Analiza și Implementarea Algoritmilor pentru Problema Comis-Voiajorului (TSP)

AA Challenge 2025-2026

Drăghici Daria-Ioana
Făgețeanu Mariana-Cătălina
322CC

11 ianuarie 2026

Rezumat

Acest raport prezintă o analiză detaliată a Problemei Comis-Voiajorului (Traveling Salesman Problem - TSP), abordând atât metode exacte, cât și euristice. S-au implementat doi algoritmi distincti: algoritmul Held-Karp (bazat pe programare dinamică) pentru instanțe mici ($N \leq 20$) și algoritmul Simulated Annealing pentru instanțe mari, unde complexitatea timpului devine prohibitivă. Lucrarea include descrierea teoretică, detalii de implementare în limbajul C, o suită de teste generate automat și o analiză comparativă a performanței (timp de execuție vs. calitatea soluției). Rezultatele demonstrează compromisul clasic între optimalitate și resurse computaționale.

Cuprins

1 Introducere

Problema Comis-Voiajorului (TSP) este una dintre cele mai studiate probleme de optimizare combinatorială din informatică și cercetare operațională. Enunțul este simplu, dar soluționarea este complexă: *"Dat fiind o listă de orașe și distanțele dintre fiecare pereche de orașe, care este cel mai scurt traseu posibil care vizitează fiecare oraș exact o dată și se întoarce la orașul de origine?"*

TSP este o problemă NP-hard, ceea ce înseamnă că nu există un algoritm cunoscut care să o rezolve în timp polinomial pentru orice instanță. Importanța sa practică este vastă, având aplicații în logistică, proiectarea microcipurilor (VLSI), secvențierea ADN-ului și planificarea rutelor.

1.1 Descrierea problemei rezolvate

TSP poate fi modelată folosind teoria grafurilor. Fie $G = (V, E)$ un graf complet ponderat, unde V este mulțimea nodurilor (orașele), E este mulțimea muchiilor (drumurile directe), iar fiecărei muchii (u, v) îi este asociat un cost nenegativ $c(u, v)$. Obiectivul este găsirea unui ciclu Hamiltonian de cost minim. Un ciclu Hamiltonian este un traseu în graf care include fiecare nod din V exact o singură dată.
În cadrul acestui proiect, ne-am propus să rezolvăm problema,implementând și comparând două abordări fundamentale:

- **Abordarea Exactă:** Folosind algoritmul Held-Karp, care garantează găsirea optimului global, dar cu un cost exponențial de timp și memorie.
- **Abordarea Euristică:** Folosind Simulated Annealing (Călirea Simulată), un algoritm probabilistic care caută o soluție "suficient de bună" într-un timp rezonabil.

1.2 Exemple de aplicații practice pentru problema aleasă

TSP are numeroase aplicații practice în diverse domenii:

- **Logistică și distribuire:** planificarea rutelor pentru flote de vehicule, livrări rapide, colectarea deșeurilor, reduceri semnificative de combustibil, timp de muncă și uzură a vehiculelor.
- **Fabricarea circuitelor integrate:** găsirea căii optime pentru burghierea plăcilor cu circuite
- **Secvențierea ADN-ului:** asamblarea fragmentelor de ADN
- **Astronomie:** planificarea observațiilor telescopice pentru mai multe obiecte cerești
- **Producție:** programarea mașinilor pentru a minimiza timpul de setare între operații

2 Demonstrație NP-Hard

Definiție formală a TSP (varianta de decizie)

Fie graful neorientat ponderat $G=(V,E)$, unde V este mulțimea nodurilor, $|V|=n$ ($1 \le n \le 20$), iar $E \subseteq V \times V$ este mulțimea muchiilor, $|E|=M$

Fie $w: E \to \mathbb{R}^+$ funcția de ponderare (costuri)

Se cere să se găsească o permutare $\pi = (v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_{n-1}})$ a lui V astfel încât

- ciclul să fie Hamiltonian: $(v_{i_j}, v_{i_{j+1}}) \in E$ și $(v_{i_{n-1}}, v_{i_0}) \in E$, $\forall j \in \{0,1,2,\dots,N-2\}$
- să aibă cost minim: minimizează $C(\pi) = \sum_{i=0}^{n-2} w(v_i, v_{i+1}) + w(v_{n-1}, v_0), \forall i, (v_i, v_{i+1}) \in E$

Varianta de decizie (TSP-DECISION)

Fie graful neorientat ponderat $G=(V,E)$, unde V este mulțimea nodurilor, $|V|=n$ ($1 \le n \le 20$), iar $E \subseteq V \times V$ este mulțimea muchiilor, $|E|=M$

Fie $w: E \to \mathbb{R}^+$ funcția de ponderare (costuri)

Fie B un întreg pozitiv $\in \mathbb{Z}_+$

Se cere să se găsească o permutare $\pi = (v_{i_0}, v_{i_1}, v_{i_2}, \dots, v_{i_{n-1}})$ a lui V astfel încât

- ciclul să fie Hamiltonian: $(v_{i_j}, v_{i_{j+1}}) \in E$ și $(v_{i_{n-1}}, v_{i_0}) \in E$, $\forall j \in \{0,1,2,\dots,N-2\}$
- să aibă costul total $\le B$

O problemă de decizie P este în NP dacă

1. Pentru o instanță $\dot{I}$ a lui P, dacă răspunsul este DA, $\exists$ o soluție C
2. Se poate verifica în timp polinomial că C este corect pentru $\dot{I}$

O problemă P este în NP-completă dacă $P \in NP$ și P este NP-Hard

Probleme cunoscute NP-complete (din teorema lui Karp, 1972)

1. Hamiltonian Cycle Problem

   Fie graful neorientat $G=(V,E)$, unde V este mulțimea nodurilor, $|V|=n$ ($1 \le n \le 20$), iar $E \subseteq V \times V$ este mulțimea muchiilor.

   Se cere să se găsească un circuit Hamiltonian

2. Hamiltonian Path Problem

   Fie graful neorientat $G=(V,E)$, unde V este mulțimea nodurilor, $|V|=n$ ($1 \le n \le 20$), iar $E \subseteq V \times V$ este mulțimea muchiilor.

