



UNIVERSITATEA DIN
BUCUREŞTI



FACULTATEA
DE
MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licență

**Problema comis-voiajorului: Algoritmi – Studiu
comparativ**

Absolvent

Denisa Predescu

Coordonator științific

Lect.dr. Ruxandra Marinescu-Ghemeci

București, iunie-iulie 2023

Rezumat

Prin această lucrare mi-am propus să discut despre problema comis-voiajorului (TSP), examinând rezultate cunoscute și aspecte algoritmice prin care a fost încercată rezolvarea ei. Problema dorește determinarea celui mai scurt drum pe care un comis-voiajor îl poate parcurge pentru a trece prin n orașe o singură dată și a se întoarce, la final, la cel de origine știind că fiecare distanță dintre două orașe are o anumită lungime stabilită anterior.

TSP o problemă de minimizare complexă care face parte din clasa problemelor NP-complete. Prin urmare, încercarea de a determina ciclul minim devine dificilă când instanța are un număr considerabil de orașe. În această lucrare, se prezintă trei tipuri de algoritmi cunoscuți pentru TSP, și anume algoritmi exacți, algoritmi de aproximare și algoritmi euristică de tip greedy. Pentru fiecare categorie sunt discutați algoritmi, propunându-se pentru fiecare dintre aceștia câte o modalitate de rezolvare prezentată și explicată prin pseudocod, apoi implementată în Python.

La final sunt realizate studii comparative între algoritmii de același tip și nu numai. În comparații se pune accentul pe timpul de rulare și apropierea soluțiilor găsite de cea optimă. Instanțele folosite sunt preluate din biblioteca online *TSPLIP* și de pe *infoarena*.

De asemenea, este implementată o aplicație web cu scopul de a fi de ajutor altor persoane care doresc înțelegerea conceptelor și a algoritmilor. Se vine în sprijinul lor cu exemple sugestive, prezentări pas cu pas, rulare de cod și o sinteză a informațiilor discutate pe parcursul acestei lucrări.

Through this paper I have proposed to discuss the traveling salesman problem (TSP), examining the known results and the algorithmic aspects by which it has been attempted to be solved. The problem seeks to determine the shortest route that a traveling salesman can take to

pass through n cities only once and return at the end to his home city knowing that each distance between two cities has a predetermined length.

TSP is a complex minimization problem that is part of the NP-complete class. Therefore, the way to determine the minimum cycle becomes difficult when the instance has a considerable number of cities. In this paper, three types of known algorithms for TSP are presented, namely exact algorithms, approximation algorithms, and greedy heuristic algorithms. For each category, algorithms are presented, proposing for each of them a solution presented and explained by pseudocode, then implemented in Python.

At the end, comparative studies are carried out between algorithms of the same type and not only. In the comparisons, the focus is on the running time and the proximity of the found solutions to the optimal one. The instances used are taken from the *TSPLIP* online library and *infoarena*.

A web application is also implemented to help others who want to understand the concepts and algorithms. It supports them with suggestive examples, step-by-step presentations, running code on their own instances, and a synthesis of the information discussed throughout this paper.

Cuprins

Introducere	6
1. Problema comis-voiajorului	8
1.1. Definirea problemei.....	8
1.2. Variații.....	10
1.2.1. TSP asimetric – aTSP	10
1.2.2. Metric TSP	13
1.2.3. Alte variații.....	13
1.3. NP-completitudine.....	16
2. Algoritmi exacti	18
2.1. Programarea dinamică	18
2.2. Branch and Bound	23
3. Algoritmi de aproximare	31
3.1. Factor de aproximare pentru TSP.....	31
3.2. Algoritmul arborelui dublat.....	33
3.3. Algoritmul lui Christofides.....	41
4. Algoritmi euristicici de tip greedy	56
4.1. Construirea unui ciclu hamiltonian.....	56
4.1.1. Algoritmul de inserție a celui mai îndepărtat nod.....	57
4.1.2. Algoritmul de inserție a celui mai apropiat nod	60
4.1.3. Algoritmul de inserție cu cel mai mic cost.....	61
4.1.4. Algoritmul cel mai apropiat vecin.....	65
4.2. Micșorarea costului unui ciclu hamiltonian.....	67
4.2.1. Algoritmul 2-OPT	69
4.2.2. Algoritmul 3-OPT	72
5. Comparații între algoritmi	77
5.1. Algoritmi exacti	78
5.2. Algoritmi approximativi	80
5.3. Algoritmi euristicici	87
5.4. Compararea algoritmilor approximativi și euristicici	94

6. Aplicație web	95
6.1. Tehnologii utilizate.....	95
6.1.1. Flask.....	95
6.1.2. TypeScript	96
6.1.3. Angular	97
6.1.4. Graphology și SigmaJs.....	99
6.2. Arhitectura aplicației create.....	99
6.2.1. Back-end-ul aplicației.....	101
6.2.2. Front-end-ul aplicației.....	103
6.3. Funcționalități aplicație	105
6.3.1. Validarea inputului.....	105
6.3.2. Schimbarea limbii afișate.....	108
6.3.3. Integrarea de grafuri dinamice	109
6.3.4. Integrarea code-editorului	111
6.3.5. Crearea secțiunii de input-output	112
6.3.6. Integrarea exemplelor pas cu pas.....	113
6.3.7. Integrarea sliderului pentru poze	113
6.4. Prezentarea aplicației.....	114
7. Concluzii	124
Bibliografie	126

Introducere

Problema comis-voiajorului este o problemă complexă și foarte cunoscută fiind creați o multitudine de algoritmi cu scopul de a rezolva corect sau de a obține o soluție cât mai apropiată de soluția optimă.

Problema este o generalizare a unei alte probleme cunoscute din teoria grafurilor, problema ciclului hamiltonian care este una din cele mai vechi probleme NP-complete. Fiind o problemă dificilă din punct de vedere computațional, există mai multe rezultate teoretice și algoritmi dezvoltăți în literatură.

Scopul acestei lucrări este a prezenta aspecte algoritmice legate de problema comis voiajorului fiind realizate studii comparative între diferite tipuri de algoritm și implementată o aplicație web cu scopul de a facilita documentarea și înțelegerea algoritmilor.

În ziua de azi sunt multe mijloace de căutare și de informare, dar din păcate, de multe ori informațiile prezentate sunt incomplete sau greu de urmărit, utilizatorii fiind nevoiți să acceseze mai multe surse. Aplicația dezvoltată vine în ajutorul celor dormici de cunoaștere, oferindu-le un site simplist, ușor de navigat prin el și în care algoritmii sunt împărțiți pe secțiuni pentru a fi avea o structură logică. Sunt prezentate multiple mijloace pentru a putea ține pasul cu multitudinea informațiilor. Pentru a le oferi o experiență plăcută și folositoare, aplicația conține exemple pas cu pas a multor dintre algoritmii prezenți, în cazul în care nu au fost create astfel de exemple, sunt expuse imagini sugestive și multiple, toate acestea cu scopul de a ușura înșușirea noțiunilor prezentate și de a face studiul captivant. În plus, fiecare algoritm discutat vine cu pseudocod detaliat și cod implementat în Python. Este posibilă testarea codului pe instanțe proprii acceptând diferite tipuri de input dat de utilizator și primirea de informații

suplimentare referitoare la pași intermediari determinați în cod pentru a putea ține pasul cu algoritmii expuși.

Pentru a realiza o aplicație accesibilă pentru cât mai mulți utilizatori, informațiile din aplicație sunt disponibile în două limbi, și anume limba română și limba engleză.

Lucrarea este formată din 7 capitole. După o prezentare generală a problemei în care sunt descrise și variații ale problemei și încadrarea în clasa problemelor NP-complete, următoarele 3 capitole sunt dedicate celor 3 tipuri de algoritmi: exacti (în care sunt examinați algoritmii bazați pe programare dinamică și pe Branch and Bound), aproximativi (algoritmul arborelui dublu și algoritmul lui Christofides) și euristică (farthest insertion, nearest insertion, cheapest insertion, nearest neighbor care determină un ciclu hamiltonian urmărind strategii greedy și algoritmii 2-OPT și 3-OPT care micșorează soluția prin modificări aduse ciclului hamiltonian determinat prin una dintre cele patru euristici enumerate pentru a obține o soluție de cost mai mic). Codul implementat în Python poate fi accesat prin intermediul platformei GitHub [27]. În *Capitolul 5* sunt comparate performanțelor algoritmilor prezenați anterior în raport cu valoarea soluție obținute și timpul de execuție pentru a determina pentru diferite tipuri de instanțe algoritmii care oferă soluții bune într-un timp rezonabil. În capitolul următor este prezentată aplicația web. Informațiile expuse în aplicație sunt transpusă din partea teoretică a lucrării într-un mod ușor de urmărit și mult mai vizual, lăsând utilizatorului posibilitatea de a testa algoritmii discuți pe diferite tipuri de instanțe. Codul aplicației poate fi urmărit pe platforma GitHub [28].

Lucrarea se încheie cu o serie de concluzii și posibilități de dezvoltare viitoare a aplicației.

1. Problema comis-voiajorului

Problema comis-voiajorului (prescurtat TSP) este una dintre cele mai cunoscute probleme computaționale de optimizare, o problemă de interes datorită faptului că este întâlnită în practică sub diverse forme. Din această cauză, are o istorie lungă de încercări de a o rezolva, găsirea unui algoritm cât mai eficient chiar dacă vom vedea că este foarte posibil să nu se poată crea un algoritm care să ofere o soluție în timp polinomial, fiind demonstrat faptul că problema este NP-completă.

1.1. Definirea problemei

TSP presupune găsirea rutei optime prin care comis-voiajorul poate vizita n orașe, trecând prin fiecare o singură dată și întorcându-se la final în orașul de unde a plecat. Ruta optimă înseamnă drumul cel mai scurt, de costul minim sau timpul minim, toate fiind în fapt variante de a exprima aceeași problemă.

Fie $G = (V, E)$ un graf neorientat pentru care V este mulțimea a n noduri (orașe), E este mulțimea a m muchii între noduri (drumurilor dintre orașe) și fiecare muchie (i, j) are asociată o lungime ne-negativă numită distanță dintre x și y notată $d(i, j)$. Un graf neorientat ponderat poate fi privit ca fiind complet creând muchii fictive între nodurile neadiacente de lungime infinit.

