DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI
INGEGNERIA DELL'INFORMAZIONE

ESPLORAZIONE DELLE SOLUZIONI AL PROBLEMA DEL COMMESSO VIAGGIATORE: DAGLI ALGORITMI ESATTI AL RED-BLACK ANT COLONY SYSTEM

MOUHIEDDINE SABIR

Laurea in Ingegneria informatica

Relatore: Domenico Salvagnin
Professore Associato,Università degli Studi di Padova

Commissione Esaminatrice

Presidente: Luca Tonin
Professore Associato, Università degli Studi di Padova

Relatore: Domenico Salvagnin
Professore Associato, Università degli Studi di Padova

Anno Accademico: 2023-2024
Data di laurea: 26/09/2024

LAUREA IN INGENGERIA INFORMATICA

Università degli Studi di Padova
settembre, 2024

Dedico questa tesi ai miei genitori, il cui amore incondizionato, supporto costante e sacrifici infiniti hanno reso possibile questo traguardo. La vostra fiducia in me e il vostro incoraggiamento sono stati il motore di questo percorso. Grazie per essere il mio pilastro e la mia fonte di ispirazione

Ringraziamenti

Desidero innanzitutto esprimere la mia sincera gratitudine al mio relatore, Domenico, per la sua guida e il suo prezioso supporto durante tutto il percorso di questa ricerca. I suoi consigli e il suo incoraggiamento sono stati fondamentali per il completamento di questo lavoro.

A livello personale, vorrei esprimere la mia profonda gratitudine ai miei genitori, per il loro amore, sostegno e pazienza. La vostra fiducia e il vostro incoraggiamento sono stati il mio punto di forza durante tutto questo percorso.

Infine, un grazie di cuore ai miei amici, per la loro amicizia e per aver condiviso con me i momenti più impegnativi e quelli più leggeri di questa esperienza.

A tutti coloro che hanno contribuito, anche indirettamente, al mio percorso accademico, va il mio sincero riconoscimento e apprezzamento.

”

«Traveling — it leaves you speechless, then turns you into a storyteller.»

— Ibn-Battuta, The Travels of Ibn Battutah
(Maghrebi traveller, explorer and scholar)

Sommario

Questa tesi intraprende un'esplorazione approfondita del Problema del Commesso Viaggiatore (TSP), un problema fondamentale nel campo dell'ottimizzazione computazionale, noto per la sua premessa semplice ma risoluzione complessa. Al centro di questa indagine c'è un duplice obiettivo: analizzare la natura intricata del TSP attraverso varie prospettive algoritmiche e introdurre una variante della tecnica di Ant Colony Optimization (ACO), denominata Red-Black Ant Colony System, mirata a migliorare l'efficienza e l'accuratezza della soluzione.

Parole Chiave: Ant Colony System, Travelling Salesman, Metaheuristics, NP-hard problems, Approximation algorithms

Abstract

This thesis embarks on a comprehensive exploration of the Travelling Salesman Problem (TSP), a cornerstone problem in the field of computational optimization, known for its straightfor- ward premise yet complex resolution. At the core of this investigation lies a dual objective: to dissect the intricate nature of TSP through various algorithmic lenses and to introduce a variant of the Ant Colony Optimization (ACO) technique, termed the Red-Black Ant Colony System, aimed at enhancing solution efficiency and accuracy.

Keywords: Ant Colony System, Travelling Salesman, Metaheuristics, NP-hard problems, Approximation algorithms

Indice

Elenco delle figure

Elenco delle tabelle

Acronyms

ACO Ant Colony Optimization *(pp. vii, 32–35)*
ACS Ant Colony System *(pp. 5–7, 12, 35–44)*
ASrank Ant System with Rank-based Pheromone Update *(p. 41)*

BnB Branch and Bound *(pp. 15, 16)*

GA Genetic Algorithm *(pp. 12, 27)*

LK Lin-Kernighan *(pp. 23, 24)*

MMAS Max-Min Ant System *(pp. 33, 41)*
MTZ Miller-Tucker-Zemlin *(p. 3)*

NNS Nearest Neighbor Search *(pp. 20, 21)*
NP Nondeterministic Polynomial time *(pp. vi, 8, 10–12)*

OX Order Crossover *(p. 29)*

P Polynomial time *(pp. vi, 10, 11)*
PMX Partially Mapped Crossover *(p. 29)*

RB-ACS Red-Black Ant Colony System *(pp. viii, ix, 5–7, 42–45)*

SA Simulated Annealing *(pp. 25, 26)*

TS Tabu Search *(pp. 30–32)*
TSP Travelling Salesman Problem *(pp. vi, vii, 1, 2, 4–35, 37, 38, 40–42, 44, 45)*

Capitolo 1

Introduzione

1.1 Informazioni di Base

1.1.1 Definizione e Importanza del TSP

L'importanza del TSP va oltre la sua formulazione apparentemente semplice, ramificandosi in vari campi e applicazioni. In logistica e trasporti, le soluzioni al TSP consentono un routing ottimale in grado di portare a significativi risparmi di costi e miglioramenti di efficienza. Nella produzione, gli algoritmi di risoluzione per TSP vengono utilizzati per ottimizzare i movimenti di trapani nella produzione di circuiti stampati o di bracci robotici nelle linee di assemblaggio per ridurre i tempi di produzione. Oltre a questi, gli algoritmi per TSP vengono impiegati nel sequenziamento del DNA, nell'astronomia e persino nella creazione di opere d'arte, mostrando la sua versatilità e importanza nel risolvere problemi del mondo reale.

1.1.2 Panoramica Storica

Il Travelling Salesman Problem (TSP) rappresenta una delle sfide più studiate nella matematica computazionale e nell'ottimizzazione, con una storia che si intreccia con lo sviluppo della ricerca matematica e operativa. La sua evoluzione dalle prime esplorazioni concettuali a un problema cruciale nella progettazione degli algoritmi è una testimonianza della sua complessità e della sua ampia applicabilità. Lo studio del TSP può essere fatto risalire ai lavori di Sir William Rowan Hamilton e Thomas Penyngton Kirkman nel XIX secolo, incentrati sui cicli e sui percorsi hamiltoniani. Karl Menger, negli anni '20, ha introdotto il "Problema del Messaggero", gettando le basi per la formulazione moderna del TSP e sottolineando la ricerca di un routing efficiente.

L'avvento dei computer e delle tecniche di ricerca operativa ha segnato una fase significativa nella ricerca sul TSP. I primi pionieri come George Dantzig e Delbert Ray Fulkerson hanno applicato la programmazione lineare e i metodi dei piani di taglio, illustrando la complessità computazionale del problema e il suo potenziale per le applicazioni pratiche.

Negli anni '40, il TSP ha iniziato a trovare rilevanza in agricoltura e statistica, con i ricercatori che utilizzavano gli algoritmi risolutivi del problema per ottimizzare i processi di rilevamento e raccolta dei dati, mostrando la versatilità del TSP oltre la matematica teorica.

Negli ultimi decenni, l'esplorazione di algoritmi euristici e bio-ispirati, in particolare il lavoro di Marco Dorigo sull'ottimizzazione combinatoria mediante colonie di formiche, ha aperto nuove vie per risolvere il TSP, evidenziando l'innovazione continua nell'affrontare la sua NP-hardness.

1.2 Il Problema

Il Problema del Viaggiatore: Data una serie di città e le distanze tra ogni coppia di città, trovare il percorso più breve possibile che visiti ogni città esattamente una volta e torni alla città originale.

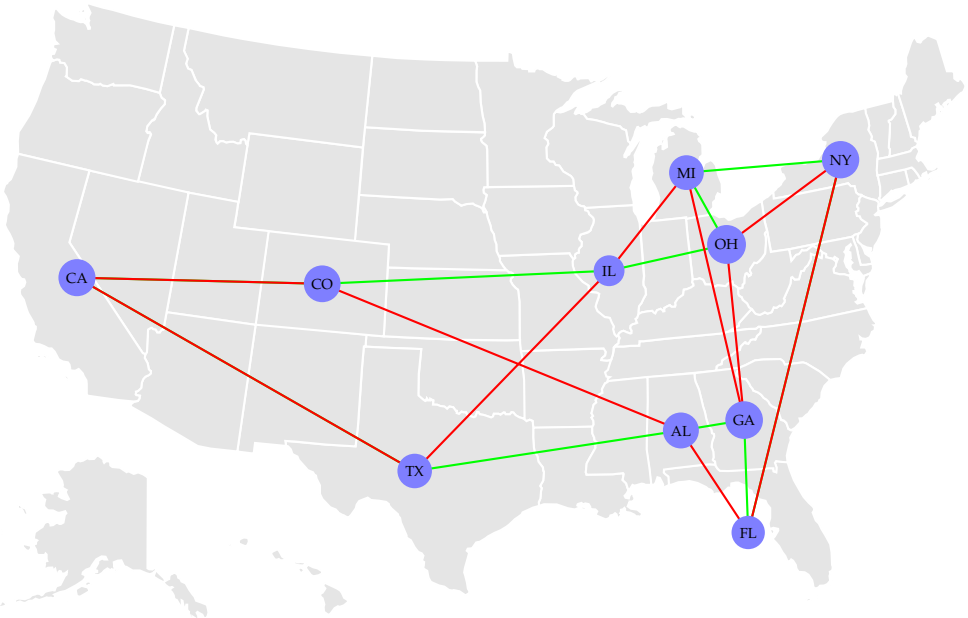


Figura 1.1: Due possibili soluzioni per il (TSP) in un'istanza semplificata degli Stati Uniti, uno più corto (in verde) e uno più lungo (in rosso).

1.2.1 Formulazione Matematica

Dato un grafo $G = (V, E)$, dove V è l'insieme dei vertici (città) e E è l'insieme degli archi (percorsi tra le città), con ogni arco $(i, j) \in E$ assegnato un peso w_{ij} che rappresenta il costo di viaggio dalla città i alla città j, il TSP cerca un ciclo hamiltoniano di peso minimo. La

funzione obiettivo da minimizzare è:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} w_{ij} x_{ij},$$

soggetta ai vincoli:

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \quad \forall j \in V,$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, \quad \forall i \in V,$$

insieme ai vincoli di eliminazione dei sottoitinerari, assicurando che ogni città sia visitata esattamente una volta e il tour sia ciclico.

1.2.2 Formulazioni di Programmazione Lineare Intera

1.2.2.1 Formulazione di Miller–Tucker–Zemlin

La formulazione Miller-Tucker-Zemlin (MTZ) introduce variabili ausiliarie u_i per l'ordinamento, insieme alle variabili binarie x_{ij}:

Obiettivo:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij},$$

Soggetto a:

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \quad \forall j,$$

$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, \quad \forall i,$$

$$u_i - u_j + 1 \leq (n-1)(1 - x_{ij}), \quad \forall 2 \leq i \neq j \leq n,$$

$$2 \leq u_i \leq n, \quad \forall 2 \leq i \leq n.$$

Questa formulazione impone un singolo tour attraverso variabili ausiliarie di ordinamento, prevenendo efficacemente i sottoitinerari.

1.2.2.2 Formulazione di Dantzig–Fulkerson–Johnson

La formulazione DFJ, nota per la sua efficienza, introduce direttamente i vincoli di eliminazione dei sottoitinerari:

Obiettivo:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij},$$

Soggetto a:

$$\sum_{i=1, i\neq j}^{n} x_{ij} = 1, \quad \forall j,$$

$$\sum_{j=1, j\neq i}^{n} x_{ij} = 1, \quad \forall i,$$

$$\sum_{i\in S, j\notin S} x_{ij} \geq 1, \quad \forall S \subset V, S \neq \emptyset, S \neq V.$$

Questa formulazione combatte i sottoitinerari assicurando che almeno un arco esca da ogni sottinsieme di vertici, garantendo un singolo tour connesso.

1.2.3 Varianti e Applicazioni

Il TSP si adatta a un'ampia gamma di sfide pratiche, dimostrando la sua versatilità. Le varianti chiave includono:

- **TSP Simmetrico vs. Asimmetrico (sTSP vs. aTSP):** sTSP implica che il peso di un arco tra due nodi è uguale indipendentemente dal verso di percorrenza, mentre aTSP consente pesi distinti, riflettendo la complessità delle rotte del mondo reale.
- **TSP Euclideo:** Questa variante posiziona le città in un piano euclideo, enfatizzando le distanze dirette e in linea retta, fondamentale per la navigazione dei droni e la pianificazione delle infrastrutture.
- **TSP con Finestre Temporali:** Aggiunge specifici intervalli di tempo entro i quali ogni città deve essere visitata, cruciale per la pianificazione nei servizi di consegna e nelle operazioni di manutenzione.
- **Problema di Routing dei Veicoli (VRP):** Un'estensione del TSP che coinvolge più veicoli, centrale per ottimizzare la logistica della flotta e le reti di distribuzione.

L'applicazione del TSP e delle sue varianti si estende a vari campi, mostrando la sua utilità nell'affrontare complessi problemi di ottimizzazione:

- **Ottimizzazione Logistica:** Il TSP euclideo e il VRP sono essenziali per ridurre il costo le rotte di consegna e migliorare l'efficienza logistica.
- **Produzione:** Il TSP aiuta a ottimizzare i processi produttivi, riducendo i tempi di fermo e migliorando i flussi di lavoro complessivi.
- **Pianificazione e Scheduling:** Il TSP con Finestre Temporali garantisce operazioni tempestive, ottimizzando l'allocazione delle risorse per i servizi con requisiti di tempistica critici.

- **Progettazione di Reti:** Gli algoritmi TSP facilitano lo sviluppo di reti di telecomunicazioni e trasporti economiche ed efficienti.

- **Genomica:** In genetica, il TSP supporta i progressi nel sequenziamento del genoma, offrendo informazioni sulle strutture e sui processi biologici complessi.

1.3 Obiettivi della Tesi

Questa tesi mira ad esplorare il Problema del Viaggiatore (TSP) sia da un punto di vista teorico che pratico. L'attenzione è rivolta all'esame del Red-Black Ant Colony System RB-ACS, una variante dell'algoritmo Ant Colony System Ant Colony System (ACS), con alcune modifiche, per comprenderne l'efficacia nell'affrontare le sfide del TSP. Lo studio ha due obiettivi: migliorare la comprensione del TSP nell'ottimizzazione computazionale e valutare le prestazioni di RB-ACS modificato rispetto alle soluzioni convenzionali.

1.3.1 Obiettivi Principali

La tesi è guidata dai seguenti obiettivi:

- Esaminare il corpus esistente di lavori sul TSP, raccogliendo le sue basi matematiche, l'evoluzione e l'insieme di strategie precedentemente ideate per la sua risoluzione.

- Dettagliare il funzionamento del RB-ACS, enfatizzandone le basi teoriche, la meccanica operativa e le sue distinzioni rispetto agli approcci più tradizionali.

- Effettuare un confronto sperimentale sistematico del RB-ACS contro gli algoritmi standard di risoluzione del TSP su una gamma di insiemi di problemi, mirando a discernerne l'efficacia operativa e la scalabilità.

- Identificare gli scenari pratici in cui l'algoritmo modificato potrebbe essere applicato, mostrando la sua rilevanza e i potenziali benefici in contesti tangibili.

1.3.2 Vincoli Teorici

L'esplorazione all'interno di questa tesi è adattata per adattarsi alle risorse computazionali disponibili, concentrandosi su istanze di TSP che bilanciano la presentazione di una sfida notevole e la fattibilità dell'analisi entro i vincoli della nostra configurazione computazionale. Un aspetto notevole di questa ricerca riguarda la reimplementazione in Rust di tutti gli algoritmi discussi, a causa della non disponibilità del codice sorgente originale per alcuni di questi algoritmi. Questa necessità mette in luce una possibile variazione nelle metriche di prestazione, che potrebbero non allinearsi completamente con quelle derivate dalle implementazioni originali degli autori. Tali differenze potrebbero derivare da vari fattori, tra cui le caratteristiche intrinseche di Rust come linguaggio di programmazione e

le complessità coinvolte nel tradurre la logica algoritemica complessa in codice. Di conseguenza, mentre si mira a fornire un'analisi comparativa delle prestazioni algoritmiche, è imperativo considerare questi fattori di reimplementazione come potenziali variabili che influenzano i risultati.

Inoltre, mentre si riconosce il potenziale di applicazioni del mondo reale degli algoritmi in studio, gli studi approfonditi di casi che vadano oltre l'esplorazione teorica e la sperimentazione computazionale esulano dall'ambito di questa tesi di laurea. L'attenzione principale rimane sulle basi teoriche e sulla valutazione computazionale delle soluzioni TSP.

1.3.3 Considerazioni Pratiche

La praticità di questa tesi è influenzata da diverse limitazioni che guidano il suo approccio sperimentale e la metodologia di valutazione:

- La valutazione delle prestazioni del RB-ACS viene condotta attraverso un sottoinsieme attentamente scelto di istanze di TSP. Queste istanze mirano a fornire una rappresentazione diversificata ma non esaustiva di potenziali scenari problematici, evidenziando il limite intrinseco nella cattura dell'intero spettro delle configurazioni di TSP.

- A causa di limiti di risorse di calcolo, la scalabilità dell'algoritmo proposto viene testata in un ambiente vincolato. Questa limitazione potrebbe ostacolare l'estrapolazione dei risultati delle prestazioni a istanze di TSP significativamente più grandi o complesse.

- La tesi considera le implicazioni pratiche delle sue scoperte in un contesto teorico e computazionale, senza addentrarsi in test sul campo o in studi di casi specifici per applicazioni.

1.4 Struttura della Tesi

1.4.1 Panoramica dei Capitoli

Il Capitolo 1 introduce il Problema del Viaggiatore (TSP), coprendo cosa sia, perché sia importante e un po' della sua storia. Esso delinea le principali domande a cui questa tesi mira a rispondere e delinea brevemente i metodi utilizzati. Il Capitolo 2 analizza il quadro generale della complessità computazionale, concentrandosi sulla famosa questione P vs. NP, per aiutare a capire perché risolvere il TSP può essere così impegnativo. La discussione prosegue nei Capitoli 3 e 4 per esplorare i diversi modi di affrontare il TSP. Il Capitolo 3 esamina le soluzioni esatte mentre il Capitolo 4 discute i metodi euristici e meteuristici, incluse le ben note strategie come gli algoritmi genetici e il simulated annealing. Il Capitolo 5 si concentra sul ACS, spiegando come funziona, perché è ispirato dalla natura e come si

applica alla risoluzione del TSP. Quindi, il Capitolo 6 introduce una variante dell'ACS, il RB-ACS. Questo capitolo spiega come è stato sviluppato, le nuove idee che porta e come i test dimostrano che potrebbe offrire un vantaggio nella risoluzione delle istanze di TSP. Il Capitolo 7 presenta i risultati sperimentali ottenuti. Infine, il Capitolo 8 conclude la tesi riassumendo i punti salienti di questa tesi. Questa tesi mira a contribuire alla conversazione sulla risoluzione del TSP, cercando di colmare il divario tra le sfide teoriche e le soluzioni del mondo reale, e aprire la porta a ulteriori indagini sui problemi di ottimizzazione.

Capitolo 2

Quadro Teorico: Complessità ed efficienza

2.1 Complessità Computazionale

La complessità computazionale si riferisce allo studio dei requisiti di risorse degli algoritmi, concentrandosi principalmente sulle risorse di tempo e spazio in funzione della dimensione dell'input. Questo campo categorizza i problemi in base alle risorse minime necessarie per risolverli, fornendo un quadro teorico per comprendere la difficoltà intrinseca dei problemi computazionali e l'efficienza degli algoritmi progettati per risolverli. [1]

2.1.1 Complessità Temporale

La complessità temporale è una misura della quantità di tempo di calcolo che un algoritmo impiega per completarsi in funzione della lunghezza dell'input. Viene spesso espressa utilizzando la notazione Big O, che descrive il limite superiore del tasso di crescita del tempo di esecuzione [2]. Comprendere la complessità temporale di un algoritmo è cruciale per prevederne la scalabilità e la fattibilità nelle applicazioni pratiche, soprattutto per i problemi noti per essere computazionalmente intensivi come il TSP.

Il TSP, essendo NP-hard, non ha una soluzione in tempo polinomiale nota che possa risolvere in modo efficiente tutte le istanze del problema. L'approccio a forza bruta per il TSP, ad esempio, ha una complessità temporale fattoriale ($O(n!)$), rendendolo computazionalmente irrealizzabile anche per istanze del problema di dimensioni moderatamente grandi. Ciò ha portato all'esplorazione di vari algoritmi euristici e meteuristici che mirano a soluzioni accettabili in tempo polinomiale ($O(n^k)$), dove k è una costante, scambiando precisione per efficienza e applicabilità pratica. [3]

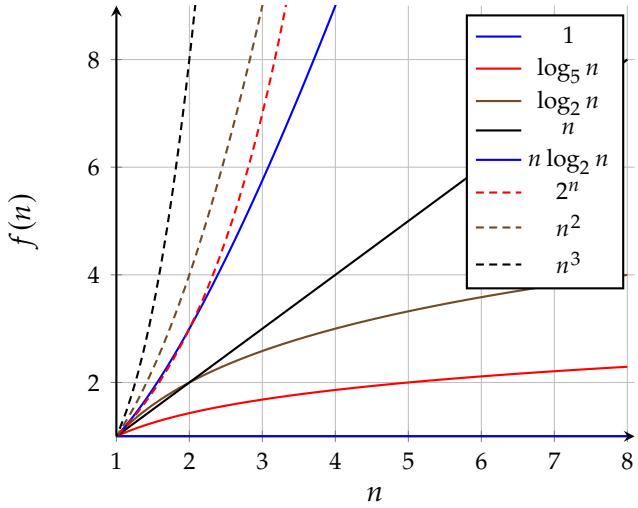


Figura 2.1: Tasso di crescita delle diverse complessità

Tabella 2.1: Numero di possibili permutazioni per numero di città

Numero città	Numero Permutazioni
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800
11	39,916,800
12	479,001,600
13	6,227,020,800
14	87,178,291,200
15	1,307,674,368,000
16	20,922,789,888,000
17	355,687,428,096,000
18	6,402,373,705,728,000
19	121,645,100,408,832,000
20	2,432,902,008,176,640,000
25	15,511,210,043,330,985,984,000,000

2.1.2 Complessità Spaziale

La complessità spaziale misura la quantità totale di memoria di cui un algoritmo necessita in funzione della dimensione dell'input. Come la complessità temporale, la complessità spaziale è cruciale per valutare l'efficienza di un algoritmo, in particolare in scenari in cui le risorse di memoria sono limitate. Nel contesto del TSP e di problemi di ottimizzazione simili, la complessità spaziale di un algoritmo può influire significativamente sulla sua usabilità in applicazioni del mondo reale, dove le soluzioni spesso devono essere calcolate in tempo reale o su hardware con capacità di memoria limitate. [4].

Ad esempio, gli approcci di programmazione dinamica per il TSP, come l'algoritmo di Held-Karp, offrono una complessità temporale più favorevole rispetto alla forza bruta ($O(n^2 2^n)$) ma a costo di una complessità di spazio esponenziale ($O(n^2)$). Tali compromessi tra complessità temporale e spaziale sono considerazioni centrali nella progettazione degli algoritmi, soprattutto quando si sviluppano nuovi algoritmi destinati a istanze di problemi su larga scala, come quelle riscontrate in applicazioni di logistica e routing.

2.2 Il Problema P vs. NP

Il problema P vs NP rappresenta una delle questioni aperte più significative nel campo dell'informatica teorica. Si tratta di stabilire se ogni problema la cui soluzione può essere verificata in modo efficiente (in tempo polinomiale) da un computer possa anche essere risolto altrettanto velocemente (sempre in tempo polinomiale). La classe P include problemi che possono essere risolti rapidamente, mentre la classe NP comprende problemi per i quali una soluzione data può essere verificata rapidamente. [5].

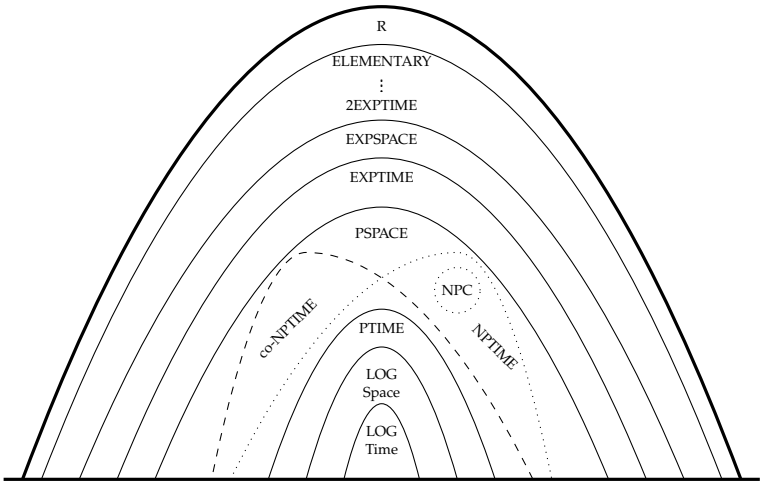


Figura 2.2: Classificazione dei problemi computazionali

2.2.1 Definizione e Rilevanza

Formalmente, un problema appartiene alla classe P se può essere risolto in tempo polinomiale, il che significa che la quantità di tempo necessaria per risolvere il problema cresce in modo polinomiale con la dimensione dell'input. La classe NP , d'altra parte, comprende i problemi per i quali una soluzione data può essere verificata in tempo polinomiale. Il problema P vs. NP chiede essenzialmente se queste due classi sono uguali; cioè, se ogni problema che può essere verificato rapidamente può anche essere risolto rapidamente.

La rilevanza del problema P vs. NP risiede nelle sue implicazioni per un vasto campo di discipline, dalla crittografia e la progettazione di algoritmi ai processi decisionali e oltre. Una prova che P è uguale a NP potrebbe potenzialmente rivoluzionare i campi della matematica e dell'informatica, consentendo soluzioni efficienti a una moltitudine di

problemi complessi attualmente considerati intrattabili. Al contrario, provare che P non è uguale a NP formalizzerebbe la difficoltà computazionale intrinseca di questi problemi.

2.2.2 Implicazioni per il TSP

La classificazione del TSP come NP-hard sottolinea la complessità e le sfide computazionali associate al trovare una soluzione esatta per grandi istanze del problema. Questa classificazione ha implicazioni significative sia per la ricerca teorica che per le applicazioni pratiche nel campo. Implica che a meno che P non sia uguale a NP non ci si dovrebbe aspettare di scoprire un algoritmo in grado di risolvere rapidamente (in tempo polinomiale) e accuratamente ogni istanza del TSP. Questa consapevolezza ha spostato il focus di gran parte della ricerca sul TSP verso lo sviluppo di algoritmi euristici e meteuristici, che mirano a trovare soluzioni sufficientemente buone in un lasso di tempo ragionevole, piuttosto che inseguire l'elusivo obiettivo di una soluzione esatta per tutte le possibili istanze. [6].

Inoltre, le implicazioni del fatto che il TSP sia NP-hard si estendono oltre lo sviluppo degli algoritmi. Influenza il modo in cui i problemi vengono modellati in scenari pratici in cui sorgono problemi simili al TSP, come la logistica e il routing, la progettazione di reti e la pianificazione. In queste applicazioni, l'enfasi è spesso rivolta al trovare soluzioni che si avvicinino all'ottimale ma possano essere ottenute molto più rapidamente di quanto permetterebbe un algoritmo esatto. Questo approccio consente all'industria di raggiungere efficienza e risparmi di costi anche quando si affrontano problemi di routing e pianificazione complessi.

Pertanto, le implicazioni della classificazione del TSP risuonano sia nello studio accademico della teoria degli algoritmi che nelle considerazioni pratiche sull'applicazione di queste teorie in situazioni del mondo reale.

2.3 Complessità del TSP

La dimostrazione che il TSP è NP -hard è stata presentata per la prima volta da Richard M. Karp nel 1972. Il lavoro di Karp, parte di un articolo che identificava 21 problemi NP-complete, ha gettato le basi per comprendere i limiti computazionali delle soluzioni algoritmiche per il TSP e problemi simili. Dimostrando che la versione decisionale del TSP (determinare se esiste un tour di una data lunghezza o inferiore) è NP-complete, Karp ha dimostrato che la versione di ottimizzazione del TSP, in cui l'obiettivo è trovare il tour più breve possibile, è NP -hard.

La dimostrazione di Karp impiega una tecnica nota come riduzione, in cui un problema NP-complete noto viene trasformato in un'istanza del TSP in tempo polinomiale. Questo metodo illustra che se un algoritmo in tempo polinomiale potesse risolvere il TSP, potrebbe, per estensione, risolvere tutti i problemi in NP, una proposizione che rimane non verificata.

Algoritmo	Complessità Temporale - Spaziale	Descrizione	Vantaggi	Svantaggi
Held-Karp	$O(n^2 2^n)$ - $O(n^2)$	Algoritmo esatto che utilizza la programmazione dinamica	Fornisce soluzione ottimale	Inattuabile per grandi n
Nearest Neighbor	$O(n^2)$ - $O(n)$	Algoritmo euristico greedy	Semplice e veloce	Spesso lontano dall'ottimale
Christofides	$O(n^3)$ - $O(n^2)$	Algoritmo di approssimazione con rapporto di 3/2	Migliore approssimazione per TSP metrico	Ancora non esatto
Algoritmi Genetici	$O(n^2)$ - $O(n)$	Algoritmo metaeuristico ispirato alla selezione naturale	Flessibile e adattabile	Nessuna garanzia di soluzione ottimale

Tabella 2.2: Confronto tra diversi algoritmi per TSP

2.3.1 Impatto sulla progettazione degli algoritmi

L'NP-hardness del TSP ha un profondo impatto sulla progettazione degli algoritmi per risolverlo, favorendo lo sviluppo di varie strategie volte a superare gli ostacoli computazionali:

- **Algoritmi esatti:** Nonostante l'NP-hardness del TSP, sono stati ideati algoritmi esatti, come l'algoritmo di Held-Karp. Questi algoritmi garantiscono una soluzione ottimale, ma lo fanno a un costo computazionale che aumenta esponenzialmente con le dimensioni del problema, rendendoli impraticabili per istanze di grandi dimensioni.

- **Approcci euristici:** Per affrontare istanze di TSP più grandi, vengono impiegati algoritmi euristici. Questi algoritmi non garantiscono una soluzione ottimale, ma possono spesso trovare buone soluzioni in una frazione del tempo richiesto dai metodi esatti. Esempi includono l'algoritmo Nearest Neighbor e l'algoritmo di Christofides.

- **Algoritmi meteuristici:** Per istanze ancora più complesse, gli approcci meteuristici, come il Genetic Algorithm Genetic Algorithm (GA) e l'Ant Colony System ACS, offrono strategie flessibili che esplorano lo spazio delle soluzioni in modo più ampio, mirando a sfuggire agli ottimi locali e avvicinarsi più efficacemente agli ottimi globali. Queste tecniche attingono ispirazione dai processi naturali e hanno il vantaggio dell'adattabilità a diverse istanze di problema.

CAPITOLO 3

ALGORITMI ESATTI PER IL TSP

Questo capitolo fornisce un esame approfondito degli algoritmi esatti per risolvere il TSP, incorporando pseudocodice dettagliato per chiarire i principi operativi di ciascun metodo. Attraverso questi algoritmi, esploriamo il panorama computazionale della ricerca di soluzioni ottimali al TSP.

3.1 Panoramica degli Algoritmi Esatti

Gli algoritmi esatti per il TSP sono caratterizzati dalla loro capacità di trovare invariabilmente la soluzione ottimale, sebbene con costi computazionali che possono diventare proibitivi man mano che il numero di città aumenta. Questi algoritmi servono come base teorica e pratica per comprendere i limiti della risoluzione computazionale del TSP.

3.1.1 Metodo a Brute-Force

Il metodo a forza bruta esamina sistematicamente ogni possibile tour per identificare quello con la distanza totale più corta. Nonostante la sua semplicità, la crescita esponenziale dei calcoli lo rende impraticabile di dimensione non banale.

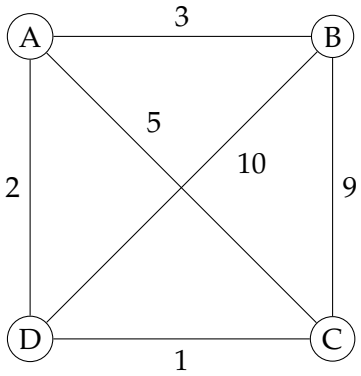
La complessità computazionale di questo metodo è $O(n!)$, che riflette il numero fattoriale di tour che devono essere valutati.

3.1.2 Algoritmo di Bellman-Held-Karp

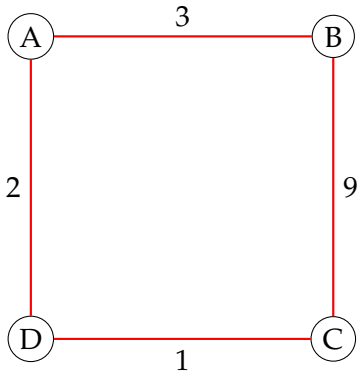
L'algoritmo di Bellman-Held-Karp rappresenta un significativo avanzamento nelle soluzioni esatte del TSP sfruttando la programmazione dinamica per ridurre l'onere computazionale. Questo approccio calcola il percorso più breve dividendo il problema in sottoproblemi più piccoli e memorizzando i risultati di questi sottoproblemi per evitare calcoli ridondanti sfruttando la programmazione dinamica.

Algorithm 1 TSP Brute Force

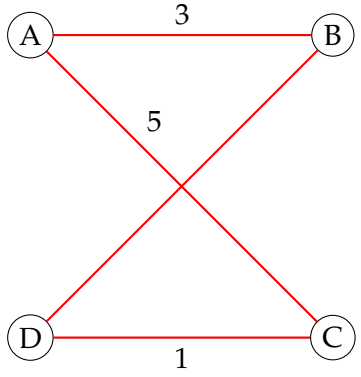
```
1:  procedure BruteForceTSP(cities)
2:      min_distance ← ∞
3:      min_tour ← ∅
4:      for all tours t of cities do
5:          distance ← TourDistance(t)
6:          if distance < min_distance then
7:              min_distance ← distance
8:              min_tour ← t
9:          end if
10:     end for
11:     return min_tour
12: end procedure
```



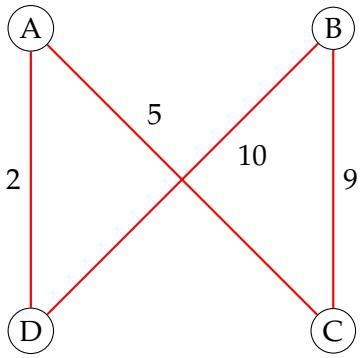
(a) Grafo di partenza con quattro città



(b) Percorso ottimale con distanza totale 15



(c) Percorso alternativo con distanza totale 19



(d) Percorso alternativo con distanza totale 26

Figura 3.1: Brute force TSP con quattro città

Algorithm 2 Algoritmo di Bellman-Held-Karp

```
1: procedure BellmanHeldKarp(cities)
2:     Crea una tabella distances per memorizzare i percorsi più brevi
3:     Inizializza tutte le voci in distances a ∞
4:     distances[1][{1}] ← 0                                  ▷ Città di partenza
5:     for m = 2 to |cities| do
6:         for all sottoinsiemi S di cities di dimensione m contenenti 1 do
7:             for all j ∈ S, j ≠ 1 do
8:                 distances[j][S] ← min_{k≠j, k∈S}(distances[k][S \ {j}] + distance(k, j))
9:             end for
10:        end for
11:    end for
12:    return min_j(distances[j][cities] + distance(j, 1))
13: end procedure
```

L'algoritmo di Bellman-Held-Karp opera con una complessità temporale di $O(n^2 \cdot 2^n)$ 3.2, rendendolo significativamente più efficiente del metodo a brute-force per istanze di problema di dimensioni moderate. Tuttavia, la componente esponenziale della sua complessità limita ancora la sua applicabilità pratica a istanze relativamente piccole di TSP.

3.1.2.1 Branch and Bound

Branch and Bound (Branch and Bound (BnB)) è una tecnica algoritmica generale che può essere applicata a vari problemi di ottimizzazione combinatoria, incluso il TSP. Esplora sistematicamente lo spazio delle soluzioni dividendolo in sottoinsiemi più piccoli (branching) e utilizzando limiti per eliminare i sottoinsiemi che non contengono la soluzione ottimale, riducendo così lo spazio di ricerca.

L'efficienza di BnB per il TSP dipende in modo significativo dalla funzione di limite utilizzata per stimare il limite inferiore delle soluzioni parziali. Un limite efficace può portare a riduzioni sostanziali dello spazio di ricerca, consentendo all'algoritmo di trovare la soluzione ottimale più rapidamente rispetto ai metodi di ricerca esaustiva. Tuttavia, la sua complessità temporale peggiore rimane $O(n!)$ 3.2, il che significa che per istanze molto grandi del problema, anche questo approccio sofisticato potrebbe non essere pratico.

3.2 Analisi degli Algoritmi Esatti

3.2.1 Fattibilità Computazionale

Gli algoritmi esatti per il TSP, come il metodo a forza bruta, la programmazione dinamica e l'algoritmo di Bellman-Held-Karp, offrono garanzie teoriche per trovare il tour ottimale. Tuttavia, la loro fattibilità computazionale è notevolmente compromessa all'aumentare del numero di città. Il tasso di crescita fattoriale della complessità del metodo a forza bruta

Algorithm 3 BnB per il TSP

```
1:  procedure BranchAndBoundTSP(cities)
2:      Inizializza una coda di priorità Q con una soluzione parziale che parte dalla prima città
3:      min_distance ← ∞
4:      while Q non è vuota do
5:          Prendi la soluzione parziale con il limite inferiore più basso da Q
6:          if rappresenta un tour completo then
7:              if il suo costo è inferiore a min_distance then
8:                  Aggiorna min_distance e registra il tour
9:              end if
10:         else
11:             Dividi aggiungendo una città alla soluzione parziale
12:             Calcola i limiti per le nuove soluzioni parziali
13:             Aggiungi le nuove soluzioni parziali a Q
14:         end if
15:     end while
16:     return min_distance
17: end procedure
```

e il tasso di crescita esponenziale della complessità Bellman-Held-Karp limitano la loro applicazione pratica a istanze relativamente piccole del TSP.

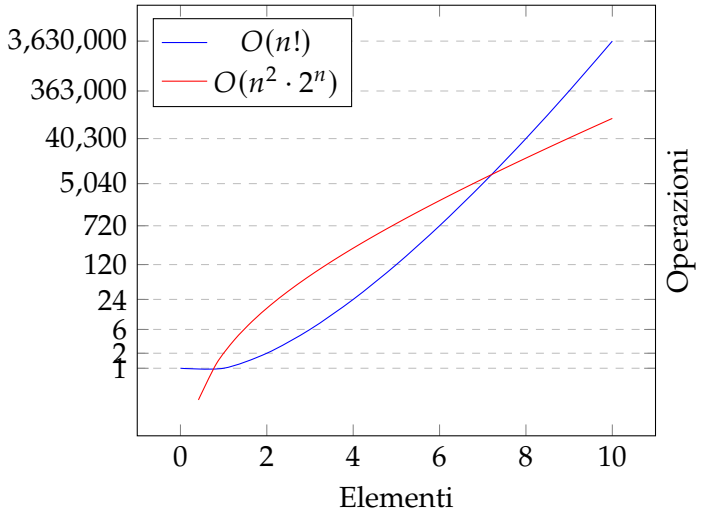


Figura 3.2: Complessità degli Algoritmi Esatti

3.2.2 Vantaggi e Svantaggi

Riassumendo:

Vantaggi:

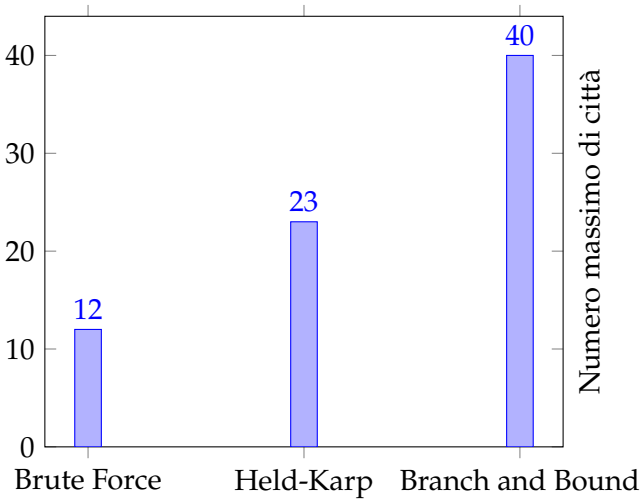


Figura 3.3: Dimensione massima del problema risolvibile in un'ora (approssimativo)

- Garanzia di trovare una soluzione ottimale.
- Forniscono un benchmark per valutare le prestazioni di algoritmi euristici e meteuristici.

Svantaggi:

- Complessità temporale esponenziale che li rende impraticabili per grandi istanze.
- Significative risorse computazionali richieste anche per problemi di dimensioni moderate.
- La programmazione dinamica e Held-Karp, pur essendo più efficienti del metodo a forza bruta, affrontano ancora limitazioni a causa della loro complessità spaziale e della necessità di un calcolo estensivo.

3.2.3 Concorde

Concorde è considerato uno dei risolutori esatti più efficienti per risolvere il TSP. Sviluppato da un gruppo di ricercatori guidati da Applegate et al., questo risolutore sfrutta una combinazione di metodi avanzati come il branch-and-bound, il cutting-plane e tecniche di programmazione lineare e combinatoria [7]. Concorde è in grado di risolvere istanze del TSP con di migliaia di città in tempi ragionevoli, rendendolo uno strumento di riferimento per confronti sperimentali e applicazioni pratiche di grande scala.

L'efficienza di Concorde è dovuta alla sua capacità di ridurre drasticamente lo spazio di ricerca mediante il branch-and-bound, migliorato con l'uso di piani di taglio per raffinare i limiti inferiori delle soluzioni. In aggiunta, Concorde implementa euristiche efficaci per identificare buone soluzioni iniziali, che aiutano a limitare l'esplorazione delle soluzioni subottimali. Concorde è anche stato utilizzato per risolvere problemi di benchmark nella

letteratura del TSP, come le istanze del problema TSPLIB [8], dimostrando la sua efficacia per problemi di dimensioni reali [9].

Vantaggi:

- Capacità di risolvere grandi istanze del TSP.
- Utilizzo combinato di tecniche avanzate di ottimizzazione.
- Implementazioni efficienti e ben documentate.

Svantaggi:

- Richiede elevate risorse computazionali per istanze di dimensioni estremamente grandi.
- La complessità delle tecniche implementate può renderlo difficile da adattare a problemi leggermente diversi dal TSP.

L'esplorazione degli algoritmi esatti getta le basi per comprendere le complessità computazionali intrinseche del TSP. Sebbene la loro applicazione pratica sia limitata dalle loro esigenze computazionali, rimangono cruciali per l'analisi teorica e per preparare il terreno per le tecniche risolutive alternative discusse nei capitoli successivi.

Capitolo 4

Algoritmi Euristici e Metaeuristici

4.1 Approcci Euristici

Gli approcci euristici per risolvere problemi come il TSP sono strategie progettate per trovare soluzioni sufficientemente buone entro un tempo ragionevole, specialmente quando una soluzione esatta non è fattibile a causa della complessità [10] del problema. Questi metodi sono essenziali nei casi in cui il costo computazionale per trovare la soluzione ottimale è proibitivo, come è comune con i problemi NP-difficili come il TSP [11].

4.1.1 Concetto e Necessità

Il concetto di approcci euristici si basa sull'idea di fare ipotesi educate o seguire algoritmi intuitivi che mirano a trovare soluzioni che siano vicine al miglior possibile, senza necessariamente garantire l'ottimalità [12]. Le euristiche sono caratterizzate dalle loro strategie empiriche che le rendono significativamente più veloci rispetto ai metodi esatti nella pratica, sebbene a costo di potenzialmente perdere la soluzione ottimale.

La necessità delle euristiche deriva dall'intrattabilità computazionale di molti problemi del mondo reale [13]. Le euristiche offrono un'alternativa pragmatica fornendo soluzioni che, pur non essendo garantite come ottimali, sono sufficientemente accurate per scopi pratici e possono essere ottenute molto più rapidamente [14].

4.1.2 Algoritmi Greedy

Gli algoritmi greedy costituiscono una classe significativa di approcci euristici caratterizzati dalla loro strategia di fare la scelta localmente ottimale in ogni fase con la speranza di trovare un ottimo globale [11]. Nel contesto del TSP e di problemi di ottimizzazione simili, gli algoritmi greedy semplificano i processi decisionali suddividendo un problema

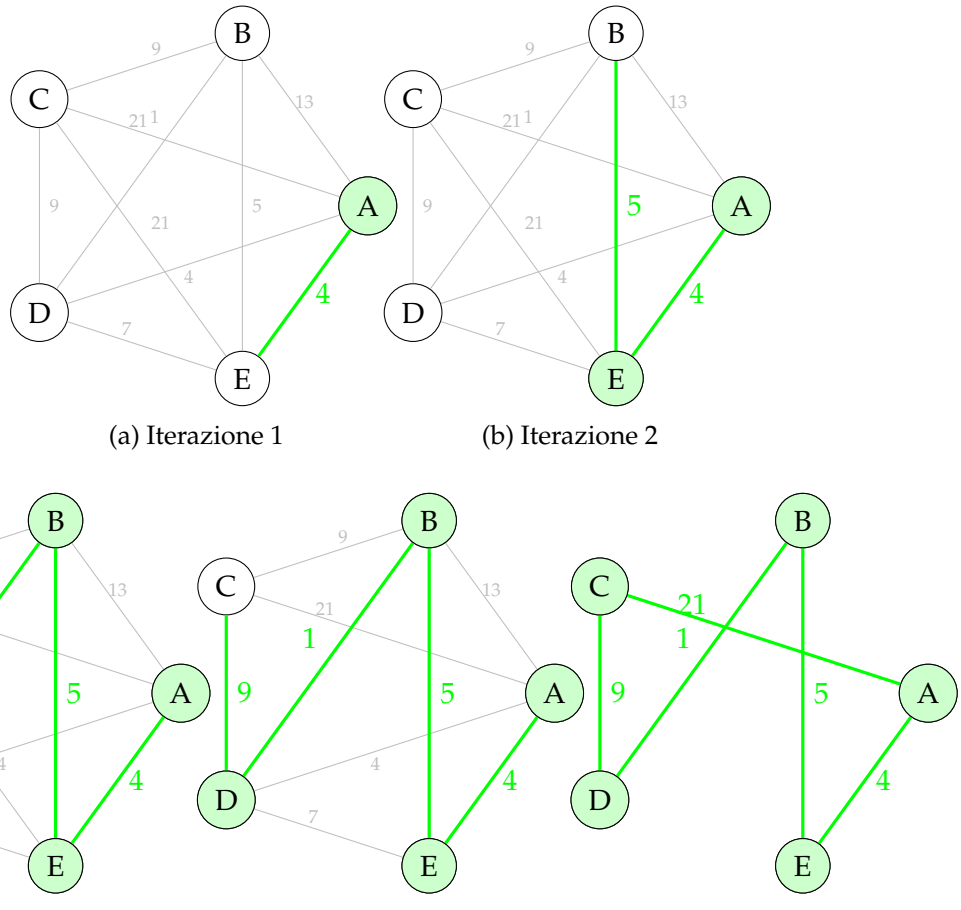
in una serie di passaggi e scegliendo la migliore opzione disponibile in ogni passaggio senza considerare le conseguenze più ampie di queste scelte[10].

4.1.2.1 Definizione e Principio

Un algoritmo greedy costruisce una soluzione pezzo per pezzo, scegliendo sempre il pezzo successivo che offre il beneficio più immediato[12]. Questo approccio è semplice e diretto, rendendolo un'opzione attraente per molti problemi in cui una soluzione rapida e facile è più preziosa di una ottimale. L'efficacia degli algoritmi greedy varia ampiamente da un problema all'altro; in alcuni casi, raggiungono una soluzione ottimale, mentre in altri, possono raggiungere solo una soluzione subottimale[13].

4.1.2.2 Applicazione al TSP: Nearest Neighbor Search

Nel TSP, un'applicazione classica dell'algoritmo greedy è il Nearest Neighbor Search Nearest Neighbor Search (NNS), dove l'algoritmo costruisce un tour partendo da una città arbitraria e, ad ogni passo, estende il tour spostandosi verso la città non visitata più vicina fino a quando tutte le città sono visitate[15].



(a) Iterazione 1 (b) Iterazione 2

(c) Iterazione 3 (d) Iterazione 4 (e) Iterazione 5

Algorithm 4 TSP NNS

```
1: procedure NearestNeighborSearch(cities, distances)
2:     start ← seleziona una città arbitraria da cities
3:     current ← start
4:     tour ← lista contenente start
5:     unvisited ← cities \ {start}
6:     while unvisited ≠ ∅ do
7:         next ← città in unvisited con distanza minima da current
8:         aggiungi next a tour
9:         rimuovi next da unvisited
10:        current ← next
11:    end while
12:    aggiungi start a tour per chiudere il ciclo
13:    return tour
14: end procedure
```

Pseudocodice Questa strategia greedy assicura che ad ogni passo, il tour cresca aggiungendo la destinazione più vicina possibile, ottimizzando per il passo immediato successivo senza considerare la lunghezza complessiva del tour. Sebbene questo metodo non garantisca la scoperta del tour più breve possibile, riduce significativamente il tempo di calcolo rispetto ai metodi di ricerca esaustiva.

4.1.2.3 Vantaggi e Limitazioni

Vantaggi:

- *Semplicità*: Gli algoritmi greedy sono semplici da implementare e comprendere.
- *Efficienza*: Spesso forniscono soluzioni rapidamente, rendendoli adatti per problemi in cui la velocità è cruciale.

Limitazioni:

- *Ottimalità*: Le scelte greedy non portano sempre alla soluzione ottimale, specialmente per problemi complessi come il TSP.
- *Corto raggio di azione*: Concentrandosi sull'ottimalità locale, potrebbero trascurare soluzioni migliori che richiedono scelte iniziali non ottimali.

Gli algoritmi greedy, con la loro semplicità ed efficienza intrinseche, giocano un ruolo cruciale nella risoluzione euristica dei problemi, specialmente nei casi in cui ottenere una soluzione esatta è computazionalmente impraticabile. La loro applicazione al TSP evidenzia l'equilibrio tra efficienza computazionale e qualità della soluzione, un tema che risuona attraverso lo studio degli algoritmi euristici e metaeuristici.

Tabella 4.1: Risultati dell'algoritmo Nearest Neighbour

Istanza	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
berlin52	14	8980.92	7542.00	19.08
eil51	15	513.61	426.00	20.57
eil76	17	711.99	538.00	32.34
lin105	43	20362.76	14379.00	41.61
pr124	136	69299.43	59030.00	17.40
d198	183	18620.07	15780.00	18.00
lin318	466	54033.58	42029.00	28.56
u574	1295	46881.87	36905.00	27.03
fl1577	9547	27940.91	22249.00	25.58
rl5915	236473	707498.63	565530.00	25.10

4.2 Algoritmi di Local Search

Gli algoritmi di ricerca locale rappresentano una classe di metodi euristici progettati per esplorare lo spazio delle soluzioni di un problema di ottimizzazione spostandosi iterativamente da una soluzione a una soluzione vicina nello spazio di ricerca. Questi algoritmi sono particolarmente efficaci per problemi come il TSP. La ricerca locale fornisce un meccanismo per migliorare una soluzione iniziale attraverso piccole modifiche localizzate [10, 13].

4.2.1 Tecniche 2-opt e 3-opt

Le tecniche 2-opt e 3-opt sono strategie di ricerca locale specifiche utilizzate per affinare le soluzioni del TSP. Funzionano rimuovendo iterativamente due o tre archi dal tour e riconnettendo i segmenti in un ordine diverso, mirando a ridurre la lunghezza totale del tour con ogni operazione.

4.2.1.1 Tecnica 2-opt

L'algoritmo 2-opt controlla sistematicamente ogni coppia di archi nel tour e determina se lo scambio di essi porterebbe a un percorso più breve. Questo processo viene ripetuto fino a quando non possono essere apportati ulteriori miglioramenti.

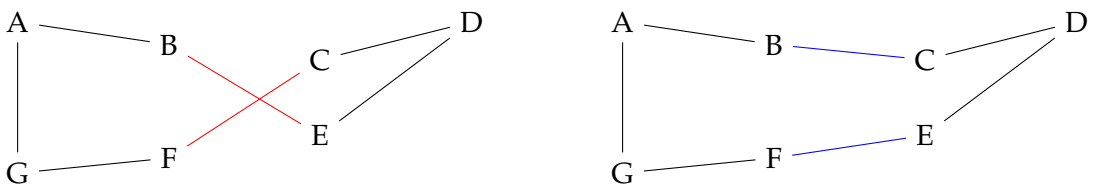


Figura 4.2: Scambio 2-opt

Algorithm 5 Tecnica 2-opt per TSP

```
1: procedure TwoOpt(tour, distances)
2:     improved ← true
3:     while improved do
4:         improved ← false
5:         for i ← 1 to length(tour) - 1 do
6:             for j ← i + 1 to length(tour) do
7:                 if TwoOptSwap(tour, i, j) < TourDistance(tour) then
8:                     tour ← TwoOptSwap(tour, i, j)
9:                     improved ← true
10:                end if
11:            end for
12:        end for
13:    end while
14:    return tour
15: end procedure
```

Tabella 4.2: Risultati dell'algoritmo Nearest Neighbour con ottimizzazione 2-opt

Istanza	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
berlin52	124	8060.65	7542.00	6.88
eil51	284	440.90	426.00	3.50
eil76	313	599.05	538.00	11.35
lin105	2959	16199.70	14379.00	12.66
pr124	3268	62757.01	59030.00	6.31
d198	4443	16165.31	15780.00	2.44
lin318	9562	46408.41	42029.00	10.42
u574	26923	39896.00	36905.00	8.10
fl1577	200151	24214.30	22249.00	8.83
rl5915	6327036	620822.08	565530.00	9.78

4.2.1.2 Tecnica 3-opt

La tecnica 3-opt estende l'idea del 2-opt considerando tre archi per la riconnessione. Questo consente una gamma più ampia di riarrangiamenti ad ogni passo, potenzialmente portando a miglioramenti più significativi a scapito di una maggiore complessità computazionale.

Pseudocodice per la Tecnica 3-opt è concettualmente simile al 2-opt ma coinvolge più condizioni per gli scambi, riflettendo la maggiore complessità di considerare tre archi alla volta.

4.2.2 Lin–Kernighan

L'algoritmo di Lin-Kernighan (Lin-Kernighan (LK)) è una delle tecniche più avanzate e potenti di ricerca locale per risolvere il TSP. Proposto da Shen Lin e Brian W. Kernighan

nel 1973, questo algoritmo estende le idee della tecnica k-opt, rendendola più flessibile e adattabile. L'algoritmo non si limita a una fissata dimensione k degli scambi, ma decide dinamicamente il numero di archi da rimuovere e ricollegare, rendendolo molto efficace nel migliorare soluzioni iniziali.[16]

Algorithm 6 Lin-Kernighan Algorithm per il TSP

```
1:  procedure LinKernighan(T)                          ▷ T è il tour iniziale
2:      T' ← T                                         ▷ Inizializza T' con T
3:      repeat
4:          for ogni città x_1 in T' do
5:              k ← 2
6:              repeat
7:                  Seleziona un insieme di k archi (x_1, y_1), (x_2, y_2), ..., (x_k, y_k) da T'
8:                  Genera un nuovo tour T'' rimuovendo gli archi e riconnettendo secondo
    la mossa k-opt
9:                  if T'' è più corto di T' then
10:                     T' ← T''
11:                     break          ▷ Esce dal loop interno se c'è un miglioramento
12:                 end if
13:                 k ← k + 1
14:             until Nessun ulteriore miglioramento
15:         end for
16:     until Tutte le città sono state provate come città iniziali
17:     return T'
18: end procedure
```

4.2.2.1 Descrizione

L'algoritmo di LK inizia con una soluzione iniziale e tenta di migliorare iterativamente questa soluzione tramite una serie di mosse k-opt generalizzate. Ogni mossa k-opt cerca di ridurre la lunghezza totale del tour scambiando k archi del tour attuale con nuovi archi, migliorando così il percorso complessivo.

L'algoritmo di LK è stato largamente studiato e migliorato nel corso degli anni. Per un'analisi dettagliata dell'algoritmo e delle sue estensioni, si possono consultare i seguenti riferimenti [10, 13, 15].

Gli algoritmi di ricerca locale, comprese le tecniche 2-opt e 3-opt offrono metodi potenti per migliorare le soluzioni del TSP. Affinando iterativamente una soluzione iniziale, bilanciano l'esplorazione dello spazio delle soluzioni e lo sfruttamento di configurazioni buone conosciute, fornendo mezzi efficaci per avvicinarsi a soluzioni ottimali per problemi di ottimizzazione complessi.

4.3 Algoritmi metaeuristici

Gli algoritmi metaeuristici sono strategie di alto livello progettate per navigare nello spazio di ricerca di problemi complessi di ottimizzazione in modo efficiente. Questi algoritmi non garantiscono una soluzione ottimale, ma sono efficaci nel trovare soluzioni molto buone entro un lasso di tempo ragionevole, specialmente per problemi in cui i metodi tradizionali falliscono a causa della maledizione della dimensionalità o della presenza di numerosi ottimi locali.

4.3.1 Simulated Annealing

Il Simulated Annealing (Simulated Annealing (SA)) si presenta come una tecnica stocastica progettata per approssimare l'ottimo globale di una data funzione obiettivo. Questo metodo trova le sue radici nel processo fisico di tempera nella metallurgia, che comporta il riscaldamento e il raffreddamento controllato di un materiale per aumentare la dimensione dei suoi cristalli e ridurre i loro difetti. L'algoritmo SA imita questo processo introducendo una "temperatura" metaforica che diminuisce gradualmente secondo un programma prestabilito. Inizialmente, l'algoritmo è più propenso ad accettare soluzioni che sono peggiori della soluzione attuale, permettendogli di esplorare ampiamente lo spazio delle soluzioni e di evitare di rimanere intrappolato in ottimi locali prematuramente. Man mano che la temperatura diminuisce, l'algoritmo diventa più selettivo, concentrandosi sulla soluzione ottimale.

4.3.1.1 Applicazione al TSP

Nel contesto del TSP, l'algoritmo SA inizia con una soluzione arbitraria, tipicamente un tour casuale tra le città. Successivamente, affina iterativamente questo tour apportando lievi modifiche, come lo scambio di due città nella sequenza del tour. La decisione di accettare una nuova soluzione è probabilistica e dipende sia dalla differenza nella qualità della soluzione sia dalla temperatura attuale. La capacità di questo metodo di accettare soluzioni peggiori diminuisce probabilisticamente nel tempo man mano che la temperatura si abbassa, simulando il processo di raffreddamento nella tempera.

Tabella 4.3: Risultati dell'algoritmo Simulated Annealing

Istanza	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
berlin52	28982	7544.37	7542.00	0.03
87eil76	39587	572.81	538.00	6.47
29lin105	52928	14993.92	14379.00	4.28
44d198	102362	16318.76	15780.00	3.41
49eil51	131542	434.25	426.00	1.94
pr124	207125	60675.98	59030.00	2.79
67u574	398181	43936.09	36905.00	19.05
lin318	506369	47651.44	42029.00	13.38
fl1577	1514830	27584.16	22249.00	23.98
rl5915	22563370	680777.61	565530.00	20.38

Pseudocodice Il pseudocodice per applicare SA per risolvere il TSP è delineato di seguito. Questa rappresentazione algoritmica enfatizza il miglioramento iterativo di un tour, tenendo conto dell'accettazione di soluzioni subottimali basata su un programma di temperatura decrescente per sfuggire ai minimi locali.

Algorithm 7 Simulated Annealing

```
1:  procedure SATSP(cities, initialTemp, coolingRate, stoppingTemp)
2:      bestSolution ← generateInitialSolution(cities)
3:      currentSolution ← bestSolution
4:      currentTemp ← initialTemp
5:      while currentTemp > stoppingTemp do
6:          newSolution ← perturbSolution(currentSolution, cities)
7:          currentCost ← calculateCost(currentSolution)
8:          newCost ← calculateCost(newSolution)
9:          acceptanceProb ← AcceptanceProbability(currentCost, newCost, currentTemp)
10:         if acceptanceProb > randomValue(0, 1) then
11:             currentSolution ← newSolution
12:             if newCost < calculateCost(bestSolution) then
13:                 bestSolution ← newSolution
14:             end if
15:         end if
16:         currentTemp ← updateTemperature(currentTemp, coolingRate)
17:     end while
18:     return bestSolution
19: end procedure
```

In questo pseudocodice, *generateInitialSolution* crea un tour iniziale casuale delle città. *perturbSolution* altera leggermente la soluzione attuale, tipicamente scambiando due città nel tour. *calculateCost* valuta la distanza totale (o costo) di un tour. *calculateAcceptanceProbability* determina la probabilità di accettare una nuova soluzione basata sul suo costo, sul costo della soluzione attuale e sulla temperatura attuale, seguendo il criterio di Metropolis[17]. La temperatura viene aggiornata in ogni iterazione dalla funzione

updateTemperature, che riduce la temperatura in base al tasso di raffreddamento fino a raggiungere la temperatura di arresto.

4.3.2 Algoritmi Genetici

Gli Algoritmi Genetici (GA) sono una classe di metodi euristici ispirati ai principi dell'evoluzione biologica e della selezione naturale. Introdotti da John Holland negli anni '60, questi algoritmi sono particolarmente efficaci nell'affrontare problemi di ottimizzazione complessi e non lineari, come il TSP.

4.3.2.1 Principi fondamentali

Gli algoritmi genetici operano su una popolazione di soluzioni potenziali, codificate come "cromosomi". I principi fondamentali includono:

- **Selezione**: Le soluzioni migliori (più "adatte") hanno maggiori probabilità di essere selezionate per la riproduzione.
- **Crossover**: Combinazione di parti di due soluzioni genitoriali per crearne una nuova.
- **Mutazione**: Modifiche casuali introdotte nelle soluzioni per mantenere la diversità genetica.
- **Evoluzione**: Le nuove generazioni tendono a migliorare nel tempo, convergendo verso soluzioni ottimali o quasi ottimali.

4.3.2.2 Applicazione al TSP

Nel contesto del TSP, un algorithmo genetico tipicamente opera come segue:

1. **Codifica**: Ogni tour è rappresentato come una sequenza di città (cromosoma).
2. **Popolazione iniziale**: Generazione di un insieme casuale di tour validi.
3. **Funzione di fitness**: Valutazione della lunghezza di ogni tour (fitness inversamente proporzionale alla lunghezza).
4. **Selezione**: Scelta dei tour migliori per la riproduzione (es. selezione a torneo o roulette).
5. **Crossover**: Combinazione di parti di due tour genitori (es. crossover parzialmente mappato - PMX).
6. **Mutazione**: Piccole modifiche casuali nei tour (es. scambio di due città).
7. **Sostituzione**: Creazione di una nuova generazione combinando genitori e figli.

8. **Iterazione**: Ripetizione del processo per un numero prefissato di generazioni o fino al raggiungimento di un criterio di arresto.

Tabella 4.4: Risultati dell'algoritmo Genetico

Istanza	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
eil51	22815	440.85	426.00	3.49
berlin52	26104	7918.09	7542.00	4.99
eil76	44547	570.63	538.00	6.06
d198	108498	16548.42	15780.00	4.87
lin105	109871	14573.34	14379.00	1.35
pr124	137120	61453.95	59030.00	4.11
lin318	320190	45165.59	42029.00	7.46
u574	1260895	40516.22	36905.00	9.79
fl1577	4033113	24024.81	22249.00	7.98
rl5915	33367821	648205.16	565530.00	14.62

4.3.2.3 Pseudocodice

Algorithm 8 Algoritmo Genetico per TSP

```
1:  procedure GeneticAlgorithmTSP(cities, populationSize, generations, mutationRate)
2:      population ← InitializePopulation(cities, populationSize)
3:      for g ← 1 to generations do
4:          EvaluateFitness(population)
5:          newPopulation ← ∅
6:          while |newPopulation| < populationSize do
7:              parent1 ← SelectParent(population)
8:              parent2 ← SelectParent(population)
9:              child ← Crossover(parent1, parent2)
10:             if Random() < mutationRate then
11:                 Mutate(child)
12:             end if
13:             newPopulation ← newPopulation ∪ {child}
14:         end while
15:         population ← newPopulation
16:     end for
17:     return BestSolution(population)
18: end procedure
```

4.3.2.4 Vantaggi

Gli algoritmi genetici offrono diversi vantaggi significativi per la risoluzione del TSP:

- **Esplorazione globale**: Capacità di esplorare ampie regioni dello spazio delle soluzioni, riducendo il rischio di convergenza prematura verso ottimi locali.

- **Parallelizzabilità**: La natura della popolazione si presta bene al calcolo parallelo, migliorando l'efficienza computazionale.
- **Flessibilità**: Facilmente adattabili a varianti del TSP e ad altri problemi di ottimizzazione combinatoria.
- **Robustezza**: Capacità di gestire funzioni obiettivo rumorose o mal definite.
- **Soluzioni multiple**: Possono fornire un insieme di soluzioni buone, non solo una singola soluzione ottimale.

4.3.2.5 Svantaggi

Nonostante i loro punti di forza, gli Algoritmi Genetici presentano anche alcune limitazioni:

- **Tempo di convergenza**: Possono richiedere molte generazioni per convergere, specialmente per problemi di grandi dimensioni.
- **Sensibilità ai parametri**: Le prestazioni dipendono fortemente dalla scelta di parametri come dimensione della popolazione, tasso di mutazione, metodo di selezione, ecc.
- **Non garantiscono l'ottimo globale**: Come altre metaeuristiche, non garantiscono di trovare la soluzione ottima globale.
- **Difficoltà di codifica**: Per alcuni problemi, la rappresentazione efficace delle soluzioni come "cromosomi" può essere complessa.
- **Perdita di diversità**: Possono soffrire di convergenza prematura se non gestiti correttamente, portando a soluzioni sub-ottimali.

4.3.2.6 Varianti e miglioramenti

Numerose varianti e miglioramenti sono stati proposti per ottimizzare le prestazioni degli Algoritmi Genetici per il TSP:

- **Operatori di crossover specializzati**: Come il Partially Mapped Crossover (PMX) (Partially Mapped Crossover) o l'Order Crossover (OX) (Order Crossover), progettati specificamente per preservare l'ordine relativo delle città.
- **Tecniche di diversificazione**: Introduzione di meccanismi per mantenere la diversità della popolazione, come il "crowding" o la "nicchia ecologica".
- **Ibridazione**: Combinazione con tecniche di ricerca locale (come 2-opt) per un'ottimizzazione fine delle soluzioni.

- **Adattamento dei parametri**: Strategie per regolare dinamicamente i parametri dell'algoritmo durante l'esecuzione.
- **Rappresentazioni alternative**: Utilizzo di codifiche diverse per i tour, come la rappresentazione basata su percorsi o su adiacenza.

Gli Algoritmi Genetici rappresentano un approccio potente e flessibile per affrontare il TSP e altri problemi di ottimizzazione combinatoria. La loro capacità di esplorare efficacemente spazi di soluzioni vasti e complessi li rende particolarmente adatti per istanze di grandi dimensioni o con caratteristiche non standard. Tuttavia, l'efficacia degli Algoritmi Genetici dipende fortemente dalla scelta appropriata di operatori genetici, parametri e strategie di implementazione. La ricerca continua in questo campo mira a migliorare ulteriormente le prestazioni e l'applicabilità di questi algoritmi a una gamma sempre più ampia di problemi di ottimizzazione.

4.3.3 Tabu Search

Tabu Search (Tabu Search (TS)) è una metaeuristica di ottimizzazione che estende la ricerca locale incorporando strutture di memoria per guidare il processo di ricerca. Introdotta da Fred Glover nel 1986, questa tecnica è particolarmente efficace per problemi di ottimizzazione combinatoria come il TSP.

4.3.3.1 Principi fondamentali

L'idea chiave di TS è l'uso di una memoria adattiva (la lista tabu) per evitare di tornare a soluzioni recentemente visitate, permettendo così all'algoritmo di esplorare nuove aree dello spazio delle soluzioni e potenzialmente sfuggire agli ottimi locali. La lista tabu memorizza caratteristiche delle soluzioni recenti o mosse effettuate, vietando temporaneamente il loro riutilizzo.

4.3.3.2 Applicazione al TSP

Nel contesto del TSP, Tabu Search opera tipicamente come segue:

1. Inizia con una soluzione iniziale (un tour completo).
2. Ad ogni iterazione, esplora il vicinato della soluzione corrente (ad esempio, scambiando coppie di città).
3. Seleziona la migliore mossa non tabu, anche se peggiora la soluzione corrente.
4. Aggiorna la lista tabu, aggiungendo la mossa appena effettuata e rimuovendo le mosse più vecchie se necessario.
5. Ripete il processo per un numero predefinito di iterazioni o fino a soddisfare un criterio di arresto.

4.3.3.3 Pseudocodice

Algorithm 9 Tabu Search per TSP

```
1: procedure TabuSearchTSP(initialSolution, maxIterations, tabuListSize)
2:     currentSolution ← initialSolution
3:     bestSolution ← currentSolution
4:     tabuList ← inizializza lista vuota
5:     for iteration ← 1 to maxIterations do
6:         neighborhood ← generaVicinato(currentSolution)
7:         bestCandidate ← null
8:         for candidate in neighborhood do
9:             if (candidate migliore di bestCandidate AND mossa non in tabuList) OR
10:        (candidate migliore di bestSolution) then
11:                bestCandidate ← candidate
12:            end if
13:        end for
14:        currentSolution ← bestCandidate
15:        if currentSolution migliore di bestSolution then
16:            bestSolution ← currentSolution
17:        end if
18:        Aggiorna tabuList
19:    end for
20:    return bestSolution
21: end procedure
```

4.3.3.4 Vantaggi

Il TS offre diversi vantaggi significativi:

- **Evita ottimi locali**: La lista tabu permette all'algoritmo di accettare temporaneamente mosse peggiorative, aiutando a sfuggire agli ottimi locali.
- **Esplorazione efficiente**: Bilancia efficacemente l'esplorazione di nuove aree dello spazio delle soluzioni e lo sfruttamento delle soluzioni buone trovate.
- **Adattabilità**: Può essere facilmente adattato a vari problemi di ottimizzazione combinatoria.
- **Memoria a breve e lungo termine**: Può incorporare strategie di memoria a breve e lungo termine per guidare la ricerca in modo più intelligente.

4.3.3.5 Svantaggi

Nonostante i suoi punti di forza, TS presenta anche alcune limitazioni:

- **Sensibilità ai parametri**: Le prestazioni possono dipendere fortemente dalla scelta dei parametri, come la dimensione della lista tabu e i criteri di aspirazione.

- **Complessità computazionale**: L'esplorazione del vicinato ad ogni iterazione può essere computazionalmente costosa per problemi di grandi dimensioni.
- **Non garantisce l'ottimo globale**: Come altre metaeuristiche, non garantisce di trovare la soluzione ottima globale.
- **Difficoltà di implementazione**: L'implementazione efficiente delle strutture di memoria e dei criteri di aspirazione può essere complessa.

TS rappresenta un potente strumento nell'arsenale delle metaeuristiche per il TSP e altri problemi di ottimizzazione combinatoria. La sua capacità di evitare ottimi locali e di esplorare efficacemente lo spazio delle soluzioni lo rende particolarmente adatto per istanze di problemi di grandi dimensioni o complessi. Tuttavia, come per molte tecniche avanzate, la sua efficacia dipende da un'attenta calibrazione e implementazione.

4.4 Ant Colony Optimization

L'Ant Colony Optimization (ACO) è un algoritmo metaeuristico ispirato al comportamento delle colonie di formiche reali [18]. Questo approccio simula il modo in cui le formiche trovano i percorsi più brevi tra le fonti di cibo e il loro nido, applicando questo concetto alla risoluzione di problemi di ottimizzazione combinatoria come il TSP [19].

4.4.1 Principi Fondamentali dell'ACO

L'ACO si basa su diversi principi chiave:

1. **Feromoni**: Le formiche depositano una sostanza chimica chiamata feromone lungo i percorsi che seguono. Questo serve come meccanismo di comunicazione indiretta tra le formiche [20].
2. **Evaporazione**: I feromoni evaporano nel tempo, permettendo all'algoritmo di "dimenticare" percorsi meno ottimali e prevenire la convergenza prematura [21].
3. **Probabilità di scelta**: Le formiche scelgono il loro percorso in base alla quantità di feromoni presenti e alla distanza tra le città, con un elemento di casualità [22].

4.4.2 Applicazione al TSP

Nel contesto del TSP, l'ACO opera come segue:

1. **Inizializzazione**: Si deposita una piccola quantità di feromoni su tutti i percorsi possibili.

2. **Costruzione del Tour**: Ogni formica costruisce un tour completo, scegliendo la prossima città da visitare in base alla formula:

$$P_{ij}^k = \frac{(\tau_{ij})^\alpha \cdot (\eta_{ij})^\beta}{\sum_{l \in N_i^k} (\tau_{il})^\alpha \cdot (\eta_{il})^\beta}$$

dove:

- P_{ij}^k è la probabilità che la formica k si sposti dalla città i alla città j
- τ_{ij} è la quantità di feromoni sul percorso (i, j)
- η_{ij} è l'inverso della distanza tra le città i e j
- α e β sono parametri che controllano l'importanza relativa dei feromoni e della distanza
- N_i^k è l'insieme delle città non ancora visitate dalla formica k

3. **Aggiornamento dei Feromoni**: Dopo che tutte le formiche hanno completato i loro tour, i feromoni vengono aggiornati secondo la formula:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k$$

dove:

- ρ è il tasso di evaporazione dei feromoni
- m è il numero di formiche
- $\Delta\tau_{ij}^k$ è la quantità di feromoni depositati dalla formica k sul percorso (i, j), solitamente proporzionale alla qualità del tour trovato

4. **Iterazione**: I passi 2 e 3 vengono ripetuti per un numero predefinito di iterazioni o fino a quando non si raggiunge un criterio di terminazione.

Il seguente pseudocodice descrive l'implementazione di base dell'ACO per il TSP:

Le procedure `CostruisciTour` e `AggiornaPheromoni` sono rappresentate in forma semplificata e possono essere ulteriormente dettagliate in un'implementazione reale.

4.4.3 Varianti e Miglioramenti

Diverse varianti dell'ACO sono state proposte per migliorare le prestazioni dell'algoritmo sul TSP:

- **MAX-MIN Ant System (Max-Min Ant System (MMAS))**: Introduce limiti superiori e inferiori alla quantità di feromoni, prevenendo la stagnazione [21].

```
1: procedure CostruisciTour(G, α, β)
2:     Scegli una città di partenza casuale
3:     while ci sono città non visitate do
4:         Scegli la prossima città usando la formula di probabilità
5:     end while
6:     return tour completo
7: end procedure
8: procedure AggiornaPheromoni(G, ρ, Q)
9:     for ogni arco (i, j) in G do
10:        τ_ij ← (1 − ρ)τ_ij
11:        for k = 1 to m do
12:            if (i, j) è nel tour_k then
13:                τ_ij ← τ_ij + Q/length_k
14:            end if
15:        end for
16:    end for
17: end procedure
```

Algorithm 10 ACO per il TSP

```
1: procedure ACO-TSP(G, m, α, β, ρ, Q, max_iterations)
2:     G ← grafo delle città
3:     m ← numero di formiche
4:     α, β ← parametri di controllo
5:     ρ ← tasso di evaporazione dei feromoni
6:     Q ← costante per il deposito di feromoni
7:     max_iterations ← numero massimo di iterazioni
8:     Inizializza i feromoni τ_ij = τ_0 per ogni arco (i, j) in G
9:     best_tour ← ∅
10:    best_length ← ∞
11:    for iteration = 1 to max_iterations do
12:        for k = 1 to m do
13:            tour_k ← CostruisciTour(G, α, β)
14:            length_k ← CalcolaLunghezzaTour(tour_k)
15:            if length_k < best_length then
16:                best_tour ← tour_k
17:                best_length ← length_k
18:            end if
19:        end for
20:        AggiornaPheromoni(G, ρ, Q)
21:    end for
22:    return best_tour, best_length
23: end procedure
```

- **Ant Colony System (ACS)**: Utilizza una regola di transizione più aggressiva e un aggiornamento locale dei feromoni [19].
- **Rank-Based Ant System**: Modifica la regola di aggiornamento dei feromoni in base alla qualità dei tour trovati [23].

4.4.4 Vantaggi e Sfide

L'ACO offre diversi vantaggi per la risoluzione del TSP:

- **Adattabilità**: Può facilmente adattarsi a cambiamenti nel problema, come l'aggiunta o la rimozione di città.
- **Qualità delle soluzioni**: Spesso trova soluzioni di alta qualità, specialmente per istanze di grandi dimensioni del TSP.

Tuttavia, l'ACO presenta anche alcune sfide:

- **Convergenza**: Può convergere prematuramente a soluzioni sub-ottimali se i parametri non sono correttamente calibrati.
- **Tempo di calcolo**: Per problemi di grandi dimensioni, può richiedere un tempo di calcolo significativo per raggiungere soluzioni di alta qualità.
- **Parametrizzazione**: L'efficacia dell'algoritmo dipende fortemente dalla scelta dei parametri, che può essere difficile da ottimizzare.

4.4.5 Conclusioni

L'ACO rappresenta un approccio potente e flessibile per affrontare il Traveling Salesman Problem. Ispirandosi ai processi naturali di foraggiamento delle formiche, l'ACO offre un meccanismo robusto per esplorare lo spazio delle soluzioni del TSP, bilanciando efficacemente l'esplorazione di nuove soluzioni con lo sfruttamento delle informazioni accumulate. Nonostante le sfide legate alla parametrizzazione e al tempo di calcolo, l'ACS continua a essere un'area attiva di ricerca e sviluppo nel campo dell'ottimizzazione combinatoria [24].

Capitolo 5

Ant Colony System di Dorigo et al

5.1 Fondamenti

5.1.1 Ispirazione Biologica

L'Ant Colony System (ACS) trae ispirazione delle formiche reali, che possono trovare in modo efficiente i percorsi più brevi tra le fonti di cibo e il loro nido. Questa capacità è attribuita ai sentieri di feromoni tracciati dalle formiche, facilitando una forma indiretta di comunicazione che guida altre formiche verso percorsi ottimali[3, 25].

La stigmergia, un meccanismo di coordinazione indiretta tra agenti, gioca un ruolo cruciale in questo processo[26]. Le formiche depositano feromoni mentre si muovono, creando un feedback positivo che rafforza i percorsi più efficienti nel tempo[27].

5.1.2 Principi Algoritmici

L'ACS sfrutta una colonia di formiche artificiali che lavorano cooperativamente per esplorare e sfruttare soluzioni per problemi di ottimizzazione. L'approccio integra sia l'esplorazione di nuove soluzioni sia lo sfruttamento di soluzioni buone conosciute attraverso un equilibrio tra l'influenza dei feromoni e la desiderabilità euristica.[3, 25]

Dorigo et al. hanno introdotto diversi miglioramenti rispetto agli algoritmi precedenti basati sulle formiche:

- Una regola di transizione di stato più forte che bilancia meglio l'esplorazione di nuovi archi e lo sfruttamento della conoscenza accumulata sui problemi.
- Una regola di aggiornamento globale dei feromoni applicata solo agli archi appartenenti al miglior tour.

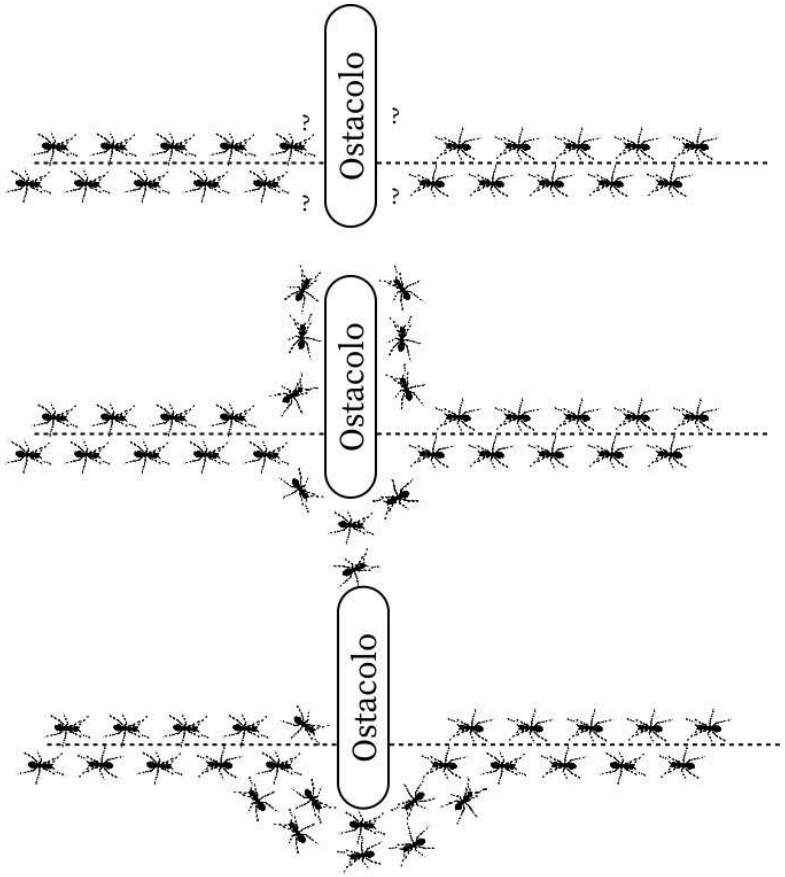


Figura 5.1: Comportamento delle formiche di fronte ad un ostacolo

- L'applicazione di una regola di aggiornamento locale dei feromoni durante la costruzione delle soluzioni.

Nella figura 5.1

1. Le formiche scelgono randomicamente tra i percorsi disponibili
2. Poiché le formiche si muovono a una velocità approssimativamente costante, quelle che scelgono il percorso inferiore, più corto, raggiungono il punto di decisione opposto più velocemente di quelle che scelgono il percorso superiore, più lungo.
3. Il feromone si accumula a un ritmo più elevato sul percorso più breve. Le formiche scelgono il percorso inferiore in base alla concentrazione di feromone.

5.1.3 Applicazione al TSP

Il TSP è particolarmente adatto per l'applicazione dell'ACS a causa della sua necessità di soluzioni efficienti per la ricerca di percorsi. L'algoritmo simula il comportamento delle formiche per costruire percorsi, con il processo di ottimizzazione guidato sia da informazioni euristiche (ad es., distanze tra le città) sia da sentieri di feromoni che rappresentano una forma di esperienza appresa[25, 28].

Nel contesto del TSP, ogni formica costruisce un tour visitando tutte le città esattamente una volta. La scelta della prossima città da visitare è influenzata sia dalla distanza (informazione euristica) che dalla quantità di feromoni sull'arco che collega le città (esperienza accumulata).

5.2 Dettagli implementativi

5.2.1 Regole di Aggiornamento dei Feromoni

Le regole di aggiornamento dei feromoni nell'ACS sono cruciali per guidare il processo di ricerca verso aree promettenti dello spazio delle soluzioni. Queste regole sono suddivise in aggiornamenti locali e globali.

5.2.1.1 Regola di Aggiornamento Locale

Ogni volta che una formica attraversa un arco (i, j), applica la regola di aggiornamento locale dei feromoni per diminuire il livello di feromoni, incoraggiando l'esplorazione da parte di altre formiche. La regola di aggiornamento locale è data da:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \rho \cdot \tau_0$$

dove τ_{ij} è la concentrazione di feromoni sull'arco (i, j), ρ è il tasso di evaporazione locale ($0 < \rho < 1$) e τ_0 è il livello iniziale di feromoni.

L'effetto di questa regola è di ridurre la quantità di feromoni su un arco visitato, rendendo quell'arco meno desiderabile per le formiche successive. Ciò aumenta l'esplorazione di percorsi alternativi e aiuta a prevenire la stagnazione precoce su soluzioni subottimali.

5.2.1.2 Regola di Aggiornamento Globale

Dopo che tutte le formiche hanno completato i loro tour, viene applicata la regola di aggiornamento globale agli archi che appartengono al miglior tour (sia globalmente che nell'iterazione corrente). L'aggiornamento globale migliora il sentiero di feromoni sui percorsi critici, rendendoli più attraenti per le formiche future. La regola di aggiornamento globale è definita come:

$$\tau_{ij} = (1 - \alpha) \cdot \tau_{ij} + \alpha \cdot \Delta\tau_{ij}^{\text{best}}$$

dove α è il tasso di evaporazione globale ($0 < \alpha < 1$) e $\Delta\tau_{ij}^{\text{best}}$ rappresenta i feromoni depositati dalla migliore formica, definito come:

$$\Delta\tau_{ij}^{\text{best}} = \begin{cases} (L_{\text{best}})^{-1} & \text{se } (i, j) \text{ appartiene al miglior tour} \\ 0 & \text{altrimenti} \end{cases}$$

dove L_{best} è la lunghezza del miglior tour.

Questa regola intensifica i feromoni sui percorsi che hanno portato alle migliori soluzioni, guidando la ricerca verso regioni promettenti dello spazio delle soluzioni.

Tabella 5.1: Risultati dell'algoritmo ACS

Istanza	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
eil51	17083	434.91	426.00	2.09
berlin52	18411	7606.66	7542.00	0.86
eil76	33136	554.04	538.00	2.98
d198	94445	16584.44	15780.00	5.10
lin105	123877	14981.78	14379.00	4.19
pr124	292446	60760.86	59030.00	2.93
lin318	338493	46646.52	42029.00	10.99
u574	452117	42512.54	36905.00	15.19
fl1577	2880286	25939.85	22249.00	16.59
rl5915	35872198	706188.19	565530.00	24.87

5.2.2 Regole di Movimento delle Formiche

Le formiche selezionano la prossima città da visitare basandosi su una regola decisionale probabilistica che bilancia l'esplorazione di nuovi percorsi con lo sfruttamento dei sentieri di feromoni e delle informazioni euristiche.

5.2.2.1 Regola di Transizione

In ACS la scelta della prossima città da visitare è governata dalla regola di transizione pseudo-casuale-proporzionale. Data una formica k nella città i, la probabilità di muoversi verso la città j è determinata da:

$$j = \begin{cases} 2\max_{l \in N_i^k}\{[\tau_{il}] \cdot [\eta_{il}]^\beta\} & \text{se } q \leq q_0 \text{ (sfruttamento)} \\ J & \text{altrimenti (esplorazione bilanciata)} \end{cases}$$

dove q è un numero casuale uniformemente distribuito in $[0, 1]$, q_0 è un parametro ($0 \leq q_0 \leq 1$) che determina il bilanciamento tra sfruttamento ed esplorazione, τ_{il} è la quantità di feromoni sull'arco (i, l), η_{il} è il valore euristico (solitamente l'inverso della distanza), β è un parametro che controlla l'importanza relativa dell'informazione euristica, e N_i^k è l'insieme delle città non ancora visitate dalla formica k posizionata nella città i.

J è una città casuale scelta secondo la distribuzione di probabilità:

$$p_{ij}^k = \frac{[\tau_{ij}] \cdot [\eta_{ij}]^\beta}{\sum_{l \in N_i^k} [\tau_{il}] \cdot [\eta_{il}]^\beta}$$

Questa regola di transizione favorisce lo sfruttamento delle informazioni accumulate (quando $q \leq q_0$) ma lascia spazio all'esplorazione (quando $q > q_0$).

5.2.3 Pseudocodice

Algorithm 11 ACS per il TSP

Inizializza i livelli di feromoni $\tau_{ij} = \tau_0$ per tutti gli archi (i, j)
Inizializza la migliore soluzione globale S_{gb} e la sua lunghezza L_{gb}
for ogni iterazione **do**
    for ogni formica $k = 1, \ldots, m$ **do**
        Posiziona la formica k su una città di partenza casuale
        Inizializza il tour parziale $S_k = \emptyset$
        repeat
            Seleziona la prossima città j usando la regola di transizione
            Aggiungi (i, j) a S_k
            Applica l'aggiornamento locale dei feromoni a (i, j)
            $i \leftarrow j$
        until il tour S_k è completo
        if $L(S_k) < L_{gb}$ **then**
            $S_{gb} \leftarrow S_k$
            $L_{gb} \leftarrow L(S_k)$
        end if
    end for
    Applica l'aggiornamento globale dei feromoni al miglior tour S_{gb}
end for
return la migliore soluzione trovata S_{gb}

5.2.4 Analisi della Complessità

La complessità temporale dell'ACS per iterazione è $O(n^2 m)$, dove n è il numero di città e m è il numero di formiche. Tuttavia, il numero di iterazioni necessarie per la convergenza può variare significativamente a seconda delle caratteristiche del problema e dei parametri dell'algoritmo.

5.2.5 Parametri Chiave e loro Impatto

L'efficacia dell'ACS dipende in modo cruciale dalla corretta impostazione di diversi parametri:

- ρ: Tasso di evaporazione locale dei feromoni
- α: Tasso di evaporazione globale dei feromoni
- β: Importanza relativa dell'informazione euristica
- q_0: Bilanciamento tra sfruttamento ed esplorazione
- m: Numero di formiche
- τ_0: Livello iniziale di feromoni

La scelta ottimale di questi parametri può variare a seconda del problema specifico e richiede spesso una fase di tuning.

5.3 Varianti e Estensioni

Dalla pubblicazione originale dell' ACS sono state proposte numerose varianti e estensioni:

- MAX-MIN Ant System (MMAS): Introdotto da Stützle e Hoos, MMAS limita i valori dei feromoni entro un intervallo predefinito per prevenire la stagnazione prematura. [29]
- Rank-Base Ant System (Ant System with Rank-based Pheromone Update (ASrank)): In questa variante, le formiche sono classificate in base alla qualità delle loro soluzioni, e la quanità dei fermoni depositati è proporzionale al rango.
- ACS Ibridi: Combinazioni dell'ACS con altre tecniche di ottimizzazione, come algoritmi genetici o Tabu Search, hanno mostrato risultati promettenti su vari problemi di ottimizzazione combinatoria.

5.4 Applicazioni oltre il TSP

Sebbene inizialmente sviluppato per il TSP, l'ACS e le sue varianti hanno trovato applicazioni in una vasta gamma di domini:

- Problemi di routing di veicoli
- Scheduling e pianificazione
- Assegnazione di frequenze nelle reti di telecomunicazione
- Bioinformatica e folding delle proteine
- Ottimizzazione di reti e grid computing
- Problemi di ottimizzazione multi-obiettivo

La versatilità dell'ACS lo rende un approccio potente per affrontare problemi complessi di ottimizzazione in vari campi dell'ingegneria e della scienza.

Capitolo 6

Red-Black Ant Colony System

6.1 Introduzione

Questo capitolo presenta una versione modificata dell'algoritmo ACS, chiamata Red-Black Ant Colony System (RB-ACS), che mira a migliorare le prestazioni dell'ACS su istanze di grandi dimensioni del TSP. Il RB-ACS incorpora diverse migliorie chiave all'approccio ACS standard, tra cui la ricerca in parallelo, l'inizializzazione migliorata dei feromoni e l'impostazione di parametri differenziati per i due gruppi di formiche. Queste modifiche sono progettate per consentire al RB-ACS di esplorare lo spazio di ricerca in modo più efficace e di convergere efficientemente verso soluzioni ottimali o quasi-ottimali, anche per problemi di TSP su larga scala [30].

6.2 L'Algoritmo Red-Black Ant Colony System (RB-ACS)

Mentre l'algoritmo ACS standard si è dimostrato efficace nella risoluzione di istanze del TSP, le sue prestazioni possono peggiorare all'aumentare delle dimensioni del problema. Il Red-Black Ant Colony System (RB-ACS) introduce diverse modifiche chiave all'approccio ACS standard per migliorarne la scalabilità e la qualità delle soluzioni per problemi di TSP su larga scala [30].

6.2.1 Inizializzazione dei Feromoni

Nell' ACS standard, il livello iniziale di feromone su tutti i archi è impostato a un valore costante τ_0. Nel RB-ACS, gli autori propongono uno schema di inizializzazione dei feromoni diverso, in cui il livello iniziale di feromone su ciascun bordo (r, s) è impostato

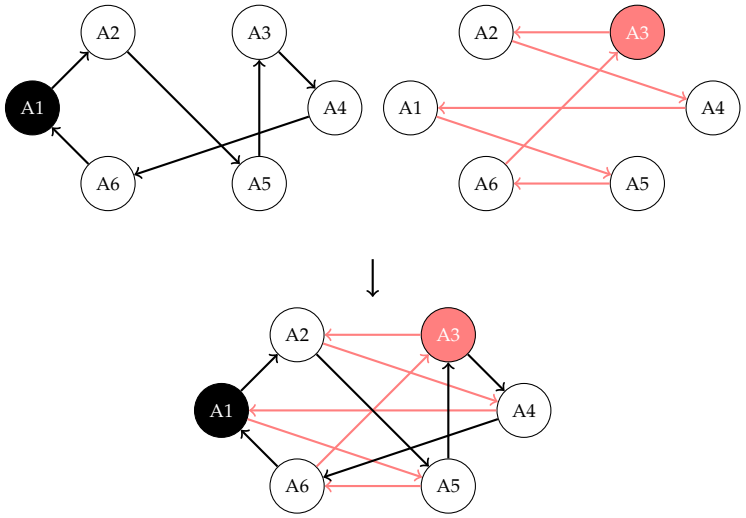


Figura 6.1: Un esempio di tour costruito da formiche rosse e nere nell'algoritmo RB-ACS.

secondo la seguente equazione:

$$\tau_{\text{init}}(r, s) = \frac{C}{\text{costo}(r, s)} \tag{6.1}$$

dove C è una costante e costo(r, s) è la distanza tra le città r e s. Questa inizializzazione modificata dei feromoni incoraggia le formiche a esplorare i archi più corti, che sono più desiderabili per il tour ottimale, assegnando loro livelli di feromone iniziali più alti [30].

6.2.2 Ricerca in Parallelo con Gruppi di Formiche Rosse e Nere

Un'altra modifica chiave nell'algoritmo RB-ACS è l'utilizzo di due gruppi separati di formiche, chiamati "rosse" e "nere", che esplorano lo spazio di ricerca in parallelo. Nell'ACS standard, un singolo gruppo di formiche costruisce i tour e le formiche possono seguire i percorsi di altre formiche, il che può portare a una convergenza prematura e a rimanere bloccati in minimi locali.

Nel RB-ACS, i gruppi di formiche rosse e nere operano in modo indipendente, mantenendo ciascuno i propri sentieri di feromoni e cercando soluzioni senza essere influenzati dall'altro gruppo. Questo approccio di ricerca parallela riduce la probabilità che l'algoritmo rimanga bloccato in minimi locali, poiché i due gruppi di formiche possono esplorare regioni diverse dello spazio di ricerca simultaneamente [30].

6.2.3 Impostazioni di Parametri Differenziate

Oltre al processo di ricerca separato, l'algoritmo RB-ACS impiega anche diverse impostazioni dei parametri per i gruppi di formiche rosse e nere. In particolare, la regola di aggiornamento locale dei feromoni e il tasso di evaporazione dei feromoni possono essere impostati su valori diversi per i due gruppi. Questa differenziazione è ispirata al comportamento osservato delle formiche reali, in cui colonie o gruppi diversi possono presentare caratteristiche distinte, come i tassi di deposizione e di evaporazione dei feromoni [30].

Utilizzando impostazioni di parametri separate per le formiche rosse e nere, l'algoritmo RB-ACS può ulteriormente migliorare la diversificazione del processo di ricerca, permettendo ai due gruppi di esplorare lo spazio di soluzione in modo più complementare.

6.2.4 Aggiornamento Globale dei Feromoni Migliorato

Nell' ACS standard, solo la formica globalmente migliore è autorizzata a depositare feromoni durante la fase di aggiornamento globale. Nel RB-ACS, gli autori propongono una regola di aggiornamento globale modificata, in cui le due migliori formiche di ciascuno dei gruppi rosso e nero sono autorizzate ad aggiornare i livelli di feromone. Questo aggiornamento globale parallelo rafforza ulteriormente la ricerca verso soluzioni di alta qualità, poiché vengono rinforzati simultaneamente diversi percorsi promettenti [30].

6.2.5 Pseudocodice

Algorithm 12 Red-Black Ant Colony System (RB-ACS) per il TSP

Inizializza i livelli di feromoni $\tau_{ij} = \tau_{\text{init}}(i, j)$ per tutti gli archi (i, j)
Inizializza la migliore soluzione globale S_{gb} e la sua lunghezza L_{gb}
for ogni iterazione **do**
    for ogni gruppo di formiche $g \in \{\text{rosso}, \text{nero}\}$ **do**
        for ogni formica $k = 1, \ldots, m_g$ **do**
            Posiziona la formica k del gruppo g su una città di partenza casuale
            Inizializza il tour parziale $S_k^g = \emptyset$
            repeat
                Seleziona la prossima città j usando la regola di transizione specifica per il gruppo g
                Aggiungi (i, j) a S_k^g
                Applica l'aggiornamento locale dei feromoni a (i, j) secondo le regole del gruppo g
                $i \leftarrow j$
            until il tour S_k^g è completo
            if $L(S_k^g) < L_{gb}$ **then**
                $S_{gb} \leftarrow S_k^g$
                $L_{gb} \leftarrow L(S_k^g)$
            end if
        end for
    end for
    for ogni gruppo $g \in \{\text{rosso}, \text{nero}\}$ **do**
        Seleziona le due migliori formiche del gruppo g
        Applica l'aggiornamento globale dei feromoni ai tour di queste formiche
    end for
end for
return la migliore soluzione trovata S_{gb}

Tabella 6.1: Risultati dell'algoritmo RB-ACS

Istanza	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
berlin52	34002	7681.45	7542.00	1.85
eil51	35268	445.59	426.00	4.60
eil76	40567	562.41	538.00	4.54
lin105	77720	14785.44	14379.00	2.83
d198	136998	17097.74	15780.00	8.35
pr124	468942	61167.76	59030.00	3.62
lin318	469358	46823.70	42029.00	11.41
u574	826684	43702.95	36905.00	18.42
fl1577	4210934	26788.92	22249.00	20.41
rl5915	21861319	716108.43	565530.00	26.63

6.3 Conclusione

L'algoritmo RB-ACS presentato in questo capitolo può essere considerato un approccio promettente per risolvere problemi di ottimizzazione combinatoria complessi oltre il TSP, come il bilanciamento del carico nelle reti di telecomunicazioni, il dispacciamento economico del carico, la schedulazione dei processi e vari altri campi. I principi generali del RB-ACS, tra cui la ricerca parallela, le impostazioni dei parametri differenziate e le strategie di aggiornamento migliorato dei feromoni, possono essere potenzialmente applicati a una vasta gamma di problemi di ottimizzazione, rendendolo un contributo prezioso nel campo dell'intelligenza di sciame e degli algoritmi metaeuristici.

Capitolo 7

Risultati Sperimentali

In questo capitolo, viene presentata un'analisi approfondita dei risultati sperimentali ottenuti dall'implementazione e valutazione di diversi algoritmi risolutivi per il *Travelling Salesman Problem* (TSP). Gli esperimenti sono stati condotti su una vasta gamma di istanze del problema, variando in dimensione e complessità, al fine di valutare le prestazioni degli algoritmi in termini di qualità della soluzione, tempo di esecuzione e scalabilità.

7.1 Metodologia Sperimentale

7.1.1 Setup

Gli esperimenti sono stati eseguiti su un cluster di calcolo (DEI Blade) con nodi equipaggiati con processori Intel Xeon Gold 5118 CPU @ 2.30/3.20GHz e 128 GB di RAM. Tutti gli algoritmi sono stati implementati in Rust.

7.1.2 Istanze del Problema

È stato utilizzato un insieme diversificato di istanze del TSP, tratte principalmente dalla TSPLIB [8]. Le istanze variano da problemi di piccole dimensioni (51 città) a problemi di grande scala (fino a 5915 città). Alcune delle istanze utilizzate includono:

- eil51, berlin52, st70, eil76, pr76 (problemi di piccole dimensioni)
- lin105, pr124, lin318, rat575 (problemi di medie dimensioni)
- rat783, pr1002, pcb1173, d1655, fl1577, u2319 (problemi di grandi dimensioni)
- rl5915 (problema di grandissima scala)

7.1.3 Algoritmi Implementati

Sono stati implementati e valutati i seguenti algoritmi:

- Nearest Neighbor (NN) e NN con 2-Opt (NN2Opt)
- Simulated Annealing (SA) e SA con 2-Opt (SA2Opt)
- Genetic Algorithm (GA) e GA con 2-Opt (GA2Opt)
- Ant Colony System (ACS) e ACS con 2-Opt (ACS2Opt)
- Red-Black Ant Colony System (RBACS) e RBACS con 2-Opt (RBACS2Opt)

7.1.4 Metriche di Valutazione

Le principali metriche utilizzate per valutare le prestazioni degli algoritmi sono:

- **Tempo di esecuzione**: misurato in millisecondi.
- **Qualità della soluzione**: lunghezza del tour trovato.
- **Gap percentuale**: calcolato come $(lunghezza - ottimo)/ottimo * 100\%$, dove "ottimo" è la lunghezza del tour ottimo noto per l'istanza.

7.2 Analisi dei Risultati

7.2.1 Prestazioni Generali

Dall'analisi dei dati sperimentali, emergono alcune tendenze generali:

- Gli algoritmi di base (NN, SA, GA, ACS, RB-ACS) mostrano prestazioni significativamente migliorate quando combinati con la procedura di ottimizzazione locale 2-Opt.
- ACS e RBACS, insieme alle loro varianti 2-Opt, tendono a produrre soluzioni di qualità superiore rispetto agli altri approcci, specialmente per istanze di grandi dimensioni.
- GA mostra prestazioni altamente variabili, con alcuni risultati eccellenti ma anche alcuni estremamente scarsi, soprattutto su istanze di grandi dimensioni.

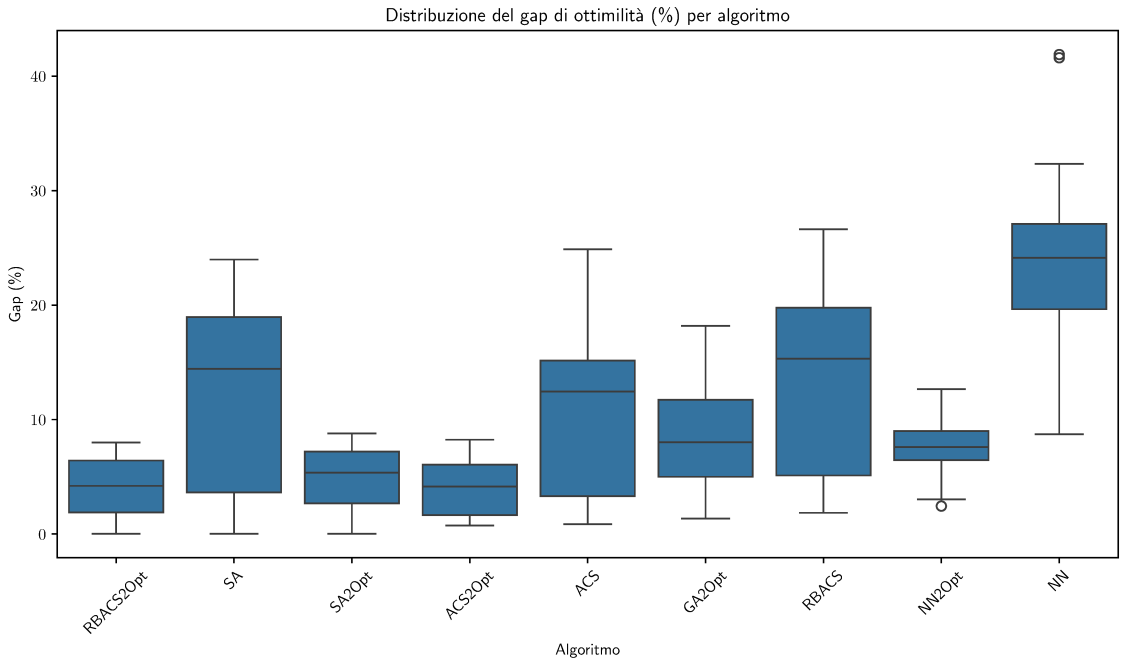


Figura 7.1: Distribuzione del gap di ottimalità per algoritmo

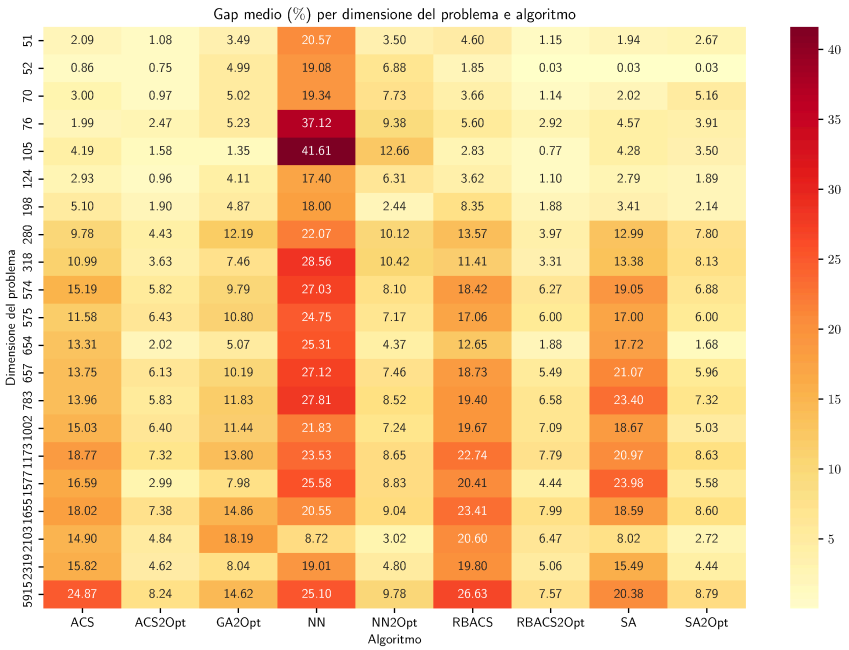


Figura 7.2: Heatmap del gap di ottimilità

Tabella 7.1: Risultati NN vs NN+2Opt

	Istanza	algorithm	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
0	berlin52	NN	14	8980.92	7542.00	19.08
1	berlin52	NN2Opt	124	8060.65	7542.00	6.88
2	d198	NN	183	18620.07	15780.00	18.00
3	d198	NN2Opt	4443	16165.31	15780.00	2.44
4	eil76	NN	17	711.99	538.00	32.34
5	eil76	NN2Opt	313	599.05	538.00	11.35
6	fl1577	NN	9547	27940.91	22249.00	25.58
7	fl1577	NN2Opt	200151	24214.30	22249.00	8.83
8	lin105	NN	43	20362.76	14379.00	41.61
9	lin105	NN2Opt	2959	16199.70	14379.00	12.66
10	lin318	NN	466	54033.58	42029.00	28.56
11	lin318	NN2Opt	9562	46408.41	42029.00	10.42
12	rl5915	NN	236473	707498.63	565530.00	25.10
13	rl5915	NN2Opt	6327036	620822.08	565530.00	9.78
14	u574	NN	1295	46881.87	36905.00	27.03
15	u574	NN2Opt	26923	39896.00	36905.00	8.10

7.2.2 Analisi Dettagliata per Algoritmo

7.2.2.1 Nearest Neighbor (NN e NN+2Opt)

NN è l'algoritmo più veloce, con tempi di esecuzione che vanno da pochi millisecondi per istanze piccole a pochi secondi per istanze molto grandi. Tuttavia, la qualità delle soluzioni è generalmente inferiore rispetto agli altri approcci:

L'aggiunta di 2-Opt migliora significativamente la qualità delle soluzioni, ma aumenta il tempo di calcolo di un ordine di grandezza o più.

7.2.2.2 Simulated Annealing (SA e SA2Opt)

SA offre un buon compromesso tra qualità della soluzione e tempo di calcolo:

Tabella 7.2: Risultati SA vs SA2Opt

	Istanza	algorithm	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
0	berlin52	SA	28982	7544.37	7542.00	0.03
1	berlin52	SA2Opt	49481	7544.37	7542.00	0.03
2	d198	SA2Opt	93813	16118.48	15780.00	2.14
3	d198	SA	102362	16318.76	15780.00	3.41
4	eil76	SA	39587	572.81	538.00	6.47
5	eil76	SA2Opt	45721	569.29	538.00	5.82
6	fl1577	SA	1514830	27584.16	22249.00	23.98
7	fl1577	SA2Opt	2391057	23489.49	22249.00	5.58
8	lin105	SA	52928	14993.92	14379.00	4.28
9	lin105	SA2Opt	54366	14882.69	14379.00	3.50
10	lin318	SA2Opt	392433	45444.25	42029.00	8.13
11	lin318	SA	506369	47651.44	42029.00	13.38
12	rl5915	SA2Opt	18690779	615257.00	565530.00	8.79
13	rl5915	SA	22563370	680777.61	565530.00	20.38
14	u574	SA2Opt	369840	39443.68	36905.00	6.88
15	u574	SA	398181	43936.09	36905.00	19.05

SA2Opt generalmente produce soluzioni migliori di SA, ma con tempi di calcolo significativamente più lunghi.

7.2.2.3 Genetic Algorithm (GA e GA+2Opt)

GA mostra le prestazioni più variabili tra tutti gli algoritmi testati:

Tabella 7.3: Risultati GA vs GA+2Opt

	Istanza	algorithm	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
0	berlin52	GA2Opt	26104	7918.09	7542.00	4.99
1	berlin52	GA	29932	11009.32	7542.00	45.97
2	d198	GA2Opt	108498	16548.42	15780.00	4.87
3	d198	GA	137106	66868.76	15780.00	323.76
4	eil76	GA2Opt	44547	570.63	538.00	6.06
5	eil76	GA	45556	977.18	538.00	81.63
6	fl1577	GA	2264528	1116417.45	22249.00	4917.83
7	fl1577	GA2Opt	4033113	24024.81	22249.00	7.98
8	lin105	GA2Opt	109871	14573.34	14379.00	1.35
9	lin105	GA	164073	44653.50	14379.00	210.55
10	lin318	GA2Opt	320190	45165.59	42029.00	7.46
11	lin318	GA	429314	363904.18	42029.00	765.84
12	rl5915	GA	20712540	38904633.04	565530.00	6779.32
13	rl5915	GA2Opt	33367821	648205.16	565530.00	14.62
14	u574	GA	471427	457563.15	36905.00	1139.84
15	u574	GA2Opt	1260895	40516.22	36905.00	9.79

7.2.2.4 Ant Colony System (ACS e ACS+2Opt)

ACS e ACS+2Opt mostrano prestazioni molto buone su tutta la gamma di istanze:

Tabella 7.4: Risultati ACS vs ACS+2Opt

	Istanza	algorithm	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
0	berlin52	ACS	18411	7606.66	7542.00	0.86
1	berlin52	ACS2Opt	20289	7598.44	7542.00	0.75
2	d198	ACS	94445	16584.44	15780.00	5.10
3	d198	ACS2Opt	101484	16079.65	15780.00	1.90
4	eil76	ACS2Opt	27703	558.76	538.00	3.86
5	eil76	ACS	33136	554.04	538.00	2.98
6	fl1577	ACS	2880286	25939.85	22249.00	16.59
7	fl1577	ACS2Opt	3852624	22914.07	22249.00	2.99
8	lin105	ACS2Opt	80683	14605.88	14379.00	1.58
9	lin105	ACS	123877	14981.78	14379.00	4.19
10	lin318	ACS2Opt	202670	43556.33	42029.00	3.63
11	lin318	ACS	338493	46646.52	42029.00	10.99
12	rl5915	ACS2Opt	34991642	612104.64	565530.00	8.24
13	rl5915	ACS	35872198	706188.19	565530.00	24.87
14	u574	ACS	452117	42512.54	36905.00	15.19
15	u574	ACS2Opt	490206	39051.51	36905.00	5.82

ACS2Opt tende a produrre soluzioni di qualità leggermente superiore rispetto ad ACS, ma con tempi di calcolo più lunghi.

7.2.2.5 Red-Black Ant Colony System (RBACS e RBACS2Opt)

RBACS e RBACS2Opt mostrano prestazioni competitive e talvolta superiori rispetto ad ACS e ACS2Opt:

Tabella 7.5: Risultati RB-ACS vs RB-ACS+2Opt

	Istanza	algorithm	Tempo (ms)	Lunghezza Tour	Lunghezza ottima	Gap
0	berlin52	RBACS	34002	7681.45	7542.00	1.85
1	berlin52	RBACS2Opt	34159	7544.37	7542.00	0.03
2	d198	RBACS	136998	17097.74	15780.00	8.35
3	d198	RBACS2Opt	283417	16076.46	15780.00	1.88
4	eil76	RBACS	40567	562.41	538.00	4.54
5	eil76	RBACS2Opt	40672	557.14	538.00	3.56
6	fl1577	RBACS	4210934	26788.92	22249.00	20.41
7	fl1577	RBACS2Opt	6887456	23236.46	22249.00	4.44
8	lin105	RBACS	77720	14785.44	14379.00	2.83
9	lin105	RBACS2Opt	262217	14489.08	14379.00	0.77
10	lin318	RBACS	469358	46823.70	42029.00	11.41
11	lin318	RBACS2Opt	603140	43421.71	42029.00	3.31
12	rl5915	RBACS	21861319	716108.43	565530.00	26.63
13	rl5915	RBACS2Opt	28403435	608329.94	565530.00	7.57
14	u574	RBACS	826684	43702.95	36905.00	18.42
15	u574	RBACS2Opt	899785	39219.59	36905.00	6.27

RB-ACS2+Opt sembra particolarmente efficace su istanze di grandi dimensioni, offrendo un buon compromesso tra qualità della soluzione e tempo di calcolo.

7.2.3 Analisi della Scalabilità

Per valutare la scalabilità degli algoritmi, è stato analizzato come il tempo di esecuzione e la qualità della soluzione variano al crescere della dimensione del problema:

- NN e NN2Opt mostrano la migliore scalabilità in termini di tempo di esecuzione, ma la qualità della soluzione degrada rapidamente per istanze più grandi.
- SA e SA2Opt mantengono una buona qualità della soluzione anche per istanze grandi, ma il tempo di calcolo cresce rapidamente.
- GA e GA2Opt mostrano una scalabilità scarsa sia in termini di tempo che di qualità della soluzione per istanze molto grandi.
- ACS, ACS2Opt, RBACS e RBACS2Opt offrono il miglior compromesso tra scalabilità del tempo di esecuzione e mantenimento della qualità della soluzione per istanze di grandi dimensioni.

7.3 Discussione

I risultati sperimentali evidenziano diversi punti chiave:

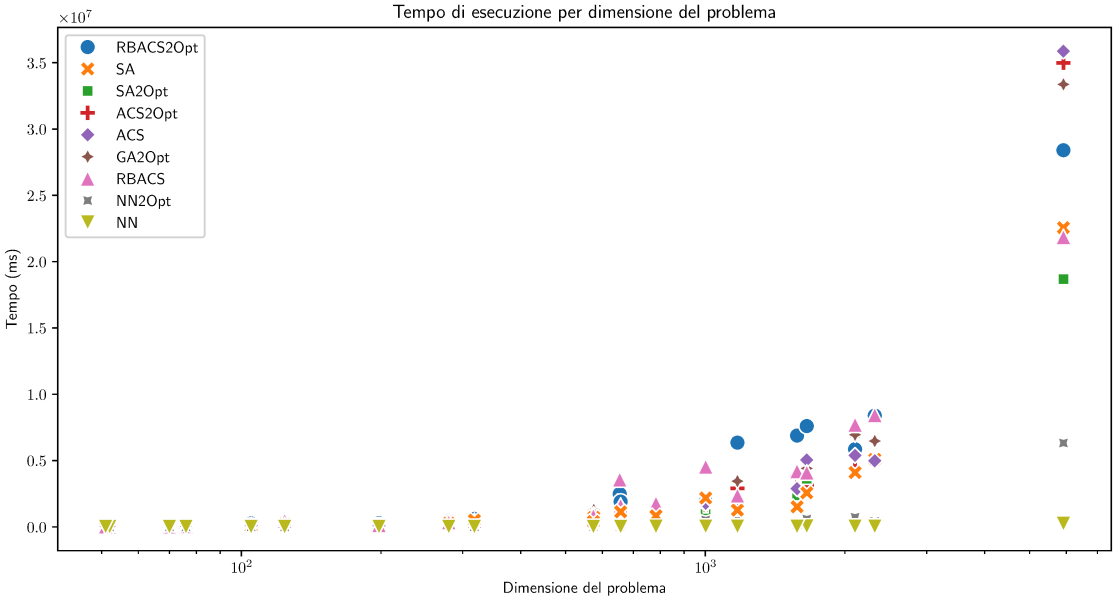


Figura 7.3: Esecuzione per dimensione del problema

1. L'importanza dell'ottimizzazione locale: l'aggiunta di 2-Opt migliora consistentemente le prestazioni di tutti gli algoritmi testati.

2. La superiorità degli approcci basati su colonie di formiche (ACS e RBACS) per istanze di medie e grandi dimensioni.

3. La variabilità delle prestazioni dell'algoritmo genetico, che suggerisce la necessità di un'attenta calibrazione dei parametri per ottenere buoni risultati.

4. Il compromesso tra tempo di calcolo e qualità della soluzione, con algoritmi più sofisticati che richiedono più tempo ma producono soluzioni migliori.

RBACS e RBACS2Opt emergono come approcci promettenti, offrendo prestazioni competitive e talvolta superiori rispetto ad ACS e ACS2Opt, specialmente su istanze di grandi dimensioni. Tuttavia, la scelta dell'algoritmo ottimale dipende dalle specifiche esigenze dell'applicazione, bilanciando la necessità di soluzioni di alta qualità con i vincoli di tempo di calcolo.

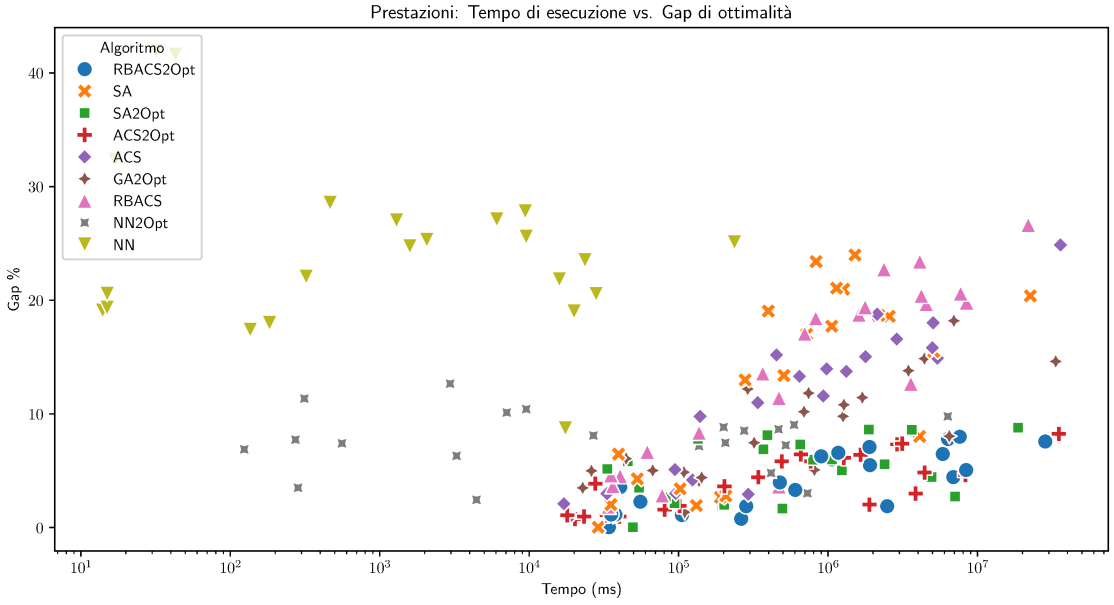


Figura 7.4: Gap per dimensione del problema

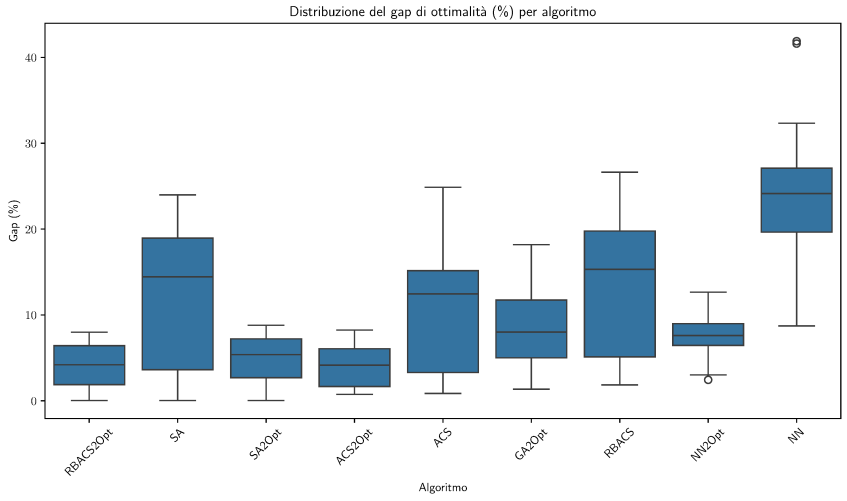


Figura 7.5: Confronto tra top 3 algoritmi: Gap

Capitolo 8

Conclusioni

In conclusione, questa tesi ha fornito un'analisi approfondita e comparativa di diversi approcci algoritmici per la risoluzione del Problema del Commesso Viaggiatore, con particolare attenzione alle prestazioni su istanze di grandi dimensioni. I risultati ottenuti non solo confermano l'efficacia di approcci metaeuristici consolidati come l'Ant Colony System, ma introducono anche una sua variante, il Red-Black Ant Colony System, che mostra potenziale per ulteriori miglioramenti e applicazioni.

8.1 Riflessioni Finali

L'evoluzione degli algoritmi per il TSP riflette una tendenza più ampia nel campo dell'ottimizzazione combinatoria: la ricerca di un equilibrio tra la qualità della soluzione e l'efficienza computazionale. Questa tesi ha dimostrato che:

- Non esiste un "algoritmo migliore" universale per il TSP. La scelta dell'approccio ottimale dipende dalle caratteristiche specifiche del problema, dalle dimensioni dell'istanza e dai vincoli computazionali.
- L'ibridazione di tecniche, come l'integrazione di procedure di ricerca locale (2-Opt) in metaeuristiche basate su popolazione, può portare a miglioramenti significativi nelle prestazioni.
- La scalabilità rimane una sfida cruciale, specialmente per problemi di grandissima scala (oltre 10,000 città). In questo contesto, approcci come RBACS che offrono maggiori opportunità di parallelizzazione diventano particolarmente rilevanti.
- La robustezza degli algoritmi, ovvero la loro capacità di mantenere buone prestazioni su una vasta gamma di istanze diverse, è un fattore critico per le applicazioni pratiche.

8.2 Osservazioni Conclusive

Il Problema del Commesso Viaggiatore, nonostante la sua apparente semplicità, continua a sfidare ricercatori e professionisti, spingendo i confini dell'ottimizzazione combinatoria e dell'informatica teorica.

Bibliografia

[1] M. R. Garey e D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979 (cit. a p. 8).

[2] D. E. Knuth. «Big Omicron and Big Omega and Big Theta». In: *ACM SIGACT News* 8.2 (1976), pp. 18–24 (cit. a p. 8).

[3] M. Dorigo e L. M. Gambardella. «Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem». In: *IEEE Transactions on Evolutionary Computation*. Vol. 1. 1. 1997, pp. 53–66 (cit. alle pp. 8, 36).

[4] M. Held e R. M. Karp. «A Dynamic Programming Approach to Sequencing Problems». In: *Journal of the Society for Industrial and Applied Mathematics* 10.1 (1962), pp. 196–210 (cit. a p. 9).

[5] C. M. Institute. *P vs NP Problem*. 2000. URL: http://www.claymath.org/millennium-problems/p-vs-np-problem (cit. a p. 10).

[6] R. M. Karp. «Reducibility Among Combinatorial Problems». In: *Complexity of Computer Computations* (1972), pp. 85–103 (cit. a p. 11).

[7] D. L. Applegate et al. *The Traveling Salesman Problem: A Computational Study*. Princeton, NJ, USA: Princeton University Press, 2007 (cit. a p. 17).

[8] G. Reinelt. *TSPLIB - A Traveling Salesman Problem Library*. Available online at http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. 1991. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (cit. alle pp. 18, 46).

[9] W. J. Cook. «In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation». In: *Princeton University Press* (2011) (cit. a p. 18).

[10] D. S. Johnson e L. A. McGeoch. «The Traveling Salesman Problem: A Case Study in Local Optimization». In: *Local Search in Combinatorial Optimization* (2002). A cura di E. H. Aarts e J. K. Lenstra, pp. 215–310 (cit. alle pp. 19, 20, 22, 24).

[11] T. H. Cormen et al. *Introduction to Algorithms*. 3rd. Cambridge, MA: MIT Press, 2009. ISBN: 9780262033848 (cit. a p. 19).

[12] C. H. Papadimitriou e K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Mineola, NY: Dover Publications, 1998. ISBN: 9780486402581 (cit. alle pp. 19, 20).

[13] E. L. Lawler et al. «The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization». In: *Wiley-Interscience Series in Discrete Mathematics and Optimization* (1985) (cit. alle pp. 19, 20, 22, 24).

[14] E. H. Aarts e J. H. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Chichester: John Wiley & Sons, 1989. ISBN: 9780471921462 (cit. a p. 19).

[15] G. Gutin e A. P. Punnen. «Heuristic Algorithms for the Traveling Salesman Problem». In: *The Traveling Salesman Problem and Its Variations*. Boston, MA: Springer, 2016, pp. 223–269. ISBN: 9781461302374 (cit. alle pp. 20, 24).

[16] S. Lin e B. W. Kernighan. «An effective heuristic algorithm for the traveling salesman problem». In: *Oper. Res.* 21 (1973), pp. 498–516 (cit. a p. 24).

[17] N. Metropolis et al. «Equation of state calculations by fast computing machines». In: *The Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092 (cit. a p. 26).

[18] M. Dorigo, V. Maniezzo e A. Colorni. «Ant system: optimization by a colony of cooperating agents». In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (1996), pp. 29–41 (cit. a p. 32).

[19] M. Dorigo e L. M. Gambardella. «Ant colony system: a cooperative learning approach to the traveling salesman problem». In: *IEEE Transactions on evolutionary computation* 1.1 (1997), pp. 53–66 (cit. alle pp. 32, 35).

[20] M. Dorigo, M. Birattari e T. Stutzle. «Ant colony optimization». In: *IEEE computational intelligence magazine* 1.4 (2006), pp. 28–39 (cit. a p. 32).

[21] T. Stützle e H. H. Hoos. «MAX–MIN ant system». In: *Future generation computer systems* 16.8 (2000), pp. 889–914 (cit. alle pp. 32, 33).

[22] M. Dorigo, G. Di Caro e L. M. Gambardella. «Ant colony optimization: a new meta-heuristic». In: *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*. Vol. 2. IEEE, 2004, pp. 1470–1477 (cit. a p. 32).

[23] B. Bullnheimer, R. F. Hartl e C. Strauss. «An improved ant system algorithm for the vehicle routing problem». In: *Annals of operations research* 89 (1999), pp. 319–328 (cit. a p. 35).

[24] M. Dorigo e T. Stützle. «Ant colony optimization: A literature review». In: *Swarm Intelligence* 4 (2010), pp. 1–41 (cit. a p. 35).

[25] M. Dorigo e L. M. Gambardella. «Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem». In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 53–66 (cit. alle pp. 36, 37).

[26] E. Bonabeau, M. Dorigo e G. Theraulaz. «Stigmergy: A novel mechanism of communication and control». In: *MIT Press* (1999) (cit. a p. 36).

[27] J.-L. Deneubourg et al. «The Self-Organizing Exploratory Pattern of the Argentine Ant». In: *Journal of Insect Behavior* 3.2 (1990), pp. 159–168 (cit. a p. 36).

[28] L. M. Gambardella e M. Dorigo. «Ant-Q: A Reinforcement Learning approach to the traveling salesman problem». In: *International Conference on Machine Learning*. 1995, pp. 252–260 (cit. a p. 37).

[29] T. Stuetzle e H. H. Hoos. «Max-Min Ant System for the Traveling Salesman Problem». In: *Proceedings of the IEEE International Conference on Evolutionary Computation*. 1997, pp. 309–314 (cit. a p. 41).

[30] M. H. Md. Rakib Hassan Md. Kamrul Hasan. «An Improved ACS Algorithm for the Solutions of Larger TSP Problems». In: *Proceedings of the ICEECE*. Dhaka, Bangladesh, 2013, pp. 1–5 (cit. alle pp. 42–44).