   Se cere să se găsească un drum Hamiltonian între 2 noduri date

3. TSP-Complet

   TSP pe graf complet

Demonstrația că TSP este NP-Hard

Vom demonstra în 2 pași

pasul 1: reducere de la Hamiltonian Cycle la TSP-Decision (pentru grafuri incomplete)

pasul 2: generalizare la cazul nostru specific

pasul 1: Reducere polinomială de la Hamiltonian Cycle Problem (HCP)

Pentru orice instanță $\dot{I}_{HC} = (G_1)$ a problemei Hamiltonian Cycle, unde $G_1 = (V, E_1)$, cu $|V|=n$, $|E|=M$, construim o instanță $\dot{I}_{TSP} = (G_2, w, B)$ a TSP-Decision astfel încât $G_2 = G_1$, w este funcția de cost definită $w(u,v) = \begin{cases} 1, & \text{dacă } (u,v) \in E_1 \\ n+1, & \text{dacă } (u,v) \notin E_1 \end{cases}$ $\forall u, v \in V$ și $B = n$ (numărul de noduri)

Pentru a construi $\dot{I}_{TSP}$ din $\dot{I}_{HC}$ vom copia graful G_1 în G_2 ($O(n+M)$)
vom atribui costuri ($O(1)$ per muchie existentă) $\Rightarrow O(n+M)$-timp polinomial

Demonstrația corectitudinii reducerii

„$\Rightarrow$" propoziția 1: Dacă G_1 are ciclu Hamiltonian, atunci $\dot{I}_{TSP}$ are soluție cu cost $\le n$

Demonstrație:

Fie $C = (v_0, v_1, \dots, v_{n-1}, v_0)$ un ciclu Hamiltonian în G_2

Toate muchiile din C există în E_1 (prin definiția ciclului Hamiltonian)

- $(v_i, v_{i+1}) \in E_1 \Rightarrow w(v_i, v_{i+1}) = 1$, $\forall i$
- $(v_{n-1}, v_0) \in E_1 \Rightarrow w(v_{n-1}, v_0) = 1$

Costul total $c_{TSP} = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) + w(v_{n-1}, v_0)$ $\Rightarrow C_{TSP} = \sum_{i=0}^{n-1} 1 = n = B$ $\Rightarrow \exists$ ciclu în TSP cu cost $n \Rightarrow C$ este soluție pentru $\dot{I}_{TSP}$ cu cost exact B

„<=" propoziția 2: Dacă TSP are soluție cu cost ≤ m, atunci G_2 are ciclu Hamiltonian

Demonstrație:

Fie $\overline{\pi_1} = (v_0, v_1, v_2, \dots, v_{n-1}, v_0)$ soluție în TSP cu $C(\overline{\pi_1}) \leq m$

Observăm că • ∀ muchii (x,y) în $\overline{\pi_1}$, $w(x,y) \geq 1$ (din definiție) $\Rightarrow C(\overline{\pi_1}) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) + w(v_{n-1}, v_0) \geq n \cdot 1 = n = B$

• numărul muchiilor în ciclu = n

$C(\overline{\pi_1}) \geq n = B$ dar știm că $C(\overline{\pi_1}) \leq n = B$ $\Rightarrow C(\overline{\pi_1}) = n = B \Rightarrow w(x,y) = 1$, ∀ muchii (x,y) consecutive în $\overline{\pi_1}$ dar toate muchiile din $\overline{\pi_1}$ sunt în E_2 $\Rightarrow \overline{\pi_1}$ este ciclu Hamiltonian în G_2

HC este NP-complet

∃ reducere polinomială HC -> TSP

reducerea păstrează răspunsul

Hamiltonian Cycle Problem este NP-completă (Karp, 1972) $\Rightarrow$ TSP-Decision pe grafuri incomplete este NP-Hard $\Rightarrow$ și problema noastră este NP-Hard

Problema noastră de optimizare (găsirea ciclului de cost minim este cel puțin la fel de dificilă ca TSP-Decision

pasul 2: Demonstrația că problema noastră este NP-Hard

propoziția 1: Dacă problema de optimizare ar putea fi rezolvată în timp polinomial, atunci și problema de decizie ar putea fi rezolvată în timp polinomial

Demonstrație:

Presupunem că avem un algoritm A care rezolvă problema de optimizare în timp polinomial

Pentru a rezolva TSP-Decision (G, w, B):

1. Rulează A pe (G, w) pentru a obține costul minim C
2. Compară C cu B:
   - dacă $C \leq B$, răspunsul este DA
   - dacă $C > B$, răspunsul este NU

$\Rightarrow$ acest algoritm ar fi polinomial <=> A ar fi polinomial

propoziția 2: Dacă TSP-Decision este NP-Hard, atunci și problema de optimizare TSP este NP-Hard

Demonstrație:

Presupunem prin reducere la absurd că problema de optimizare ar fi în P (rezolvabilă polinomial)

Din propoziția 1 $\Rightarrow$ TSP-Decision ar fi și ea în P

dar TSP-Decision pe grafuri incomplete este NP-Hard $\Rightarrow$ Vo cu $P \neq NP$ (presupunere standard în teoria complexității) => problema de optimizare TSP este NP-Hard

3 Prezentare Algoritmi

3.1 Arhitectura și Principiile Implementării

Simiar modului de generare a testelor, implementarea algoritmilor de rezolvare s-a bazat pe un set de principii software stricte, menite să asigure robustețea și corectitudinea rezultatelor indiferent de natura datelor de intrare.

3.1.1 Pilonii Implementării

Procesul de dezvoltare a urmărit patru direcții majore:

1. **Validarea Defensivă a Datelor:** Verificarea strictă a formatului de intrare și a limitelor (ex: $N \leq 0$, grafuri deconectate) înainte de lansarea execuției, pentru a preveni erorile de tip *Segmentation Fault*.
2. **Alocarea Statică vs. Dinamică:**
   - Pentru soluția exactă ($N \leq 20$), s-a optat pentru matrici statice globale (`dp[1<<20][20]`) pentru a evita overhead-ul alocării dinamice repetate și fragmentarea memoriei.
   - Pentru soluția aproximativă, structurile sunt compacte ($O(N^2)$), permițând procesarea rapidă în memoria cache a procesorului.
3. **Strategia "Fail-Fast":** Detectarea timpurie a cazurilor imposibile (ex: grafuri care nu satisfac condiția de existență a unui ciclu Hamiltonian) folosind o parcurgere DFS preliminară.
4. **Competiția Constructivă:** În loc de a se baza pe o singură euristică, soluția aproximativă generează multiple puncte de plecare și alege "campionul" pentru etapa de rafinare.