Orice graf ponderat se poate memora cu ajutorul matricei de distanță $D = (d_{ij})_{n \times n}$ unde pentru fiecare i și j avem $d_{ij} = d(i, j) =$ lungimea muchiei (i, j) . Distanța de la un nod la sine este definită a fi zero ($d_{ii} = 0$ pentru orice $i \in V$). În algoritmii descriși în capitolele următoare, vom

preferă să considerăm $d_{ii} = \infty$ pentru a elimina nevoia de a testa că vârfurile curente sunt distințe.

Definiție 1.1.

Un ciclu hamiltonian într-un graf complet G este o permutare ciclică $(1, \pi(1), \dots, \pi^{n-1}(1))$ pe mulțimea nodurilor $\{1, \dots, n\}$ din graf: $1 \rightarrow \pi(1) \rightarrow \dots \rightarrow \pi^{n-1}(1) \rightarrow 1$.

Un ciclu hamiltonian minim este un ciclu hamiltonian π astfel încât lungime totală a ciclului

$$d(\pi) = \sum_{i=1}^n d(\{i, \pi(i)\})$$

este minimă.

Problema comis voiajorului (TSP) este următoarea: dat un graf neorientat complet, să se determine ciclul hamiltonian minim. Această formă a problemei poartă numele de problema comis-voiajorului simetrică, prescurtat sTSP, sau de TSP general.

În cazul TSP general, graful fiind neorientat, $d_{ij} = d_{ji}$ pentru orice $i, j \in V$. Prin urmare, $D = D^T$ adică matricea este egală cu transpusa sa. În analogie cu viața reală, problema poate fi văzută în felul următor: nodurile sunt destinații, iar muchiile sunt străzi cu specificația că nu există străzi cu sens unic (distanța este egală indiferent de direcția de mers). Se dorește drumul cel mai scurt.

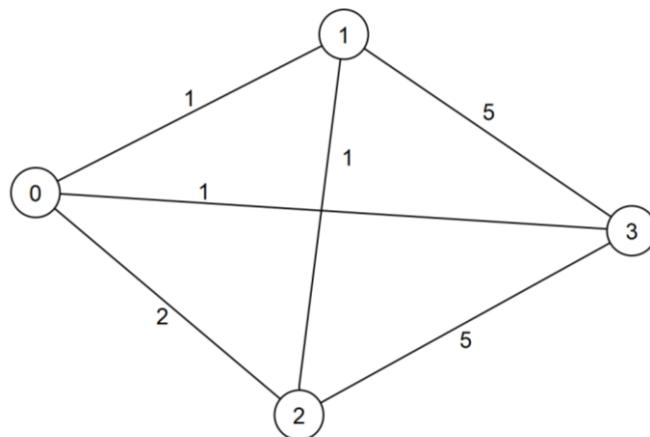


Figura 1.1. Graf neorientat cu muchii ponderate, corespunzător unui TSP general

Matricea corespunzătoare grafului din *Figura 1.1.* anterioară este $D = \begin{pmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 1 & 5 \\ 2 & 1 & 0 & 5 \\ 1 & 5 & 5 & 0 \end{pmatrix}$

și soluția optimă oferită de sTSP este lungimea 8 obținută de ciclul $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$.

1.2. Variații [3, 8, 9, 10, 11]

TSP este o problemă des studiată și aprofundată, motiv pentru care de-a lungul timpului a fost menționată sub diferite forme, variații a problemei de bază sTSP.

1.2.1. TSP asimetric – aTSP

În oglindă cu sTSP, există problema comis-voiajorului asimetrică (prescurtat aTSP). În acest caz sunt două direcții de mers, drumul de la i la j și cel de la j la i poate să difere ca lungime. Cu alte cuvinte, graful asociat este orientat, deci matricea D nu este neapărat simetrică. aTSP conține sTSP ca și caz special pentru că o muchie (i, j) din sTSP este văzută ca două muchii (i, j) și (j, i) în aTSP. Interesant este că și aTSP-ul poate fi transformat în varianta generală, caz discutat mai jos. Acest fapt este important dacă se dorește rezolvarea unui aTSP folosind un algoritm proiectat doar pentru sTSP.

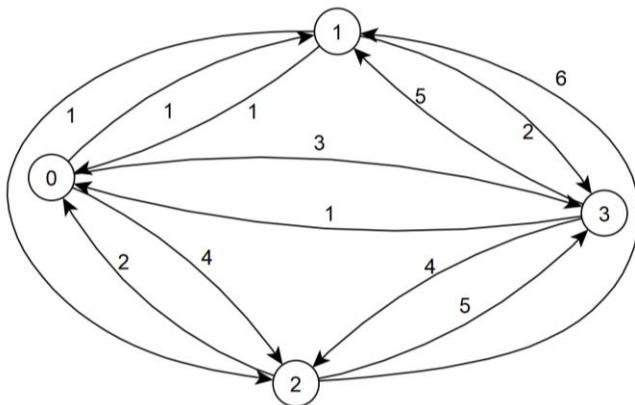


Figura 1.2. Graf orientat cu muchii ponderate. Graful este atribuit unui TSP asimetric.

Matricea corespunzătoare grafului din *Figura 1.2.* anterioară: $D = \begin{pmatrix} 0 & 1 & 4 & 1 \\ 1 & 0 & 6 & 2 \\ 2 & 1 & 0 & 5 \\ 3 & 5 & 4 & 0 \end{pmatrix}$

Teoremă 1.1. [10]

Un TSP asimetric cu n orașe poate fi transformat într-un TSP simetric cu $2n$ orașe.

Demonstrație:

Transformarea se face prin următoarele etape:

- Se creează matricea $\bar{D} = (\bar{d}_{ij})_{n \times n}$ care este identică cu matricea $D = (d_{ij})_{n \times n}$ a sTSP-ului exceptie făcând pe pozițiile \bar{d}_{ii} care au valoarea $-M$ în loc de 0, unde M este un număr foarte mare.

Se setează M cu 999 și se construiește \bar{D} folosind matricea D corespunzătoare grafului dat în *Figura 1.2.:*

$$\bar{D} = \begin{pmatrix} -999 & 1 & 4 & 1 \\ 1 & -999 & 6 & 2 \\ 2 & 1 & -999 & 5 \\ 3 & 5 & 4 & -999 \end{pmatrix}$$

- Se creează matricea $U = (u_{ij})$ unde $u_{ij} = \infty$ pentru orice $i, j \in \{1, \dots, n\}$.
- aTSP-ului cu matricea D îi corespunde un sTSP cu matricea $\tilde{D} = (\tilde{d}_{ij})_{2n \times 2n}$ definită astfel:

$$\tilde{D} = \begin{pmatrix} U & (\bar{D})^T \\ \bar{D} & U \end{pmatrix}$$

unde $(\bar{D})^T$ este matricea transpusă a lui \bar{D} .

Continuând exemplul, se pleacă de la \bar{D} pentru a se construi \tilde{D} :

$$\tilde{D} = \begin{pmatrix} & & & -999 & 1 & 2 & 3 \\ & \infty & & 1 & -999 & 1 & 5 \\ & & & 4 & 6 & -999 & 4 \\ & & & 1 & 2 & 5 & -999 \\ -999 & 1 & 4 & 1 & & & \\ 1 & -999 & 6 & 2 & & & \\ 2 & 1 & -999 & 5 & & & \\ 3 & 5 & 4 & -999 & & & \infty \end{pmatrix}$$

Graful corespunzător matricei \tilde{D} este reprezentat în Figura 1.3., ţinând cont că între nodurile între care nu există muchie este de fapt valoarea ∞ :

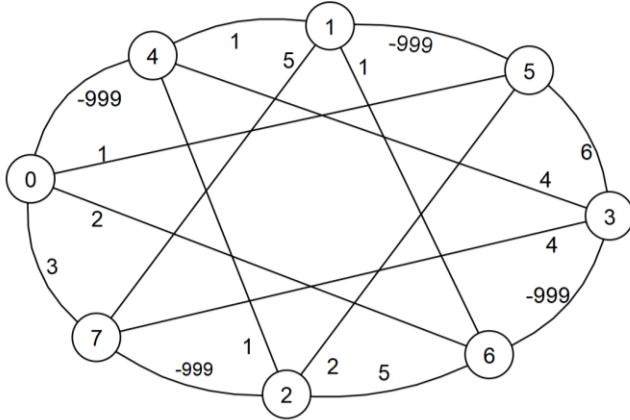


Figura 1.3. Transformarea din graful orientat prezentat în Figura 1.2.
în graful unui sTSP

4. Soluția sTSP corespunzătoare matricei \tilde{D} trebuie să treacă prin toate nodurile, aceasta însemnând că trebuie să conțină n muchii cu costul $-M$. Dublându-se numărul de noduri, orice nod din D conține două noduri în \tilde{D} , unul de intrare și unul de ieșire, aşadar o soluție este de formă:

$$i_1 \rightarrow (i_1 + n) \rightarrow i_2 \rightarrow (i_2 + n) \rightarrow \dots \rightarrow i_n \rightarrow (i_n + n) \rightarrow i_1$$

Relația dintre cele două matrice se bazează pe necesitatea eliminării celor n muchiilor adăugate:

$$\text{soluție(aTSP)} = \text{soluție(sTSP)} + nM$$

pentru care se știe că TSP-ul asimetric este reprezentat prin matricea D și TSP-ul simetric matricea prin \tilde{D} creată.

De exemplu, pentru aTSP-ul soluția optimă este 8, obținută prin ciclul $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$ (Figura 1.2.). Iar soluția în urma transformării în sTSP este determinată de ruta $0 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 2 \rightarrow 6 \rightarrow 3 \rightarrow 7 \rightarrow 0$ (Figura 1.3.) și are valoarea $-3,988$.

1.2.2. Metric TSP [3, 11]

Problema comis-voiajorului în cazul metric pleacă de la TSP-ul general pentru care matricea este simetrică, ponderile sunt nenegative și distanța de la un nod la el însuși este 0 – la acestea adăugându-se necesitatea satisfacerii inegalității triunghiului:

$$d_{ij} \leq d_{ik} + d_{kj} \quad \forall i, j, k \in V, i \neq j, i \neq k, j \neq k$$

Spre exemplu, graful din *Figura 1.1.* nu este o instanță pentru TSP metric, nerespectându-se inegalitatea triunghiului: $d_{13} > d_{10} + d_{03}$.

TSP metric este o variație importantă întrucât se întâlnește des în practică și spre deosebire de cazul general există algoritmi de aproximare cu factor constant care să rezolve problema, cel mai cunoscut fiind algoritmul lui Christofides care va fi discutat în Capitolul 3.

TSP euclidian

Un caz special de TSP metric este TSP în formă euclidiană în care distanța dintre două orașe este distanța euclidiană corespunzătoare punctelor în plan. De exemplu, în cazul în care numărul dimensiunilor în spațiul euclidian este doi echivalent cu faptul că pentru orice punct există coordonatele x și y , formula de a afla distanța dintre orașele A și B este:

$$d(A, B) = d(B, A) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}.$$

Distanța euclidiană respectă inegalitatea triunghiului.

1.2.3. Alte variații

TSP cu vizitări multiple – mTSP [8, 9]

Plecând de la sTSP, TSP cu vizitări multiple permite vizitarea același nod de mai multe ori în cazul în care acest fapt duce la determinarea unei soluții mai eficiente decât soluția oferită de sTSP.

Revenind la *Figura 1.1.*, se observă că soluția ar fi minimă dacă se permite trecerea de două ori prin nodul 0. Drumul parcurs este $0 \rightarrow 1 \rightarrow 2 \rightarrow 0 \rightarrow 3 \rightarrow 0$, iar soluția este lungimea 6.

Max TSP [8, 9]

Max TSP de diferențiază de TSP-ul general prin faptul că are ca scop găsirea drumului de lungime maximă din graful G . Acest lucru este ușor calculabil prin înlocuirea ponderilor cu inversele lor. Dar având în vedere că TSP nu acceptă muchii negative, se adună la fiecare valoare o constantă stabilită c pentru a aduce ponderile la valori pozitive și, în același timp, să nu modifice soluția obținută. Soluția dorită se calculează din soluția TSP fiind de fapt inversa diferenței dintre soluția TSP și $c * n$, unde n reprezintă numărul de noduri din graful G .

În Figura 1.4. este prezentată rezolvarea Max TSP pe graful de la Figura 1.1.:

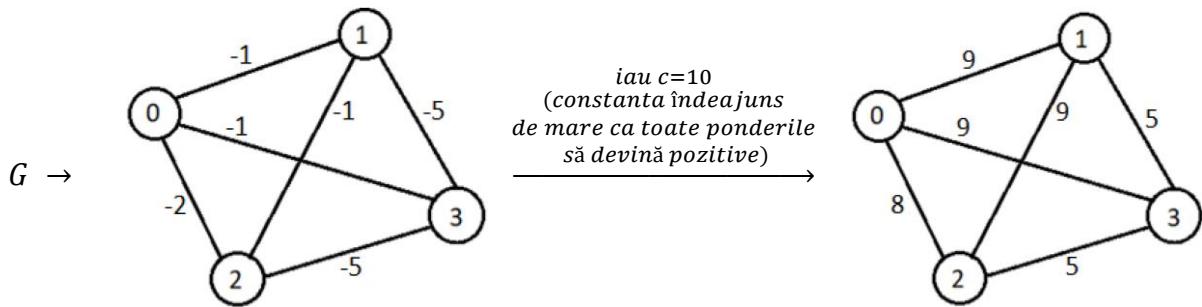


Figura 1.4. Pașii de transformare corespunzători problemei Max TSP plecând de la grafului prezentat în Figura 1.1.

$$TSP(G) = 27$$

$$\text{MaxTSP}(G) = -(TSP(G) - c * n) = 13$$

The m -salesman TSP [8]

În variațiile discutate până acum se cerea întotdeauna un singur ciclu ce cuprindea toate orașele, dar în cazul problemei m -salesmen TSP nu mai este un singur comis-voiajor, ci m . Toți cei m pleacă din același punct de start (considerat nodul de start 1). Fiecare parcurge o submulțime X_i din mulțimea de noduri $V - \{1\}$ (nodul de start este comun tuturor), având grija să nu meargă într-un oraș deja vizitat de unul dintre ceilalți comis voiajori și la final se întoarce în punctul de unde a plecat.

Problema presupune respectarea unui sir de reguli:

- I. $|X_i| \geq 1$ (fiecare comis-voiajor trebuie să parcurgă un drum)
- II. $\bigcup_{i=1}^n X_i = V - \{1\}$ (la fel ca în cazul sTSP, toate nodurile trebuie vizitate)

- III. $X_i \cap X_j = \emptyset, \forall i, j \in \{1, \dots, m\}, i \neq j$ (la fel ca în cazul sTSP, orașele trebuie vizitate o singură dată)
- IV. Lungimea ciclurilor parcuse de cei m comis-voiajori trebuie să fie minimă

TSP cu profit [9]

Varianta TSP cu profit este baza a mai multor probleme practice, precum problema cumpărătorului ambulant, job scheduling și problema rutării vehiculelor. Problema este specială prin faptul că fiecărui oraș îi este asociat un profit pe care voiajorul îl câștigă dacă alege să treacă prin acel oraș. Desigur aceasta înseamnă că nu mai este necesară vizitarea tuturor orașelor. Voiatorul alege în care să ajungă, având ca scop găsirea unei rute pentru care traseul să aibă lungime minimă și profitul să fie maxim.

Problema cumpărătorului ambulant (în engleză “Traveling Purchaser Problem”, TPP) se încadrează în variația TSP cu profit fiind deci o formă generalizată a TSP-ului. Ca date de intrare se dau o mulțime de produse și o mulțime de magazine. Fiecarui magazin îi este atribuită o cantitate pentru fiecare produs existent în mulțimea de produse. În adaos, se cunoaște de la început cantitățile dorite de cumpărătorul ambulant.

Având aceste date, se cere determinarea unei submulțimi de magazine pentru care cumpărătorul ambulant poate cumpăra cantitățile dorite din fiecare produs, minimizând costul traseului și costul cumpărării.

Așadar, se notează:

M = mulțimea magazinelor

P = mulțimea produselor

$C = (c_{ij})$ matricea în care se reține costul drumurilor; c_{ij} este costul drumului de la i la j

$D = (d_{ij})$ matricea în care se reține costul produselor la fiecare magazin; d_{ij} este costul produsului i la magazinul j . Se știe că pentru fiecare p_k se cere d_k bucăți și că la fiecare magazin se găsesc doar q_k bucăți. Dar cumpărătorul sigur poate achiziționa cantitatea dorită: $\sum q_k \geq d_k$.

Obiectivul problemei este determinarea drumului pentru care suma dintre costul transportului și costul cumpărărilor este minimă.

1.3. NP-completitudine [11]

Definiție 1.2.

O problemă se numește problemă de decizie dacă necesită un răspuns logic, de tip true/false.

Definiție 1.3.

O problemă face parte din clasa de probleme P dacă există un algoritm care oferă o soluție în timp polinomial. Clasa NP este clasa de probleme care, în cazul în care se determină o soluție, se poate verifica în timp polinomial dacă soluția este una corectă.

Teoremă 1.2.[11]

TSP face parte din clasa de probleme NP.

Demonstrație:

TSP este o problemă de optimizare (dintre toate ciclurile hamiltoniane posibile între n orașe, se dorește ciclul hamiltonian de lungime minimă), dar poate fi redusă la o problemă de decizie pentru care, dându-se un parametru M cu $M \in R^+$, se dorește să se decidă dacă se poate determina o permutare a nodurilor astfel încât $d(\pi) \leq M$. Problema în forma ei normală este cel puțin la fel de grea ca problema de decizie care îi este asociată.

Mai departe, pe lângă răspunsul true sau false, se cere să se ofere și un ciclu hamiltonian, pentru care se poate verifica în $O(n)$ dacă drumul determinat de noduri are lungime mai mică sau egală cu M . Așadar, problema de decizie oferă și o soluție pentru care poate fi verificat în timp polinomial corectitudinea răspunsului dat (dacă $d(\pi) \leq M$). Prin urmare, în analogie cu *Definiția 1.3.*, TSP în forma de problemă de decizie este NP, deci și TSP este cel puțin NP (conform *Definiției 1.2.*).

Definiție 1.4.

O problemă se numește NP-completă dacă face parte din clasa NP și orice altă problemă din NP se reduce la ea.

Teoremă 1.3. [11]

TSP face parte din clasa de probleme NP-complete.

Demonstrație:

În continuare considerăm problema CH a determinării unui ciclu hamiltonian care se știe că este NP-completă și se demonstrează pornind de la NP-completitudinea problemei comis voiajorului prin a aplica *Definiția 1.4*.

Am demonstrat în *Teorema 1.2*. că problema TSP aparține clasei NP. Presupunem că există un algoritm polinomial care rezolvă TSP.

Fie $G = (V, E)$ un graf neorientat.

Asociem lui G un graf complet G' cu ponderile $d_{ij} = \begin{cases} 1 & (i, j) \in E \\ 2 & (i, j) \notin E \end{cases}$.

Fie π soluția TSP pentru graful ponderat obținut.

Există în G un ciclu hamiltonian dacă și numai dacă în graful G' $d(\pi) = n$ deoarece, din definiția TSP-ului, π este permutarea ciclică de lungime minimă.

Într-adevăr, dacă $d(\pi) = n$, ciclul π nu conține nicio muchie de pondere 2, deci π se găsește și în graful inițial G ($G \subseteq G'$) și este soluție și pentru CH în G .

Reciproc, dacă G are ciclul hamiltonian π acest ciclu ar avea cost n în G' și ar fi soluție pentru TSP deoarece nu conține muchii de cost 2.

Așadar problema ciclului hamiltonian s-a redus la TSP, existența unei soluții de cost n pentru TSP fiind echivalentă cu existența unui ciclu hamiltonian în graf. Cum problema determinării unui ciclu hamiltonian este NP-completă nu există algoritm polinomial care să o poată rezolva, deci se ajunge la o contradicție. Presupunerea fiind falsă, se concluzionează că TSP este NP-completă.

Încadrarea problemei comis-voiajorului în clasa NP-completă înseamnă că nu se cunoaște un algoritm care să ofere o soluție în timp polinomial și găsirea unui astfel de algoritm este cel mai probabil imposibilă (nu se știe exact deoarece nu s-a putut demonstra $P = NP$, dar nici nu se exclude cu certitudine această afirmație).

Metoda prin forță brută poate fi utilizată doar pentru grafuri cu puține noduri (maxim 12-13) întrucât timpul de rulare este $O(n!)$, iar pentru un n nu foarte mare (exemplu pentru 20 de orașe) acestă căutare naivă devine imposibilă. Așadar, de-a lungul timpului, s-a încercat crearea unor algoritmi care să fie mai rapizi decât $n!$.

2. Algoritmi exacti

Algoritmii exacti sunt algoritmi care, după cum le zice numele, oferă soluții cu certitudine optime. În mod general, aceștia rulează în timp exponențial, dar se poate ajunge la timp polinomial pentru anumite instanțe speciale.

2.1. Programarea dinamică [3, 19]

Programarea dinamică este una din căile mai rapide de a rezolva TSP comparativ cu backtracking care are timp factorial. Algoritmul care folosește programare dinamică pentru TSP se numește Bellman-Help-Karp și oferă un răspuns în timpul de rulare $O(n^2 * 2^n)$. Funcționează pentru un n care poate ajunge până la 30. Numărul de orașe este încă mic, dar fiind o problemă NP-completă, este dificil de realizat un algoritm până și pentru numere relativ mici.

Programarea dinamică presupune rezolvarea problemei folosind subprograme ce returnează soluții parțiale care sunt folosite în mod recursiv cu scopul de a ajunge la soluția întregii probleme. Cu alte cuvinte, o problemă mai mare A este rezolvată folosind rezultatul unei probleme mai mici B (numită subproblemă), deci problema B trebuie să fie știută înainte de a rezolva problema A - de aici ideea de recursivitate, mergându-se de la mic la mare. Pentru a nu rezolva o problemă mai mică de mai multe ori, se memorează rezultatele parțiale și se reutilizează.

Pentru TSP o subproblemă corespunde părții de început a drumului. Se consideră că se pornește parcurgerea din orașul 1 și se ajunge în orașul j după ce au fost vizitate câteva orașe. La

fiecare pas, pentru a extinde începutul de drum, este necesar să se cunoască orașul j pentru că se dorește găsirea orașelor care de vizitat după ce s-a trecut prin j . Pentru a ști că se respectă regula TSP de a vizita fiecare oraș o singură dată, trebuie să se cunoască și identitatea nodurilor vizitate în drumul parțial pentru a nu le parurge din nou, de aceea o subproblemă corespunde unei perechi de forma (*submulțime de noduri, început de drum*).

Gândindu-ne la ultimul pas, soluția optimă de obține închizând ciclului prin aducerea nodului j înapoi la nodul de start 1. La acest pas se știu toate drumurile minime care pornesc din 1 și ajung în orice $j \in \{2, \dots, n\}$ trecând prin toate nodurile, deci există $n - 1$ candidați la soluția optimă a problemei. Soluția problemei va fi obținută astfel:

$$\text{soluția optimă} = \min \left(\begin{array}{l} \text{drumul cel mai scurt} \\ \text{de la 1 la } j \text{ conținând} \\ \text{toate nodurile din V} \end{array} \right) + d_{j1}$$

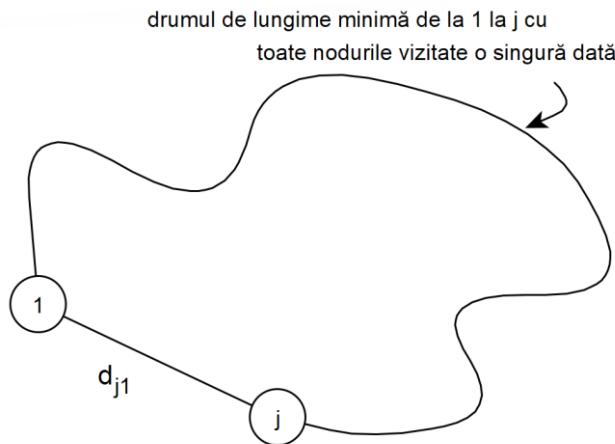


Figura 2.1. Reprezentarea ultimului pas din algoritm folosind programare dinamică. Știindu-se toate drumurile care conțin n noduri, se caută acela la care dacă de adaugă distanța până la nodul de start, se creează ciclul de lungime minimă

Relațiile de recurrentă rezultă din următorul principiu de optim: un drum optim P care pornește din 1 și se termină în j și trece prin nodurile din mulțimea S se obține dintr-un drum optim $P' \subseteq P, P' = P - \{j\}$ care începe în 1 și se termină în i și trece prin nodurile din $S - \{j\}$ adăugând muchia (i, j) (care se poate demonstra ușor prin reducere la absurd).

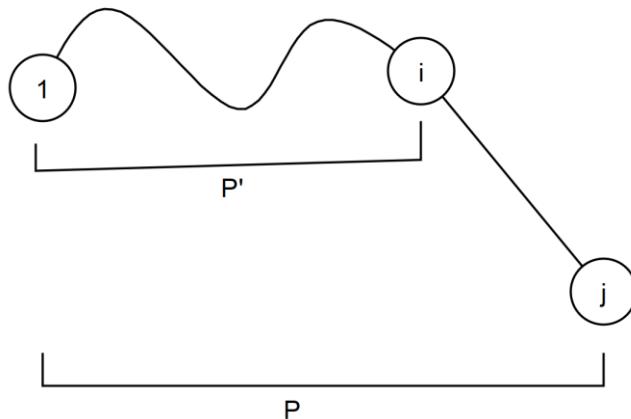


Figura 2.2. Reprezentarea cazului general al recursivității.

Se creează drumul P optim din drumul P' optim

În continuare formalizăm aceste informații și descriem algoritmul bazat pe programare dinamică.

Recursivitate

Notăție: Pentru o submulțime $S \subseteq \{1, \dots, n\}$ care conține 1 și j , fie subproblema $M(S, j) =$ lungimea celui mai scurt drum P de la nodul 1 la j vizitând nodurile din S o singură dată.

Recursivitatea prezintă două părți, cazul de oprire și relația recursivă. Cazul de oprire se referă la subproblemele care se pot rezolva direct. Având în vedere că se lucrează cu mulțimi, ne oprim când în P nu mai există un nod i intermediar astfel încât să se formeze drumul P' . Așadar dacă submulțimea S corespunzătoare drumului P este de forma $\{1, j\}$, valorile lui M sunt ponderile muchiilor din matricea de distanțe D :

$$M(S, j) = d_{1j}$$

La orice pas intermediar $M(S, j)$, problema se rezumă la stabilirea penultimul nod $i \in S$ astfel încât drumul de la 1 la i notat cu $M(S - \{j\}, i)$ plus lungimea ultimei muchii d_{ij} este minim:

$$M(S, j) = \min_{i \in S; i \neq 1, j} (M(S - \{j\}, i) + d_{ij})$$

Iar legat de ultimul pas, se știe nodul unde se dorește să se ajungă și toate cele $n - 1$ posibilități de drumuri:

$$M(V, 1) = \min_{j \in V - 1} (M(V, j) + d_{j1})$$

Algoritm

Algoritmul ce rezolvă TSP folosind programare dinamică se bazează pe modul de gândire menționat mai sus și este descris în pseudocod astfel:

1. $D \leftarrow$ matrice $n \times n$ de distanțe între oricare 2 orașe
2. $M \leftarrow$ matrice $2^n \times n$ inițializată cu ∞
3. pentru $j \leftarrow 2, n$ execută
 4. $M(\{1, j\}, j) \leftarrow d_{1j}$
 5. pentru $s \leftarrow 3, n$ execută
 6. pentru toate submulțimile $S \subseteq \{1, \dots, n\}$ cu $I \in S$ și $|S| = s$ execută
 7. pentru orice $j \in S, j \neq 1$ execută

$$M(S, j) = \min_{i \in S; i \neq 1, j} (M(S - \{j\}, i) + d_{ij})$$
 9. returnează $\min_j (M(\{1, \dots, n\}, j) + d_{j1})$

Urmărind codul scris în pseudocod, se observă la liniile 4 și 5 cazul de oprire, la linia 8 relația recursivă și la 9 ultimul pas, acela al găsirii soluției optime știind deja toate drumurile minime care conțin n noduri.

În implementarea algoritmului, submulțimile S se pot stoca prin scriere în binar a nodurilor ce le compun astfel: bitul egal cu 1 înseamnă că nodul de pe poziția respectivă există în submulțime și 0 că nu există. De exemplu, submulțimii $\{0, 1, 2\}$ îi corespunde în binar 0111 care reprezintă numărul 7. Prin urmare, în matricea M este plasat pe linia 7 a matricei. Verificarea dacă nodul j este inclus în submulțimea S se realizează prin operația de înmulțire bit cu bit ($\&$) între cele două mulțimi (S și mulțimea ce îl conține doar pe j determinată prin șiftarea la stânga a lui j : $1 << j$). Pentru $j = 2$, aceasta înseamnă: $7 \& (1 << 2) = 0111 \& 0100 = 0100$. Rezultatul este diferit de 0, aşadar j aparține mulțimii.

Pentru a determina drumul $S' = S - \{j\}$ se face XOR între S și $\{j\}$: $S \wedge (1 << j) = 0111 \wedge 0100 = 0011$. Rezultă că S' este 3.

Codul corespunzător algoritmului Bellman-Help-Karp este inclus în [27].

Exemplu

Având matricea D :

	0	1	2	3
0	inf	220022	960850	599952
1	inf	inf	316670	578985
2	781797	62190	inf	121118
3	inf	540561	585978	inf

se ajunge în urma algoritmului la matricea M :

bitul corespunzător nodului 0		0	1	2	3
0000	0	inf	inf	inf	inf
0001	1	inf	inf	inf	inf
0010	2	inf	inf	inf	inf
0011	3	inf	220022	inf	inf
0100	4	inf	inf	inf	inf
0101	5	inf	inf	960850	inf
0110	6	inf	inf	inf	inf
0111	7	inf	1023040	536692	inf
1000	8	inf	inf	inf	inf
1001	9	inf	inf	inf	599952
1010	10	inf	inf	inf	inf
1011	11	inf	1140513	inf	799007
1100	12	inf	inf	inf	inf
1101	13	inf	inf	1185930	1081968
1110	14	inf	inf	inf	inf
1111	15	inf	1248120	1384985	65781

scrierea în binar a lui S

și la soluția optimă 2166782.

Complexitate

Teoremă 2.1. [19]

Algoritmul Bellman-Help-Karp are timpul de rulare $O(n^2 * 2^n)$.

Demonstrație:

Chiar dacă se ține evidența tuturor submulțimilor S , nu contează ordinea vizitării nodurilor din S încrucât doar identificarea acestora este importantă pentru a nu parcurge un nod de mai multe ori. Acest aspect face diferența timpului de rulare dintre varianta cu programare dinamică și cea cu forță brută: în cazul în care conta ordinea dintre nodul sursă și nodul

destinație se ajungea la $n!$ subprograme pe când acum sunt 2^n posibilități de a alege S . Sunt n posibilități pentru destinația j , nodul care se întoarce în 1, și n variante de a alege nodul i (penultimul din drum), $i \in S$. Așadar, timpul de rulare rezultat este $O(n^2 * 2^n)$.

2.2. Branch and Bound [14]

Metoda Branch and Bound se aplică problemelor care pot fi rezolvate prin backtracking, deci reprezentate printr-un arbore. Comparativ cu backtrackingul, arborele metodei Branch and Bound este construit dinamic fiind eliminați subarborei care depășesc o limită superioară a soluției optime. Această limită poate fi lungimea primului ciclu găsit care, este sigur mai mare sau egală cu soluția optimă. Deci, în cazul în care o soluție parțială ar depăși-o, nu are rost să se continue drumul respectiv. Astfel nu se parurge tot arborele, micșorând complexitatea problemei, iar scopul este găsirea aceluia vârf rezultat (frunză) care oferă soluția optimă din totalul de vârfuri rezultat.

Nodurile arborelui corespund stărilor posibile în dezvoltarea soluției, stări ce corespund traseului de la rădăcină până la nodul respectiv. Ideea aceasta este întâlnită și la arborele metodei backtracking.

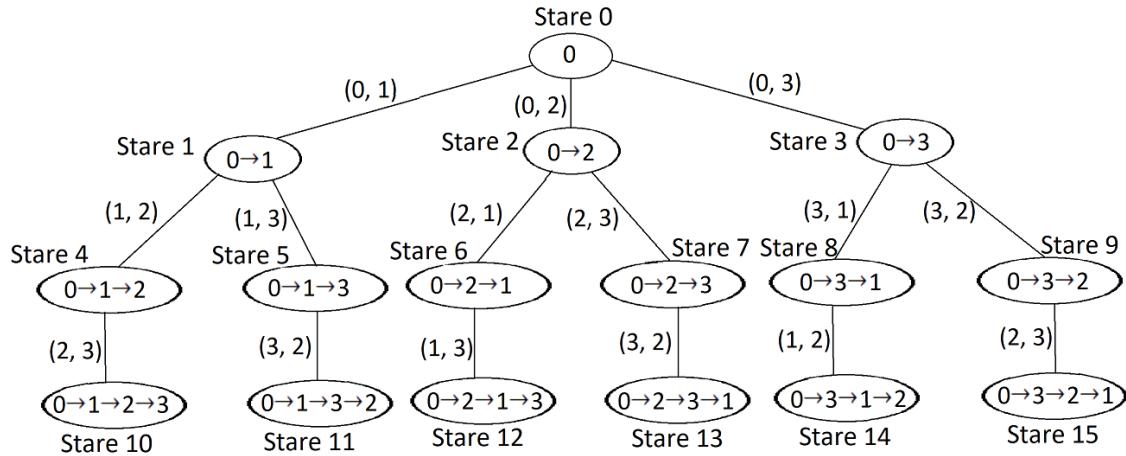


Figura 2.3. Arborele corespunzător grafului reprezentat din Figura 1.2. Nodurile arborelui conțin drumul parcurs de la rădăcină și au asociate numărul stării.

Lim reprezintă o aproximare prin adăos a minimului căutat fiind folosită pentru a elimina din L nodurile cu cost mai mare decât lim , adică nodurile care prin extindere nu

se obține soluția optimă. Prin eliminarea unui nod, se elibera tot subarborele pentru care nodul respectiv este rădăcină. L_{im} se actualizează pe parcursul algoritmului, luând valoarea oricărui ciclu cu valoare mai mică decât a sa. Prin urmare, la final reprezintă valoarea soluției optime, iar inițial, neștiind dacă instanța are soluție, este inițializat cu ∞ sau cu o limită teoretică cunoscută.

Se disting mai multe variante de a rezolva problema în funcția de ordinea în care sunt explorate nodurile nevizitate.

O prima varianta combină ideea de la parcurgerea în lățime cu o strategie best-first, alegându-se la fiecare pas din lista de noduri active L nodul cu costul minim pentru a nu extinde toate nodurile de pe fiecare nivel. Pseudocodul pentru determinarea primului nod final este următorul:

Fie L lista de vârfuri active (vârfuri accesibile), inițial conținând doar vârful de start 0.

repetă

- se alege un vârf din L al cărui cost este minim în raport cu costurile celorlalte noduri din L . Vârful ales devine cel curent
 - se generează toți fiile vârfului curent și se adaugă la L
- până când vârful curent este final

Aceasta nu este o metodă potrivită dacă nodurile ajung să fie parcurse pe niveluri. În acest caz, până să se ajungă la o frunză care să seteze limita maximă a soluției, nu vor fi eliberați subarborei. Așadar, algoritmul devine foarte asemănător cu cel de backtracking care are complexitatea $n!$. Spre exemplu, în cazul TSP euclidian, strategia propusă se comportă ca o parcurgere în lățime, vizitând multe noduri până ajunge la un nod final. În acest caz, se pot folosi alte strategii.

O altă strategie presupune parcurgerea în adâncime (DFS, depth-first search). Pseudocodul pentru determinarea primului nod final este următorul:

Se alege un nod vecin nevizitat cu nodul de start, se notează ca nod curent și i se calculează costul.

repetă

- se alege un nod nevizitat care este vecin cu nodul curent și i se calculează costul
- până când nodul curent este final

Cu ajutorul acestei metode se ajunge repede la o soluție, motiv pentru care se poate începe eliminarea subarborilor. Problema este că soluția calculată poate să nu fie restrictivă pentru că nu se pune nicio condiție când se alege nodul următor. Deci chiar dacă se ajunge la o soluție, nu se fac eliminări excesive.

Pornind de la problemele întâlnite pentru TSP euclidian folosind strategiile anterioare, este nevoie de o cale de mijloc care oferă o constrângere, dar priorizează și ajungerea la o soluție în pași relativ puțini. O soluție intermedieră este ca pentru nodul curent să se genereze toți fiile și să li se calculeze costul. Nodul ales ca nod următor este cel cu costul minim (asemănarea cu BFS). La următorul pas din nou se generează toții fiile, dar alegerea noului nod nu se face din mulțimea tuturor nodurilor întâlnite până la pasul curent, ci doar din fiile direcții ai nodului curent. În acest mod, nu se generează doar n fiile pentru a ajunge la o soluție (ca în cazul DFS-ului), ci $\frac{n(n-1)}{2}$. În schimb se câștigă când vine vorba de valoarea soluției găsite pentru că la fiecare pas se alege minimul.

Soluția stabilită se descrie astfel:

Fie nodul curent inițializat cu nodul de start

repetă

- se generează toți fiile nodului curent
- se alege ca nod curent acela cu costul minim dintre fiile

până când nodul curent este final

Când se ajunge la o frunză j care are ca tată pe i , se face un pas în spate în i și se alege ca nod curent fiul nevizitat al lui i cu costul minim. Pseudocodul complet pentru această strategie va fi descrisă în continuare.

Algoritm

Fie $f(i)$ limita inferioară pentru ciclul minim corespunzător nodului i , nod_start rădăcina, nodul de început, și $drum_optim$ variabila în care se reține drumul de la rădăcină până la frunză corespunzător soluției minime găsite.

Având în vedere ca parcurgerea merge din nodul curent în fiu, apoi în fiul fiului și aşa mai departe, se propune rezolvarea problemei folosind o metoda recursivă.

Pseudocod parcurgere(i , $drum$)

1. $distanța_minimă \leftarrow \infty$
2. $nod_următor \leftarrow -1$
3. dacă $f(i) < lim$ atunci
 - 4. dacă lista $drum$ conține toate nodurile atunci
 - 5. $lim \leftarrow f(i)$
 - 6. $drum_optim \leftarrow drum$
 - 7. altfel
 - 8. cât timp i mai are fișă j execută
 - 9. $nod_următor \leftarrow$ nodul j cu $f(j)$ minim
 - 10. $drum_următor \leftarrow drum$ la care se adaugă $nod_următor$
 - 11. $parcurgere(nod_următor, drum)$

În plus, este nevoie de o metodă care să inițializeze valorile și să apeleze funcția recursivă.

Pseudocod Branch_and_bound

1. setează nodul de start
2. $lim \leftarrow \infty$
3. $drum_optim \leftarrow$ lista vidă
4. $parcurgere(nod_start)$
5. dacă $lim = \infty$ atunci
 - 6. scrie ‘Nu există soluție’
7. altfel
 - 8. returnează $lim, drum_optim$

Partea nediscutată este legată de calculul funcției $f(i)$ astfel încât să reprezinte limita inferioară a unui ciclu. Există mai multe modalități de a defini funcția, cea mai cunoscută bazându-se pe reducerea matricei de distanțe astfel pornind de la următoarea observație. Dacă se micșorează toate elementele de pe o linie i sau coloană j din D cu o valoare α , orice ciclu hamiltonian va avea costul micșorat cu α pentru că se intră o singură dată în nodul j și se pleacă o singură dată din nodul i . Se menționează ca distanța de la un nod la el însuși nu mai este 0, ci infinit pentru a putea exista un α diferit de 0.

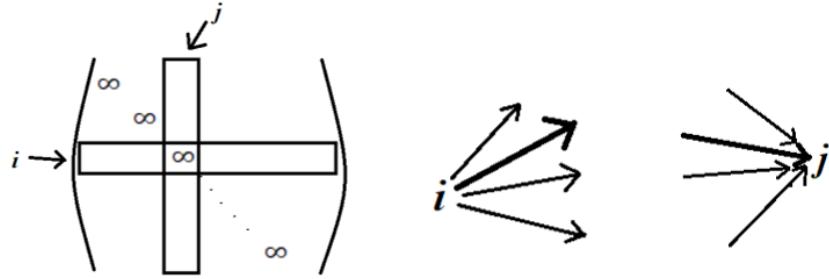


Figura 2.4. Reprezentare a faptului că într-un ciclu se intră o singură dată în nodul i și seiese o singură dată din nodul j .

Așadar apare noțiunea de matrice de distanțe redusă, matricea obținută prin micșorări repetitive până când pe orice linie și coloană apare cel puțin un 0, exceptie făcând cazul în care linia/coloana conține numai ∞ .

Revenind la limita inferioară f , $f(nod_start)$ reprezintă valoarea cu care se reduce matricea inițială D . Unui nod oarecare j îi este asociată matricea de distanțe redusă anterior de părintele său. Presupunând că j îl are ca tată pe i , se reduce D_j astfel: elementele liniei i devin ∞ (numai o singură dată seiese din i), elementele coloanei j devin ∞ (numai o singură dată se intră în j) și $D_j(j, nod_start) = \infty$ pentru a împiedica închiderea circuitului înainte de a parurge toate nodurile. Fie α cantitatea cu care s-a redus matricea D_j , atunci $f(j) = f(i) + D_i(i, j) + \alpha$.

Soluția este $\min\{f(i)\}$ unde i este frunză.

Algoritmul scris în Python este atașat în [27].

Complexitate

La bază, metoda este asemănătoare cu varianta care folosește backtracking, prin urmare, în cazul cel mai defavorabil, ajunge să parcurgă toți subarborii, deci să necesite timp exponențial. Însă, în caz favorabil, timpul de rulare este diminuat de reducerea repetată a matricei de distanță, aceasta ajutând la estimarea bună a drumului optim, și de eliminare a subarborilor care depășesc aproximarea soluției optime.

Îmbunătățire

Putem folosi ca limită superioară o limită teoretică sau valoarea obținută de un algoritm heuristic.

Spre exemplu, pentru TSP euclidian putem să ne folosim de un algoritm heuristic precum nearest insertion sau cheapest insertion despre care vom discuta în *Capitolul 4* că soluția returnată este sigur mai mică de dublul soluției optime, chiar și 2-OPT sau 3-OPT care utilizează pentru determinarea ciclului hamiltonian inițial unul dintre cele două euristici menționate. Se ia rezultatul obținut de algoritm care este o soluție parțială a instanței cerute și, plecând de la el, să aplică algoritmul Branch and bound pentru a îmbunătății limita deja știută (răspunsul algoritmului heuristic). Prin acest mod numărul de pași al algoritmului exact în discuție va scădea pentru că nu va mai fi nevoie de o parcurgere până la o frunză pentru a începe eliminarea de subarbore. Ci de la început se vor tăia dacă este cazul, micșorând timpul de rulare al algoritmului semnificativ.

Exemplu

$$D = \begin{pmatrix} \infty & 1 & 4 & 1 \\ 1 & \infty & 6 & 2 \\ 2 & 1 & \infty & 5 \\ 3 & 5 & 4 & \infty \end{pmatrix}$$

*Matricea corespunzătoare grafului din Figura 1.2. în care valoarea
de la un nod la el însuși este ∞*

Aplicăm prima reducere pentru a determina $f(nod_start)$, deci $f(0)$:

reducem linia 0 cu 1

$$\begin{pmatrix} \infty & 0 & 3 & 0 \\ 1 & \infty & 6 & 2 \\ 2 & 1 & \infty & 5 \\ 3 & 5 & 4 & \infty \end{pmatrix}$$

reducem linia 1 cu 1

$$\begin{pmatrix} \infty & 0 & 3 & 0 \\ 0 & \infty & 5 & 1 \\ 2 & 1 & \infty & 5 \\ 3 & 5 & 4 & \infty \end{pmatrix}$$

reducem linia 2 cu 1

$$\begin{pmatrix} \infty & 0 & 3 & 0 \\ 0 & \infty & 5 & 1 \\ 1 & 0 & \infty & 4 \\ 3 & 5 & 4 & \infty \end{pmatrix}$$

reducem linia 3 cu 3

$$\begin{pmatrix} \infty & 0 & 3 & 0 \\ 0 & \infty & 5 & 1 \\ 1 & 0 & \infty & 4 \\ 0 & 2 & 1 & \infty \end{pmatrix}$$

reducem coloana 2 cu 1

$$\begin{pmatrix} \infty & 0 & 2 & 0 \\ 0 & \infty & 4 & 1 \\ 1 & 0 & \infty & 4 \\ 0 & 2 & 0 & \infty \end{pmatrix}$$

Așadar, $f(0) = I + I + I + 3 + 1 = 7$ și $D_0 = \begin{pmatrix} \infty & 0 & 2 & 0 \\ 0 & \infty & 4 & 1 \\ 1 & 0 & \infty & 4 \\ 0 & 2 & 0 & \infty \end{pmatrix}$ matricea redusă.

Următorul pas este să se calculeze f pentru toți fiile rădăcinii. În cazul acesta plecând de la nodul de start 0, se ajunge la fiile 1, 2, 3. Se va arăta cum se efectuează pentru nodul 1, restul reducerilor determinându-se analor, respectându-se regulile de reducere a matricei și pseudocodul prezentat anterior.

- elementele liniei 0 devin ∞
- elementele coloanei 1 devin ∞
- $D_1(1,0) = \infty$

matricea devine: $D_1 = \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 4 & 1 \\ 1 & \infty & \infty & 4 \\ 0 & \infty & 0 & \infty \end{pmatrix}$

reducem linia 1 cu 1

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & 0 \\ 1 & \infty & \infty & 4 \\ 0 & \infty & 0 & \infty \end{pmatrix}$$

reducem linia 2 cu 1

$$\begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & 0 \\ 0 & \infty & \infty & 3 \\ 0 & \infty & 0 & \infty \end{pmatrix}$$

$$\alpha = 1 + 1 = 2$$

$$\text{Prin urmare, } f(1) = f(0) + D_0(0, 1) + \alpha = 7 + 0 + 2 = 9.$$

La final se ajunge la arborele următor. Algoritmul se oprește când întâlnește nodul final 15 și se elimină treptat toți fiile când se revine din funcția recursivă. Soluția este dată de valoarea lui $f(15)$.

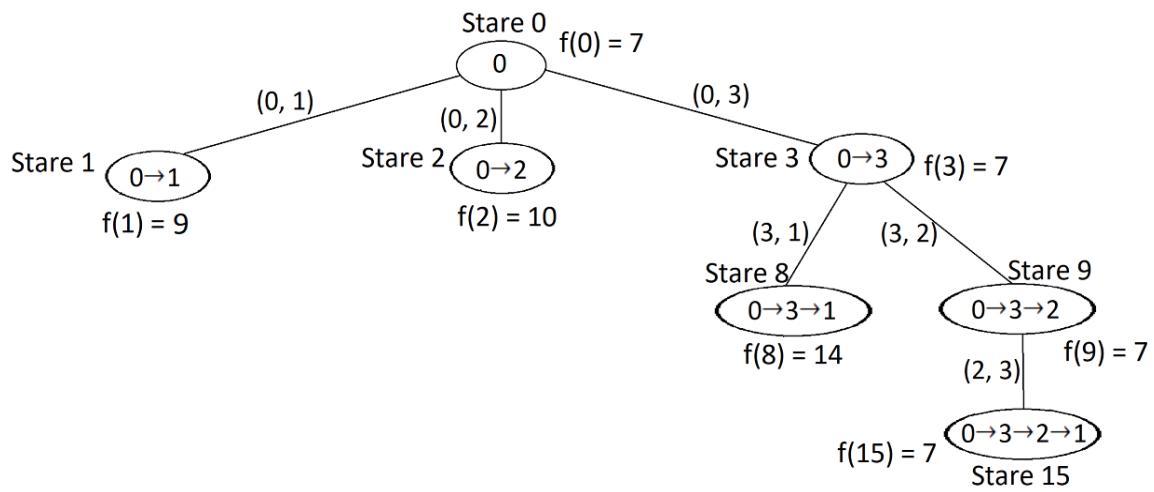


Figura 2.5 Arborele de la Figura 2.4 rezultat în urma eliminării subarborilor care au valoarea f mai mare ca valoarea lim – aproximarea prin adaos.

3. Algoritmi de aproximare [2, 13, 16, 18, 20, 24, 26]

În *Capitolul 2* s-a discutat despre algoritmi care oferă în mod sigur soluția optimă indiferent de inputul ales, acceptând însă faptul că timpul de rulare variază și că poate ajunge la timp exponențial. ,

Acum sunt aduși în discuție un alt tip de algoritmi, cei pentru care se dorește returnarea unui rezultat, pentru orice instanță, într-un anumit relativ timp scurt. Aceștia se numesc algoritmi aproximativi, necesitatea găsirii soluției optime fiind relaxată, fiind acceptabile soluții "destul de bune", apropriate de cea optimă, cu un factor de aproximare garantat.

Definiție 3.1.

Pentru o problemă de optimizare, un algoritm α -aproximativ este un algoritm polinomial care pentru toate instanțele problemei produc o soluție ALG a cărei valoare este într-un factor de α față de valoarea soluției optime OPT (ALG $\leq \alpha$ OPT pentru o problemă de minimizare și ALG $\geq \alpha$ OPT pentru o problemă de maximizare). Pentru un algoritm α -aproximativ, α reprezintă factorul de aproximare al algoritmului.

De exemplu, un algoritm 2-aproximativ pentru o problemă de minimizare este un algoritm care produce o soluție a cărui valoare este cel mult dublul soluției optime.

3.1. Factor de aproximare pentru TSP

TSP este una din cele mai studiate probleme și din privința algoritmilor de aproximare. Am demonstrat în *Subcapitolul 1.3.* că TSP este o problemă NP-completă, deci nu poate fi

rezolvată în timp polinomial. Se dorește stabilirea dacă există un algoritm α -aproximativ care să rezolve problema TSP în timp polinomial. Răspunsul este dat în *Teorema 3.1*.

Teorema 3.1.[16]

Pentru orice $\alpha > 1$, nu există un algoritm polinomial α -aproximativ care să rezolve problema comis-voiajorului în factorul de aproximare α , exceptie cazul în care $P = NP$.

Demonstrație:

Se presupune contrariul, că există un algoritm polinomial α -aproximativ care să rezolve problema comis-voiajorului în factorul de aproximare α , unde $\alpha \geq 1$. Se arată o reducere de la problema ciclului hamiltonian la TSP în forma generală.

Fie un graf $G = (V, E)$ neorientat neponderat și un alt graf G' complet care se construiește pornind de la G pe care îl completează, atribuind muchiilor ponderi. Orice muchie $(i, j) \in E$ are ponderea 1 și muchiile $(i, j) \notin E$ primesc ponderea $\alpha n + 1$. Dacă există un ciclu hamiltonian în graful G , atunci există același ciclu în G' și are lungimea n . Prin urmare, algoritmul rulat pe graful G' va returna un răspuns cel mult αn (nu conține nicio muchie de lungime $\alpha n + 1$).

Dacă în schimb nu există un ciclu hamiltonian în graful G , ciclul din G' va conține cel puțin o muchie de lungime $\alpha n + 1$. Așadar, algoritmul va întoarce un răspuns cel puțin $\alpha n + 1$.

Prin urmare, avem un algoritm polinomial care dându-i-se graful G contruiește G' și returnează *true* dacă $\leq \alpha n$ și *false* dacă $\geq \alpha n + 1$. Astfel este rezolvată problema ciclului hamiltonian care este NP-completă. Ajungând la o contradicție, cât timp nu se poate demonstra $N = NP$, TSP nu poate avea un algoritm polinomial α -aproximativ, indiferent de $\alpha > 1$.

În continuare vom considera TSP metric și vom arăta că există algoritmi de aproximare cu factor constant.

Teorema 3.2. [16]

TSP în forma metrică este NP-completă.

Demonstrație:

Se folosește aceeași idee ca la demonstrația *Teoremei 3.1*, cu observația că ponderile grafului G' respectă inegalitatea triunghiului.

Vom prezenta algoritmi aproximativi cu factor constant pentru problema TSP metric, nu și pentru problema TSP în forma sa generală pentru că în acest caz, conform *Teoremei 3.1.* nu există.

3.2. Algoritmul arborelui dublat (Double tree algorithm)

Algoritmul arborelui dublat este un algoritm aproximativ care rezolvă TSP metric a cărui nume provine din faptul că pentru rezolvarea problemei sunt dublate muchiile unui arbore parțial de cost minim pentru a se putea crea un ciclu eulerian și mai apoi ciclul hamiltonian.

Acest algoritm este 2-aproximativ, factorul de aproximare fiind justificat în *Teorema 3.5.*

Definiție 3.2.

Un graf se numește eulerian dacă conține un ciclu eulerian (ciclu în care se parcurg toate muchiile o singură dată, dar se permite repetarea nodurilor).

Conform Teoremei lui Euler, un graf neorientat fără noduri izolate are un ciclu eulerian dacă și numai dacă este conex și are toate nodurile de grad par.

Se consideră un TSP în cazul metric pentru care matricea D respectă inegalitatea triunghiului: $d_{ij} \leq d_{ik} + d_{kj}$. Notăm cu OPT soluția grafului G pentru problema TSP.

Teoremă 3.3. [2]

Fie submulțimea $S \subseteq V$. Un ciclu hamiltonian de cost minim C_S din subgraful induș de nodurile lui S respectă proprietatea $l(C_S) \leq OPT$.

Demonstratie:

Fie $S = V - \{k\}$. Un ciclu hamiltonian în graful induș de S se poate obține dintr-un ciclu hamiltonian al lui G prin scurcircuitare așa cum este vizibil în *Figura 3.1*. Respectându-se inegalitatea triunghiului, muchia (i,j) este cel mult egală cu suma celor două muchii care nu fac parte din ciclu scurcircuit. De aici rezultă că $l(C_S) \leq OPT$.

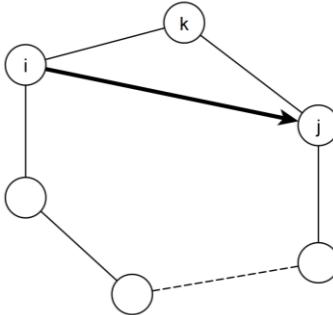


Figura 3.1. Demonstrație vizuală. Pentru un TSP metric, orice ciclu Hamiltonian creat cu o submulțime din nodurile inițial are lungimea mai mică de cât soluția optimă a grafului complet.

Pașii algoritmului arborelui dublu sunt următorii:

1. Determinarea unui arbore parțial de cost minim
2. Determinarea unui ciclu eulerian în graful obținut din arborele parțial de cost minim prin dublarea muchiilor
3. Crearea unui ciclul hamiltonian prin eliminarea din ciclul eulerian a nodurilor care se repetă

Vom detalia în continuare acești pași.

Primul pas al rezolvării problemei TSP folosind algoritmul arborelui dublat este determinarea unui arbore parțial de cost minim numit și arbore de acoperire (MST, minimum spanning tree) T . Acest poate fi obținut prin algoritmul lui Prim sau prin algoritmul lui Kruskal în timp polinomial, complexitate $O(m \log n)$.

Teoremă 3.4.

Pentru orice instanță a problemei comis-voiajorului, soluția optimă este cel puțin soluția arborelui parțial de cost minim, adică $l(T) \leq OPT$.

Demonstrație:

Dacă din ciclul hamiltonian OPT se elimină o muchie se obține un arbore parțial care are costul mai mare sau egal cu $l(T)$.

Pentru a parcurge toate muchiile din arborele de acoperire dublat în căutarea unui ciclu eulerian, se propune următoarea strategie:

Pas 1: Se ia un nod i din arbore

Pas 2:

- Dacă există o muchie de forma (i, j) nevizitată, se merge pe acea muchie, iar $i := j$. Se repetă pasul 2.

- Dacă toate muchiile care pornesc din i sunt vizitate, se dorește să se facă un pas în spate.

Dacă i este nodul ales pasul 1, i nu are unde să se întoarcă, deci stop; altfel $i := k$ unde k este nodul în care se întoarce și se repetă pasul 2.

Se ia ca exemplu arborele din *Figura 3.2.a.* și se pornește parcurgerea începând cu $i = 1$. Iar parcurgerea este următoarea: 1, 2, 3, 2, 4, 5, 6, 5, 7, 5, 8, 5, 4, 2, 1. Se ajunge la concluzia că fiecare muchie este parcursă de exact două ori, fapt ilustrat în graful din *Figura 3.2.b.*

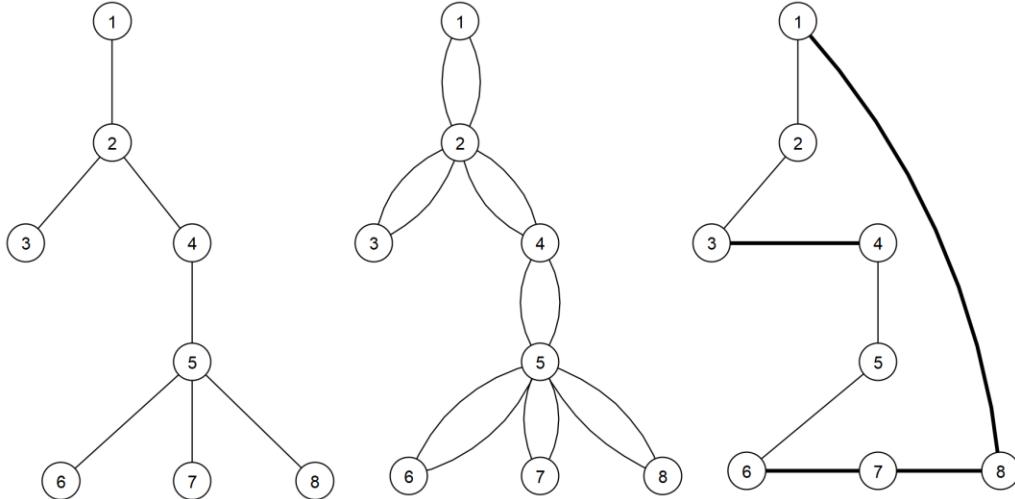


Figura 3.2. Reprezentarea unui arbore T pentru exemplificare. Figura 3.2.a. se presupune că este arborele de acoperire de cost minim. Figura 3.2.b. prezintă graful obținut din arborele pentru care fiecare muchie este dublată, iar în Figura 3.2.c. apare ciclul hamiltonian T' obținut prin shortcutting.

Chiar dacă graful este eulerian, soluția nu este un ciclu hamiltonian aşa cum se doreşte în cazul problemei TSP. Pentru a obține un ciclu hamiltonian se folosește tehnica scurtăturilor (shortcutting) prin care din parcurgerea determinată anterior (în care muchiile se repetă de două ori), se păstrează doar prima apariție a fiecărui nod, adică se sare peste nodurile deja vizitate, întorcându-ne la final la nodul de început: 1, 2, 3, 4, 5, 6, 7, 8, 1, ciclu reprezentat în *Figura 3.2.c.*

Teorema 3.5. [13]

Algoritmul arborelui dublat pentru problema comis-voiajorului pentru cazul metric este 2-aproximativ.

Demonstrație:

Având în vedere că se respectă inegalitatea triunghiului, muchiile noi obținute prin metoda shortcutting sunt sigur mai mici sau egale cu cele peste care s-a sărit. Așadar nu se va mări niciodată distanța totală a ciclului. Fie T' ciclul obținut în urma tehnicii de shortcut-uri, avem $l(T') \leq 2 l(T) \leq 2 OPT$ (din Teorema 3.4.). Pentru că $l(T') \leq 2 OPT$, acest algoritm este 2-aproximativ.

Pseudocod

Lucrând cu un TSP metric, amintim că graful asociat este neorientat, complet și se respectă inegalitatea triunghiului.

Pentru crearea arborelui parțial de cost minim se folosește algoritmul lui Prim care, plecând de la un nod de start ales în mod aleator, la fiecare pas se ține evidența nodurilor ce fac parte din arbore, a distanței minime de la fiecare nod nevizitat (care nu face parte din arbore) la arbore (nodurile care deja fac parte din arbore), aceasta din urmă pentru a putea adăuga la pasul următor nodul nevizitat cu distanță minimă la arbore. Pentru eficientizare, se folosește o coadă de priorități pentru stocarea datelor sub formă de triplete [*distanță minimă, nod, tata*].

La final, se creează o listă de adiacență notată *arbore* care reține pentru fiecare nod vecinii lui, lista fiind necesară pentru scăderea complexității parcurgerii care va fi realizată ulterior, deci a găsirii ciclului hamiltonian.

Pseudocod Prim(nod_start)

1. $distanță \leftarrow$ vector de lungime n inițializat cu ∞
2. $arbore \leftarrow$ listă de n liste ce reprezintă lista de adiacență
3. $vizitat \leftarrow$ vector de lungime n inițializat cu 0
4. $distanță[nod_start] \leftarrow 0$
5. $h \leftarrow [[0, nod_start, -1]]$ // rădăcina nu are ascendent
6. pentru *pași* $\leftarrow 1, n$ execută
7. $nod, tata \leftarrow$ nodul nevizitat cu distanță minimă din h , respectiv tatăl acestui nod

8. adaugă la lista $arbore[nod]$ nodul $tata$
9. adaugă la lista $arbore[tata]$ nodul nod
10. $vizitat[nod] \leftarrow 1$
11. pentru $vecin \leftarrow 0, n - 1$ execută
 12. dacă $d(nod, vecin) \neq \infty$ și $vizitat[vecin] = 0$ și $distanța[vecin] > d(nod, vecin)$ atunci
 13. $distanța[vecin] \leftarrow d(nod, vecin)$
 14. adaugă la h tripletul $[distanța(vecin), vecin, nod]$
15. returnează $arbore$

Următorul pas este dublarea muchiilor din arbore determinând astfel ciclul eulerian din care vom ajunge la ciclul hamiltonian prin aplicarea tehnicii de utilizare a scurtăturilor. Pentru eficientizare, nu este nevoie de efectuarea acestor pași pe rând, împreunarea lor fiind posibilă într-un mod simplu. Cele trei operații de dublare a muchiilor, de determinare a ciclului eulerian și de aplicare a tehnicii numite shortcutting rezultă de fapt o parcurgere în adâncime, un exemplu demonstrativ fiind prezentat în *Figura 3.3*. Așadar se realizează o parcurgere în adâncime astfel: se reține într-un vector de vizitați nodurile care aparțin deja ciclului și se evită reintroducerea lor în ciclu fiind acceptați în vector doar noduri nemaiîntâlnite. La final privind la ciclul din ultima reprezentare din *Figura 3.3*, se observă că nodurile alăturate formează ciclul hamiltonian dorit, iar atunci când au fost respinsă reintroducerea nodurilor deja vizitate s-a aplicat indirect shortcutting.

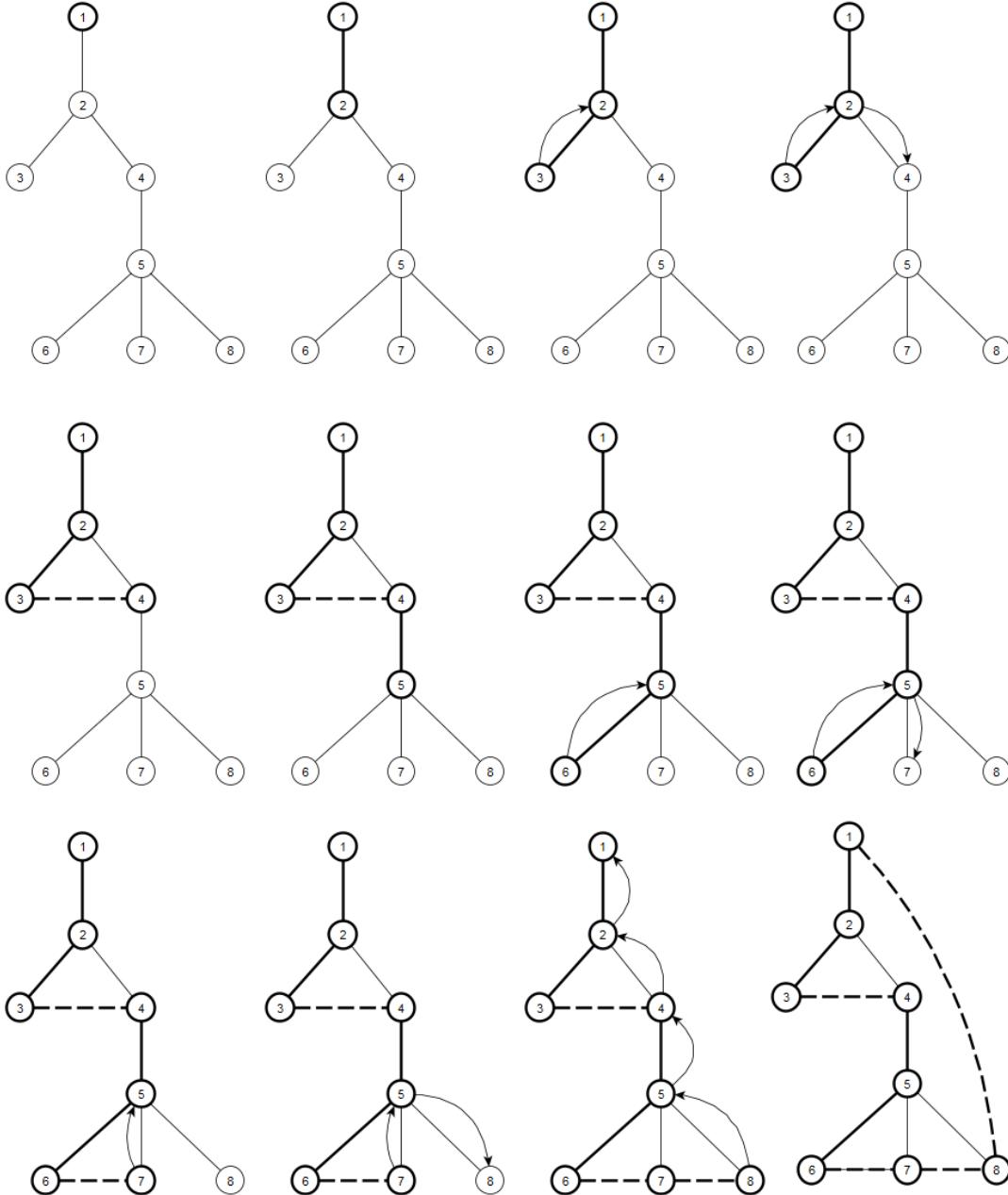


Figura 3.3. Reprezentarea modului în care funcționează o parcursere în adâncime (depth-first search, DFS) pe arborele parțial de cost minim din Figura 3.2.a. Se creează un ciclu din noduri în funcție de întâlnirea lor în parcursere, evitând reintroducerea nodurilor deja vizitate. Metoda este echivalentă cu crearea arborului dublu, a ciclului eulerian și aplicarea tehnicii scurtăturilor.

Un aspect important de specificat este modul de alegere a nodului inițial de la care se pornește parcurgerea. Având în vedere că parcurgerea este realizată alegând mai întâi nodurile care sunt pe o poziție mai mică, ciclul rezultat poate fi diferit în funcție de nodul de start. Un exemplu sugestiv este prezentat în *Figura 3.4.* pentru care se observă ca pentru arborele din *Figura 3.4.a.* unde nodul de start este 1, parcurgerea va determina ciclul 1, 0, 2, 3, 1, iar dacă este ales nodul 2 (arborele reprezentativ este în *Figura 3.4.b.*) ciclul creat este 2, 1, 0, 3, 2. Cum cele două nu sunt permutări circulare una celeilalte, sunt considerate diferite cu lungimi distincte.

Pentru a nu obliga algoritmul să construiască doar un anumit ciclu, caz rezultat din stabilirea unui nod de început care nu se modifică, se preferă alegerea lui în mod random. Astfel la rulări diferite ale programului, rezultatul final poate să fie diferit.

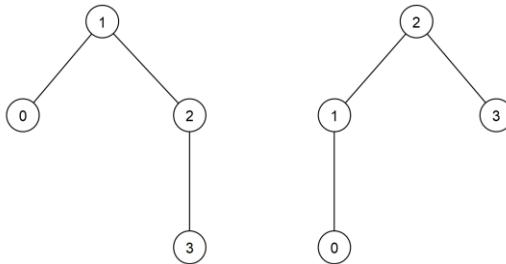


Figura 3.4. Prezentarea unui arbore simplist pentru a observa că în funcție de nodul de start parcurgerea în adâncime va rezulta cicluri diferite.

Pseudocod Ciclu_hamiltonian(arbore)

1. se alege random un nod de start și se notează cu k
2. $vizitat \leftarrow$ vector de lungime n inițializat cu 0
3. $ciclu \leftarrow$ listă inițial vidă în care se va reține ciclul hamiltonian
4. $Parcure(k, arbore)$
5. adaugă la ciclu nodul k // muchia de întoarcere

Pseudocod Parcure(i, arbore)

1. $vizitat[i] \leftarrow 1$
2. adaugă pe i la ciclu
3. pentru fiecare j din lista $arbore[i]$ adică fiecare j vecin al lui i execută
4. dacă $vizitat[j] = 0$ atunci

5.

Parcurgere(j, arbore)

Având ciclul hamiltonian, ultima funcție necesară este aceea care să determine soluția problemei adică ponderea ciclului.

Pseudocod Valoare_ciclu(ciclu)

1. $soluție \leftarrow 0$
2. pentru fiecare două noduri x și y alăturate în ciclu execută
3. $soluție \leftarrow soluție + d(x, y)$
4. returnează $soluție$

Având toate datele necesare, este nevoie să se apeleze inițial funcția *Prim(nod_start)* pentru a determina arborele parțial de cost minim prin vectorul de tată, apoi *Ciclul_hamiltonian* care primește ca parametru arborele determinat. Aceasta calculează ciclul prin apelarea funcției recursive *Parcurgere*. La final este nevoie de funcția *Valoare_ciclu(ciclu)* pentru a determina valoarea ciclului primit ca parametru care este determinat în metoda anterioară.

În [27] se poate urmări pseudocodul prezentat implementat în Python.

Complexitate

Teoremă 3.6.

Algoritmul arborelui dublu are complexitatea $O(n + m \log n)$.

Demonstrație:

Complexitatea algoritmului lui Prim este $O(m \log n)$ pentru că coada de priorități h returnează nodul cu distanță minimă până la arbore în $O(\log n)$ și per total sunt doar $O(m)$ inserări și extrageri în coada de priorități deoarece suma gradelor într-un graf este $2m$.

Complexitatea parcurgerii este $O(n + m)$ deoarece se folosește o listă de adiacență în loc de o matrice de adiacență și din acest motiv accesarea vecinilor unui nod ia O (numărul de vecini) în loc de a trece prin toate cele n noduri pentru a verifica care sunt vecinii. Parcurgerea fiind realizată pe un arbore înseamnă că $m = n - 1$, aşadar complexitatea rămâne $O(2n - 1) = O(n)$.

Iar determinare soluției (apelarea funcției *Valoare_ciclu*) costă $O(n)$ pentru că se trece prin ciclu pentru a calcula distanța dintre nodurile adiacente.

Complexitatea finală a algoritmului însumând complexitățile funcțiilor este $O(m \log n + 2n) = O(n + m \log n)$.

3.3. Algoritmul lui Christofides

Plecând de la algoritmul arborelui dublat, Christofides vine cu o îmbunătățire ce poate fi aplicată peste problemă dacă se află în cazul metric și duce la micșorarea factorului de aproximare la valoarea $\frac{3}{2}$ devenind algoritmul de aproximare cu cel mai mic factor de aproximare în momentul de față. Chiar dacă pare o aproximare mare, $\frac{3}{2}$ este pentru cazul cel mai nefavorabil și în practică (în medie) algoritmul poate dă rezultate mai bune.

Pașii algoritmului Christofides:

1. Se construiește un arbore de acoperire de cost minim
2. Se determină O - mulțimea nodurilor de grad impar din arbore
3. Se determină un cuplaj perfect de cost minim în graful induș de O
4. Se adaugă muchiile cuplajului la arbore
5. Se determină ciclul eulerian
6. Se aplică tehnica scurtăturilor peste ciclul eulerian găsit

Vom detalia în continuare acești pași.

La început se creează arborele parțial de cost minim. