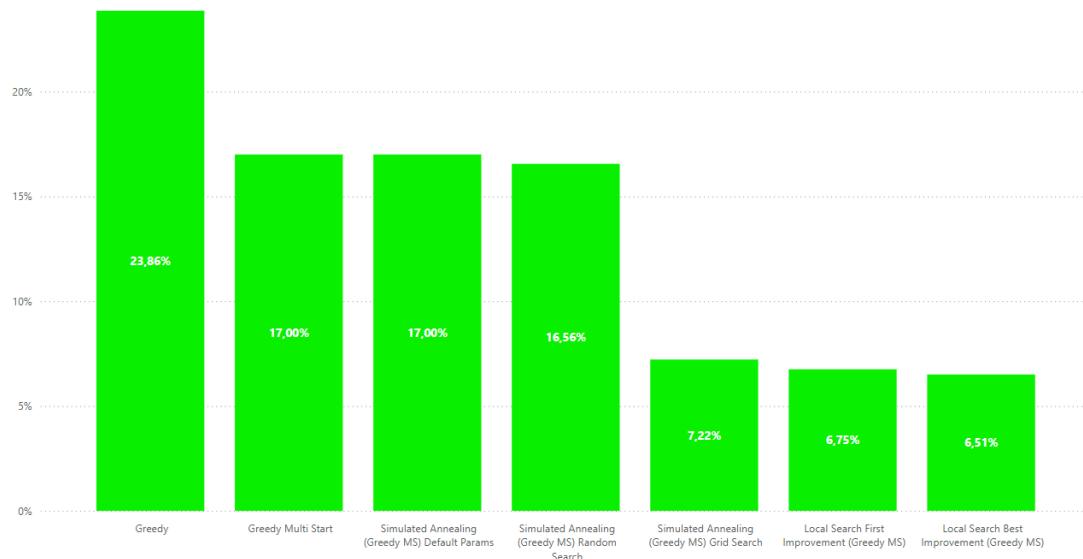
A tal fine, sono state condotte due analisi distinte, basate su due metriche differenti:

- La prima analisi si concentra sul gap percentuale medio ottenuto da ciascun algoritmo, fornendo una misura della qualità della soluzione in termini di scostamento rispetto all'ottimo noto.
- La seconda analisi considera congiuntamente qualità e tempo di esecuzione, tramite una metrica composita definita come:

$$\frac{1}{Gap * \alpha + (1 - \alpha) * Tempo}$$

Dove α è un parametro di bilanciamento tra gap e tempo. Nel nostro caso, è stato fissato a 0.65, in modo da attribuire maggiore peso alla qualità della soluzione rispetto alla velocità di calcolo.

Trend Gap% Medio



Trend Efficienza Pesata Media