3.1.2 Structura Fișierelor Sursă

Implementarea este divizată în două module independente, fiecare având o arhitectură specifică scopului său. Mai jos este prezentată pseudostructura logică a fișierelor:

```
/* TSP_exact.c - Arhitectura solutiei exacte */
Variabile Globale:
    distanta[20][20], dp[1<<20][20], parinte[...];

Functii:
    initializeaza_tabele(): Pregateste starea initiala (INF)
    held_karp():            Nucleul de Programare Dinamica
    afiseaza_drumul():      Reconstructie prin Backtracking
    main():                 Gestionare I/O si validare

/* TSP_approx.c - arhitectura solutiei hibride */
Functii Utilitare:
    hamiltonian():          DFS pentru verificarea conexitatii
Componente Euristice:
    cel_mai_apropiat():     Constructie Greedy (Nearest Neighbor)
    insertie_simpla():      Constructie O(N^3) (Simple Insertion)
    optimizare_2opt():      Rafinare prin inversare muchii

Flux Principal (main):
    1. Validare input
    2. Rulare competitiva: (NN vs Insertion) -> Best_Init
    3. Rafinare: Best_Init -> 2-OPT -> Solutie Finala
```

3.2 Descrierea modului în care funcționează algoritmii aleși

3.2.1 Algoritmul Exact: Held-Karp

Algoritmul Held-Karp utilizează programare dinamică, bazându-se pe principiul optimalității: orice sub-traseu al unui traseu optim trebuie să fie el însuși un traseu optim, reducând complexitatea de la $O(N!)$ (forță brută) la $O(N^2 2^N)$.

Definim $DP(S, i)$ ca fiind costul minim al unui drum care pleacă din nodul de start (0), vizitează toate nodurile din subsetul $S \subseteq V$ (unde $0 \in S$ și $i \in S$) și se termină în nodul i.

Relația de recurență este:

$$DP(S, i) = \min_{j \in S, j \neq i} \{DP(S \setminus \{i\}, j) + c_{ji}\}$$

Cazul de bază (pentru mulțimi ce conțin doar nodul de start și un nod adiacent) este:

$$DP(\{0, i\}, i) = c_{0i}$$

Soluția finală a problemei TSP dată de minimizarea costului de întoarcere în origine este:

$$OPT = \min_{i \neq 0} \{DP(V, i) + c_{i0}\}$$

Această metodă necesită stocarea unei matrici de dimensiune $2^N \times N$, ceea ce limitează practic aplicarea sa la $N \leq 20$ pe mașinile moderne din cauza consumului de memorie.

Mai jos este prezentat nucleul implementării din fișierul `TSP_exact.c`, care iterează prin toate măștile posibile (submulțimi) pentru a construi soluția:

```
// iteram toate submultimile posibile de noduri (masti)
for (int masca = 1; masca <= masca_plina; masca++) {
    for (int u = 0; u < nr_noduri; u++) {
        if (dp[masca][u] == INFINIT) continue;

        // incercam sa extindem drumul la toate nodurile nevizitate
        for (int v = 0; v < nr_noduri; v++) {
            if ((masca & (1 << v)) == 0) { // daca nodul v nu e in masca
                int masca_noua = masca | (1 << v);
                int cost_tranzitie = dp[masca][u] + distanta[u][v];

                // Relaxarea costului
                if (cost_tranzitie < dp[masca_noua][v]) {
                    dp[masca_noua][v] = cost_tranzitie;
                    parinte[masca_noua][v] = u;
                }
            }
        }
    }
}
```

Listing 1: Implementarea recurenței Held-Karp

3.2.2 Implementare prin Bitmasking

Pentru a implementa eficient lucrul cu submulțimi, s-a utilizat tehnica **Bitmasking**. O submulțime S este reprezentată printr-un număr întreg, unde bitul k este setat la 1 dacă și numai dacă nodul $k \in S$.

- Intersecția mulțimilor devine operație `AND` pe biți.
- Excluderea unui element ($S \setminus \{i\}$) devine `mask ^ (1 << i)`.

3.3 Soluția Aproximativă: Euristici Hibride (Constructiv + 2-OPT)

Conform documentației tehnice a proiectului (*README.md*), soluția aproximativă utilizează o abordare hibridă eficientă, compusă din doi pași: construcție rapidă și optimizare locală.

Această strategie a fost aleasă pentru a balansa timpul de execuție redus cu o calitate rezonabilă a soluției, fiind capabilă să ruleze pe instanțe mari unde algoritmii exacți eșuează.

3.3.1 Etapa 1: Construcția Soluției (Nearest Neighbor)

Algoritmul Greedy **Nearest Neighbor** construiește turul alegând mereu cel mai apropiat oraș nevizitat. Deși rapid, este "mioapă" și poate duce la costuri mari spre finalul traseului.

- Algoritmul pornește dintr-un nod arbitrar.
- La fiecare pas, se selectează nodul nevizitat care are distanța minimă față de nodul curent.
- Complexitatea acestei etape este $O(N^2)$.

Deși această metodă este foarte rapidă, ea este "mioapă" (nu anticipează costurile viitoare), rezultând adesea în soluții sub-optime din cauza muchiilor lungi forțate la finalul traseului.

```
// algoritmul nearest neighbor
for (int i = 1; i < nr_noduri; i++) {
    int urmator = -1;
    int dist_min = INF;

    // Cautam cel mai apropiat vecin nevizitat
    for (int j = 0; j < nr_noduri; j++) {
        if (vizitat[j] == false && distante[curent][j] < dist_min) {
            dist_min = distante[curent][j];
            urmator = j;
        }
    }
    // ... actualizare traseu si cost ...
}
```

Listing 2: Construcția Greedy (Nearest Neighbor)

3.3.2 Etapa 2: Optimizarea Locală (2-OPT)

Pentru a corecta deficiențele soluției inițiale, se aplică algoritmul de căutare locală **2-OPT**. Acesta îmbunătățește iterativ traseul prin eliminarea încrucișărilor:

1. Se identifică două muchii non-adiacente din tur, (A, B) și (C, D).
2. Se verifică dacă inversarea segmentului dintre ele reduce costul total:

$$dist(A, C) + dist(B, D) < dist(A, B) + dist(C, D)$$

3. Dacă inegalitatea este adevărată, se efectuează inversarea (swap) și se reia verificarea.

Această combinație asigură un compromis optim între resursele utilizate și acuratețea rezultatului, fiind net superioară unei simple abordări Greedy.

```
// Verificam daca inversarea muchiilor (a,b) si (c,d) reduce costul
int cost_segment_vechi = distante[a][b] + distante[c][d];
int cost_segment_nou = distante[a][c] + distante[b][d];

if (cost_segment_nou < cost_segment_vechi) {
    // Inversam segmentul dintre i si j
    for (int k = 0; k < (j - i + 1) / 2; k++) {
        int aux = traseu_lucru[i + k];
        traseu_lucru[i + k] = traseu_lucru[j - k];
        traseu_lucru[j - k] = aux;
    }
    imbunatatit = true;
}
```

Listing 3: Optimizarea 2-OPT

3.4 Analiza complexității soluțiilor

3.4.1 Algoritmul Held-Karp

- **Complexitate temporală:** $O(N^2 \cdot 2^N)$
- **Complexitate spațială:** $O(N \cdot 2^N)$
- **Observație:** Pentru $N = 20$, algoritmul necesită aproximativ $20 \times 2^{20} = 20.971.520$ stări. Cu fiecare stare reprezentată ca un întreg de 4 bytes, consumul de memorie este de aproximativ 80 MB.

3.4.2 Algoritmul Aproximativ (Hibrid)

- **Complexitate temporală:** $O(N^3)$.
  - *Nearest Neighbor:* $O(N^2)$.
  - *Simple Insertion:* $O(N^3)$ (cauzat de căutarea poziției optime de inserare în turul curent la fiecare pas).
  - *2-OPT:* $O(K \cdot N^2)$, unde K este numărul de iterații (limitat la 100 în implementare).
- **Complexitate spațială:** $O(N^2)$ strict pentru matricea de distanțe.
- **Observație:** Diferența este colosală. Pentru $N = 20$, $N^3 = 8.000$ operații, ceea ce este neglijabil față de cele 400.000.000 ($N^2 2^N$) ale algoritmului exact. Această eficiență permite rularea testelor pe instanțe mari ($N = 1000$) în timp util.

3.5 Prezentarea principalelor avantaje și dezavantaje pentru soluțiile luate în considerare

3.5.1 Algoritmul Held-Karp

Avantaje:

- **Garanția Optimului Global:** Algoritmul explorează implicit întreg spațiul de soluții, garantând matematic găsirea celui mai scurt drum posibil.
- **Eficiență față de Brute Force:** Reduce complexitatea factorială $O(N!)$ la una exponențială $O(N^2 2^N)$, făcând posibile rezolvări pentru $N = 20$ care ar fi imposibile prin permutări simple.
- **Determinism:** Pentru aceleași date de intrare, algoritmul va produce întotdeauna exact același rezultat și același traseu, fără variații aleatoare.
- **Reconstrucția Traseului:** Structura matricei `parinte` permite o recuperare rapidă ($O(N)$) a ordinii nodurilor odată ce costul minim a fost calculat.

Dezavantaje:

- **Limitare Severă de Memorie (Memory Bound):** Acesta este cel mai mare dezavantaj. Matricea `dp[1<<20][20]` ocupă aproximativ 80MB. Pentru $N = 24$, necesarul de memorie ar depăși 1.2 GB, iar pentru $N = 30$ ar fi imposibil de alocat pe mașini standard.
- **Scalabilitate Inexistentă:** Timpul de execuție se dublează cu fiecare nod adăugat. Diferența dintre $N = 15$ și $N = 20$ este uriașă în termeni de timp de procesare.
- **Rigiditate:** Nu poate fi întrerupt pentru a oferi o "soluție parțială". Dacă algoritmul este oprit la 90% din execuție, nu oferă niciun rezultat utilizabil.

3.5.2 Algoritmul Aproximativ: Hibrid (Constructiv + 2-OPT)

Avantaje:

- **Scalabilitate Excelentă:** Complexitatea polinomială permite rularea pe instanțe cu sute sau mii de noduri în mai puțin de o secundă.
- **Eficiență a Memoriei:** Utilizează doar $O(N^2)$ spațiu (pentru matricea de distanțe), ceea ce este neglijabil chiar și pentru grafuri foarte mari.
- **Robustețe prin Strategia "Best-of":**
  - Codul nu se bazează pe o singură euristică. Acesta rulează *Nearest Neighbor* din mai multe puncte de start (0, 1, 2) și *Simple Insertion*, alegând cea mai bună variantă inițială. Aceasta reduce riscul de a alege un punct de start nefavorabil.
- **Corecție Locală:** Etapa de *2-OPT* este foarte eficientă în eliminarea erorilor "vizuale" evidente (încrucișări de drumuri) produse de algoritmii Greedy.

Dezavantaje:

- **Blocarea în Minime Locale:** Operatorul 2-OPT acceptă doar mutări care îmbunătățesc strict costul. Odată ajuns într-un minim local (o vale din care nu poate ieși doar coborând), algoritmul se oprește, chiar dacă optimul global este în apropiere.
- **Dependența de Soluția Inițială:** Calitatea finală a optimizării 2-OPT este strâns legată de calitatea turului construit inițial. Dacă *Nearest Neighbor* generează un traseu foarte încâlcit, 2-OPT ar putea să nu reușească să îl descurce complet.
- **Lipsa Garanției:** Nu există nicio limită teoretică strânsă asupra erorii. În cel mai rău caz, soluția poate fi semnificativ mai slabă decât optimul, deși în practică acest lucru este rar pe instanțe Euclidiene.

4 Evaluare

Setul de teste pentru Problema Comis-voiajorului (TSP) a fost proiectat meticulos pentru a acoperi o gamă largă de scenarii, de la cazuri simple și degenerate la structuri complexe și pattern-uri specifice. Scopul acestui set exhaustiv este de a verifica:

- Corectitudinea algoritmilor implementați
- Robustețea la inputuri diverse și edge cases
- Performanța în diferite condiții
- Calitatea soluțiilor aproximative față de cele optime

4.1 Descrierea modalității de construire a setului de teste folosite pentru validare

4.1.1 Generarea testelor

Generatorul de teste (generate_tests.py) a fost proiectat pentru a crea un set exhaustiv și diversificat de cazuri pentru problema TSP, acoperind toate scenariile posibile de la cele simple la cele complexe. Abordarea de generare se bazează pe cinci piloni fundamentali:

1. **Generare deterministă pentru reproducibilitate** - utilizarea seed-urilor fixe asigură că testele sunt identice la fiecare rulare

2. **Structuri geometrice cunoscute** - triunghiuri, cercuri, grid-uri, forme regulate cu proprietăți matematice bine definite

3. **Distribuții aleatoare controlate** - date pseudo-aleatoare cu caracteristici statistice specifice

4. **Pattern-uri matematice și structurale** - secvențe, simetrii, fractali și alte structuri cu proprietăți particulare

5. **Cazuri limită și inputuri invalide** - validarea robustețeiei algoritmilor la condiții extreme sau eronate

4.1.2 Structura Fișierului generate_tests.py

```
generate_tests.py
          Functii utilitare:
                 create_dirs(): creeaza directoarele de input/output
                 write_test(test_id, n, edges): scrie un test in
   formatul specificat
          34 functii specifice de generare:
                 generate_test00() ... generate_test33()
                 Fiecare implementeaza un pattern specific
          Functia principala:
               generate_all(): coordoneaza executia tuturor generarilor
```

4.1.3 Principii de Implementare

1. Reproductibilitate

```
random.seed(42)  # Seed fix pentru toate testele aleatoare
```

Fiecare funcție de generare care folosește aleatoriu setează un seed specific, asigurând că testele sunt perfect reproducibile.

2. Coverage Complet

```
def generate_all():
    tests = [
        generate_test00, generate_test01, ..., generate_test33  # 34 functii
    ]
    for i, test_func in enumerate(tests):
        test_func()  # Executie secventiala
```

Toate cele 34 de teste sunt generate într-o singură rulare, menținând ordinea logică de complexitate crescătoare.

3. Validare și Format Consistent

```
def write_test(test_id, n, edges):
    m = len(edges)
    with open(f"input/test{test_id:02d}.in", "w") as f:
        f.write(f"{n} {m}\n")  # Header: numar noduri si muchii
        for u, v, w in edges:
            f.write(f"{u} {v} {w}\n")  # Lista muchiilor (u, v, cost)
```

Formatul de output este standardizat, asigurând compatibilitatea cu ambele implementări TSP.

4.1.4 Descrierea fiecărui test

Tabela 1: Descrierea testelor generate (Testele 00-16)

Test	**Nume**	**Scop**
00	nr negativ de orașe	Verifică gestionarea erorilor pentru date imposibile matematic și returnarea corectă a mesajului "Nu există soluție."
01	Un oraș	Testează cazul degenerat al TSP (n = 1) și calculul corect al costului 0 pentru traseul 0→0
02	2 orașe	Verifică logica de bază pentru cel mai simplu caz non-trivial și calculul corect al costului dus-întors
03	Triunghi echilateral	Testează simetria perfectă și faptul că toate permutările au același cost în structuri complet simetrice
04	pătrat	Verifică capacitatea de a găsi traseul optim (perimetrul) și de a evita diagonalele ineficiente
05	7 orașe coliniare	Testează gestionarea cazurilor degenerate și calculul optimului pe o structură perfect liniară
06	Cerc perfect (16 puncte)	Verifică comportamentul pe structuri circulare simetrice cu multiple soluții optime echivalente
07	Distribuție uniformă aleatoare	Testează performanța pe input impredictibil fără structuri geometrice evidente
08	Grid regulat (costuri Manhattan)	Verifică algoritmii pe structuri de rețea regulate cu metrici non-euclidiene
09	Doua clustere	Testează capacitatea de a identifica grupurile și strategia de a vizita complet un cluster înainte de salt
10	Linie dreaptă cu noise	Verifică robustețea la mici perturbații ale structurilor perfecte și stabilitatea algoritmilor
11	Costuri mari	Testează gestionarea numerelor mari, prevenirea overflow-ului și tipurile de date adecvate
12	Dimensiune maximă	Verifică scalabilitatea și limitele algoritmului Held-Karp la n maxim cu costuri extreme
13	Costuri foarte mici	Testează precizia la numere mici și gestionarea erorilor de rotunjire în comparații
14	Scale mixte exreme	Verifică stabilitatea pe costuri cu ordine de mărime diferite (de la 1 la 1M)
15	Numere întregi puteri ale lui 2	Testează operațiile pe numere cu structură binară și proprietăți matematice specifice

Test	Nume	Scop
16	Costuri cu structură complexă	Verifică capacitatea de a găsi soluții în spații fără pattern-uri geometrice evidente
17	Muchii duplicate	Testează parsarea corectă a inputului și păstrarea costului minim pentru muchii multiple
18	Structură spirală	Verifică comportamentul pe structuri ordonate crescător și problema rămânerii în zone cu index mic
19	Formă de stea	Testează gestionarea nodurilor cu grad mare și strategia de a vizita nodurile cu cost mic la momentul potrivit
20	Performanță medie	Verifică timpul de execuție și compararea directă exact vs aproximativ la dimensiune maximă
21	Graf incomplet	Testează detectarea imposibilității soluției și distincția între conectivitate și existența ciclului
22	Cluster extrem	Verifică capacitatea de a traversa între clustere dense și problema rămânerii blocat într-un cluster
23	Structură pe "axe"	Testează gestionarea structurilor pe "axe" separate și alegerea ordinii de vizitare a seturilor
24	Simetrie perfectă	Verifică comportamentul pe grafuri simetrice și echivalența tururilor prin permutări simetrice
25	Grid cu pattern alternant	Testează algoritmii pe structuri cu pattern alternant și exploatarea simetriei și periodicității
26	Graf extrem de rar (n=20)	Verifică gestionarea grafelor sparse și capacitatea de a identifica ciclurile unice
27	Costuri unitare	Testează cazul degenerat de costuri constante și simplificările posibile pentru costuri egale
28	Input incomplet	Verifică gestionarea EOF prematur și validarea completă înainte de procesare
29	Caractere nonnumerice	Testează parsarea robustă și gestionarea erorilor pentru format corupt
30	Lipsă n	Verifică gestionarea whitespace-ului în exces și distincția între EOF și spații
31	"Spike" pattern (n=10)	Testează robustețea la valori extrem de mari și strategiile de a evita outlier-i costisitori
32	Grid cu găuri (n=16)	Verifică algoritmii pe grafuri incomplete și capacitatea de a ocoli muchii lipsă
33	Structură fractală simulată (n=15)	Testează algoritmi care optimizează la diferite scale și conceptul de auto-similaritate

4.2 Specificațiile sistemului de calcul pe care ați rulat testele

Sistem de Operare: Windows 11 Pro (Host) cu WSL2, folosind kernel Linux 6.6.87.2-microsoft-standard-WSL2 și distribuția Ubuntu 24.04 LTS.

Procesor: Intel(R) Core(TM) i7-10710U CPU @ 1.10GHz (6 nuclee, 12 threads, arhitectură Comet Lake, 64-bit)

Memorie RAM: 16 GB DDR4

Memorie Cache (CPU): L1d: 192 KiB, L1i: 192 KiB, L2: 1.5 MiB, L3: 12 MiB

Stocare: SSD 1 TB (NVMe), cu 937 GB disponibili pentru sistemul de fișiere WSL

Compiler: GCC 13.3.0 (Ubuntu 13.3.0-6ubuntu2 24.04)

Optimizări: -O2 (optimizare pentru viteză fără creștere semnificativă a dimensiunii codului) și -std=c99 (standardul limbajului C)

Mediu de Execuție: WSL2 (Windows Subsystem for Linux 2) cu distribuția Ubuntu 24.04, rulat în terminalul Bash.

4.3 Ilustrarea rezultatelor evaluării soluțiilor pe setul de teste

4.3.1 Comparație Costuri Exacte vs Aproximative

Tabela 2

Test	Cost sol. exactă	Cost sol approx	Ratio	Status
00	Nu există soluție	Nu există soluție	1.00	✓
01	0	0	1.00	✓
02	20	20	1.00	✓
03	300	300	1.00	✓
04	400	400	1.00	✓
05	600	600	1.00	✓
06	600	600	1.00	✓
07	322	322	1.00	✓
08	960	960	1.00	✓
09	555	555	1.00	✓
10	690	693	1.004	∼
11	8273163	8388098	1.013	∼
12	18201058	18226104	1.001	∼
13	15	15	1.00	✓
14	546	546	1.00	✓
15	104	104	1.00	✓
16	2061700	2864168	1.389	∼
17	844	844	1.00	✓
18	2850	2850	1.00	✓
19	1180	1180	1.00	✓
20	2679	3232	1.206	∼
21	Nu există soluție	Nu există soluție	1.00	✓
22	5949	Nu există soluție	N/A	×
23	720	720	1.00	✓
24	2212	2212	1.00	✓
25	900	900	1.00	✓
26	2000	2000	1.00	✓
27	10	10	1.00	✓
28	Nu există soluție	Nu există soluție	1.00	✓
29	Nu există soluție	Nu există soluție	1.00	✓
30	Nu există soluție	Nu există soluție	1.00	✓
31	20110	21113	1.049	∼
32	960	1080	1.125	∼
33	4500	4504	1.00	✓

4.3.2 Distribuția raportului Aproximativ/Exact

Tabela 3

Interval Raport	Număr Teste	Procent
1.00	24	70.6%
1.00 - 1.05	5	14.7%
1.05 - 1.10	1	2.9%
1.10-1.20	2	5.9%
1.20-1.40	1	2.9%
Fără soluție	1	2.9%

4.3.3 Analiza Timpilor de Execuție

Tabela 4: Analiza Timpilor de Execuție (Testele 00-16)

Test	Descriere	Timp soluție exactă	Timp soluție aproximativă	Exact_Time / Approx_Time
00	nr negativ de orașe	0.01 s	0.01 s	1x
01	Un oraș	0.01 s	0.01 s	1x
02	2 orașe	0.17 s	0.01 s	17x
03	Triunghi echilateral	0.08 s	0.01 s	8x
04	pătrat	0.08 s	0.01 s	8x
05	7 orașe coliniare	0.09 s	0.01 s	9x
06	Cerc perfect (16 puncte)	0.07 s	0.01 s	7x
07	Distribuție uniformă aleatoare	0.08 s	0.01 s	8x
08	Grid regulat (costuri Manhattan)	0.11 s	0.01 s	11x
09	Doua clustere	0.08 s	0.01 s	8x
10	Linie dreaptă cu noise	0.08 s	0.01 s	8x
11	Costuri mari	0.09 s	0.01 s	9x
12	Dimensiune maximă	0.50 s	0.02 s	25x
13	Costuri foarte mici	0.08 s	0.01 s	8x

Tabela 5: Analiza Timpilor de Execuție (Testele 17-33)

Test	**Descriere**	**Timp soluție exactă**	**Timp soluție aproximativă**	**Exact_Time / Approx_Time**
14	Scale mixte	0.08 s	0.01 s	8x
15	Numere întregi puteri ale lui 2	0.07 s	0.01 s	7x
16	Costuri cu structură complexă	0.09 s	0.01 s	9x
17	Muchii duplicate	0.08 s	0.01 s	8x
18	Structură spirală	0.53 s	0.01 s	53x
19	Formă de stea	0.16 s	0.01 s	16x
20	Performanță medie	0.54 s	0.01 s	54x
21	Graf incomplet	0.08 s	0.01 s	8x
22	Cluster extrem	0.28 s	0.01 s	28x
23	Structură pe ”axe”	0.09 s	0.01 s	9x
24	Simetrie perfectă	0.08 s	0.01 s	8x
25	Grid cu pattern alternant	0.09 s	0.01 s	9x
26	Graf extrem de rar (n=20)	0.10 s	0.01 s	10x
27	Costuri unitare	0.07 s	0.01 s	7x
28	Input incomplet	0.00 s	0.01 s	N/A
29	Caractere non-numerice	0.08 s	0.01 s	8x
30	Lipsă n	0.01 s	0.01 s	1x
31	”Spike” pattern (n=10)	0.14 s	0.01 s	14x
32	Grid cu găuri (n=16)	0.09 s	0.01 s	9x
33	Structură fractală simulată (n=15)	0.09 s	0.01 s	9x

4.4 Interpretarea, succintă, a valorilor obținute pe teste. Dacă apar valori neașteptate, încercați să oferiți o explicație

4.4.1 Analiza Performanței Generală

Algoritmul Exact (Held-Karp)
Implementarea exactă bazată pe programare dinamică cu măști de biți a demonstrat 100% corectitudine pe toate testele valide, găsind întotdeauna soluția optimă atunci când aceasta există. Complexitatea algoritmului de $O(n^2 \cdot 2^n)$ s-a reflectat în timpii de execuție care cresc exponențial cu n:

- Pentru $n \leq 12$: sub 0.1 secunde
- Pentru $n = 20$: până la 0.54 secunde

Implementarea actuală limitează n la 20 din cauza alocării statice a tabelelor dp[1¡¡20][20]. Acest compromis între performanță și utilizarea memoriei este tipic pentru Held-Karp.

Algoritmul Aproximativ (Combinație de Euristici)
Soluția aproximativă a demonstrat o performanță remarcabilă în majoritatea cazurilor:

- 70.6% din teste (24/34): găsește soluția exact optimă
- 85.3% din teste (29/34): eroare $\leq 5\%$
- 8.8% din teste (3/34): eroare 5-20%
- 2.9% din teste (1/34): eșec complet

Speedup mediu: 14× față de soluția exactă, crescând exponențial cu n (de la 8× la n=10 la 54× la n=20).

4.4.2 Analiza Testelor cu Rezultate Remarcabile

Test 16: Cel Mai Slab Raport (1.389)
Context: Structură pseudo-aleatorie complexă generată prin funcția (i*123456789 + j*987654321) % 1000000 + 1.
Interpretare:

- Euristica obține un cost cu 38.9% mai mare decât optimul
- Acest test evidențiază principalul defect al algoritmilor greedy: alegerile local-optime pot duce la soluții global-suboptime
- Nearest Neighbor face alegeri bune la fiecare pas, dar acestea se însumează într-o soluție departe de optim

Explicație: Funcția de generare creează un pattern aparent aleatoriu care îi păcălește pe algoritmii greedy. Lipsa unei structuri geometrice recognoscibile face ca euristica să nu poată exploata pattern-uri pentru a găsi soluții bune.

Test 22: Eșec Complet
Context: Două clustere extreme conectate printr-un "pod" subțire.
Interpretare:

- Exact: găsește soluție cu cost 5949
- Aproximativ: returnează "Nu există soluție"
- Acesta este singurul test unde euristica eșuează complet

Explicație: Funcția hamiltonian() din implementarea aproximativă verifică doar conectivitatea grafului (prin DFS), nu existența unui ciclu Hamiltonian. Deși graful are un ciclu Hamiltonian, structura sa extremă (două clustere dense conectate prin doar 9 muchii din 100 posibile) face ca verificarea să eșueze. Această verificare pre-procesare prea restrictivă ar trebui eliminată sau îmbunătățită.

Test 31: Performanță Surprinzător de Bună (raport 1.000)
Context: "Spike pattern" cu 9 noduri apropiate și un outlier extrem (costuri de 10,000).
Interpretare:

- Exact: 20110
- Aproximativ: 20113 (diferență de doar 3 unități, 0.015%)
- Performanță excelentă în ciuda unui outlier extrem

Explicație: Algoritmul de inserție simplă gestionează bine outlier-ii pentru că îi inserează la poziția optimă, minimizând impactul asupra costului total. Acest rezultat contrazice intuiția că outlier-i mari ar trebui să degradeze semnificativ performanța euristicilor greedy.

Test 32: Performanță Acceptabilă pe Graf Incomplet (raport 1.125)
Context: Grid 4×4 cu patru muchii lipsă ("găuri").
Interpretare:

- Exact: 960
- Aproximativ: 1080 (eroare 12.5%)
- Performanță decentă pentru un graf incomplet

Explicație: Euristica gestionează relativ bine muchiile lipsă prin mecanismul său de căutare. Când nu găsește o muchie necesară (cost INF), caută alternative. Acest comportament adaptiv explică performanța acceptabilă în ciuda incompletitudinii grafului.

4.4.3 Analiza Timpilor de Execuție

Algoritmul Exact (Held-Karp)
Pattern observat: Timpul crește exponențial cu n, dar cu variații semnificative:

- Testele 18, 20, 22: timpi mari (0.53-0.54s) pentru n=20
- Testul 12: 0.50s pentru n=20 cu costuri mari
- Testul 19: 0.16s pentru n=16 (structură de stea)

Explicație: Variațiile se datorează: 1. **Numărul de muchii:** Grafuri complete (toate testele) au același număr de muchii 2. **Distribuția costurilor:** Costuri extreme pot cauza mai multe comparații 3. **Structura grafului:** Unele structuri permit pruning mai eficient în DP

Algoritmul Aproximativ
Pattern observat: Timp aproape constant (0.01s) indiferent de n sau structură.
Explicație: Complexitatea polinomială ($O(n^3)$ pentru 2-OPT) asigură timpi constanți pentru n 20. Acesta este principalul avantaj practic al euristicii față de soluția exactă.

Speedup Crescând cu n
Observație: Speedup-ul (Exact_Time / Approx_Time) crește dramatic cu n:

- n=2: 17× (dar timp absolut mic)
- n=10: 8×
- n=20: 25-54×

Implicație: Pentru n ¿ 20 (în practică), speedup-ul ar fi enorm, făcând euristica singura opțiune fezabilă.

4.4.4 Valori Neașteptate și Explicații Lor

Test 12: Performanță Excelentă la n=20 (raport 1.001)
Așteptat: Euristica ar putea avea performanță mai slabă la dimensiune maximă.
Observat: Diferență de doar 0.14% față de optim.
Explicație: Datele aleatoare uniforme sunt "ușoare" pentru euristici pentru că nu au structuri complexe care să înșele algoritmii greedy. Nearest Neighbor performează bine pe distribuții uniforme.

2. Test 10 vs Test 05: Robustețe la Zgomot
Așteptat: Zgomotul ar putea degrada semnificativ performanța.
Observat:

- Test 05 (perfect): raport 1.00
- Test 10 (cu zgomot): raport 1.004

Explicație: Euristica este robustă la mici perturbații datorită mecanismului 2-OPT care corectează alegerile suboptime făcute de Nearest Neighbor datorită zgomotului.

Test 18: Speedup Extrem (53×) dar Performanță Perfectă
Context: Structură spirală cu n=20.
Observat: Exact: 0.53s, Aproximativ: 0.01s, raport cost: 1.00
Explicație: Structura ordonată a spiralii este perfect exploatabilă de Nearest Neighbor, care găsește soluția optimă imediat. Held-Karp trebuie să exploreze toate subseturile în ciuda structurii simple.

Test 33: Diferență Minoră Neașteptată (4500 vs 4504)
Context: Structură fractală.
Observat: Diferență mică (4 unități) în ciuda structurii complexe.
Explicație: Structura fractală, deși complexă, are o organizare recursivă care poate fi exploatabilă parțial de euristică. Diferența mică sugerează că euristica a fost aproape de optim dar a rămas blocată într-un minim local.

4.4.5 Rezumarea observațiilor despre teste

Pentru Algoritmul Exact (Held-Karp):
1. **Scalabilitatea limitată** este prețul pentru optimalitate garantată
2. **Utilizarea memoriei** devine problematică pentru n mai mare decât 20 (2^{20} stări)
3. **Performanța consistentă** indiferent de structura datelor

Pentru Algoritmul Aproximativ:
1. **Verificările pre-procesare** pot fi prea restrictive (Test 22)
2. **Structurile complexe artificiale** sunt cele mai dificile (Test 16)
3. **Outlierii extremi** nu sunt întotdeauna problemă (Test 31)
4. **Speedup-ul enorm** față de exact justifică utilizarea în practică

Compromisuri Evidențiate:
1. **Optimalitate vs Timp:** Euristica oferă speedup de până la 54× cu eroare medie sub 5%
2. **Memorie vs n:** Held-Karp limitează n la 20 din cauza alocării memoriei
3. **Robustețe vs Complexitate:** Euristica e robustă la zgomot dar vulnerabilă la structuri patologice

5 Concluzii

5.1 Precizarea, în urma analizei făcute, cum am aborda problema în practică; în ce situații am opta pentru una din soluțiile alese

În urma analizei exhaustive a ambelor abordări și a rezultatelor experimentale obținute, concluzionăm că nu există un algoritm universal, iar alegerea metodei optime depinde strict de constrângerile operaționale ale problemei practice (Timp vs. Calitate vs. Resurse):

Situații în care alegeți algoritmul exact (Held-Karp):

- Când dimensiunea problemei este mică ($N \leq 20$), unde garanția matematică a optimului primează.
- Când este critică găsirea soluției optime absolute (ex: aplicații de siguranță, optimizarea circuitelor VLSI, sisteme critice medicale).
- Când aveți resurse de calcul suficiente, în special memorie RAM, deoarece algoritmul este *Memory Bound* ($O(N \cdot 2^N)$ spațiu).
- Pentru validarea și benchmarking-ul altor algoritmi euristici (folosit ca "Ground Truth" pentru calcularea erorii relative).

Situații în care alegeți algoritmul euristic (Hibrid: Constructiv + 2-OPT):

- Pentru instanțe mari ($N > 20$) și foarte mari ($N > 100$), unde abordarea exactă devine imposibilă fizic.
- Când timpul de răspuns este critic (aplicații în timp real, sisteme GPS, rutare dinamică), soluția hibridă oferind un răspuns aproape instantaneu ($O(N^3)$).
- Când resursele de calcul sunt limitate (sisteme embedded, dispozitive mobile) care nu pot aloca matrici de dimensiuni exponențiale.
- Când o soluție "suficient de bună" este acceptabilă (o eroare de 1-5% este tolerabilă în schimbul vitezei).
- Pentru probleme practice din lumea reală, unde strategia "Best-of" (selecția dintre Nearest Neighbor și Simple Insertion) evită eficient capcanele minimelor locale triviale.

Recomandarea noastră practică: Dacă am implementa un "solver" comercial pentru TSP, am adopta o Arhitectură Adaptivă care:

1. **Analizează inputul:** Verifică dimensiunea N și densitatea grafului.
2. **Ramura Exactă ($N \leq 20$):** Activează automat Held-Karp pentru a garanta optimul global, profitând de faptul că pe hardware modern execuția este rapidă pentru N mic.
3. **Ramura Euristică ($N > 20$):** Activează algoritmul hibrid. Pentru a crește robustețea, se poate extinde codul actual pentru a rula *2-OPT* din multiple puncte de start (Multistart Local Search), crescând șansele de a găsi optimul global fără a sacrifica performanța.
4. **Validare:** Adaugă un mecanism care compară soluția găsită cu o limită inferioară teoretică (ex: *1-Tree Lower Bound*) pentru a oferi utilizatorului o garanție asupra calității (ex: "Soluția este la maxim 3% de optim").

Această abordare hibridă și adaptivă asigură optimalitatea pentru probleme mici și fezabilitatea pentru probleme mari, oferind cel mai bun compromis ingineresc între calitatea soluției și resursele computaționale consumate.

6 Bibliografie

1. Held, M., & Karp, R. M. (1962). *A dynamic programming approach to sequencing problems*. Journal of the Society for Industrial and Applied Mathematics.
2. Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). *Optimization by Simulated Annealing*. Science.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
4. Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.
5. Johnson, D. S., & McGeoch, L. A. (1997). *The traveling salesman problem: A case study in local optimization*. Local search in combinatorial optimization.
6. GeeksforGeeks. (2023). *Traveling Salesman Problem using Dynamic Programming*. Disponibil la: `https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/`
7. Wikipedia contributors. (2023). *2-opt algorithm*. Disponibil la: `https://en.wikipedia.org/wiki/2-opt`
8. Wikipedia contributors. (2023). *Nearest neighbour algorithm*. Disponibil la: `https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm`
9. Python Software Foundation. (2023). *Python 3.11 Documentation: random module*. Disponibil la: `https://docs.python.org/3/library/random.html`
10. Python Software Foundation. (2023). *Python 3.11 Documentation: argparse module*. Disponibil la: `https://docs.python.org/3/library/argparse.html`
11. GNU Project. (2023). *The GNU C Library Reference Manual*. Disponibil la: `https://www.gnu.org/software/libc/manual/`
12. Kernighan, B. W., & Ritchie, D. M. (1988). *The C Programming Language* (2nd ed.). Prentice Hall.
13. Lin, S., & Kernighan, B. W. (1973). *An effective heuristic algorithm for the traveling-salesman problem*. Operations Research.
14. Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.
15. Rosenkrantz, D. J., Stearns, R. E., & Lewis, P. M. (1977). *An analysis of several heuristics for the traveling salesman problem*. SIAM Journal on Computing.
16. Croes, G. A. (1958). *A method for solving traveling-salesman problems*. Operations Research.
17. Held, M., & Karp, R. M. (1970). *The traveling-salesman problem and minimum spanning trees*. Operations Research.
18. Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag.
19. Bentley, J. L. (1992). *Fast algorithms for geometric traveling salesman problems*. ORSA Journal on Computing.
20. Karp, R. M. (1972). *Reducibility Among Combinatorial Problems*. In R. E. Miller & J. W. Thatcher (Eds.), Complexity of Computer Computations. Plenum Press.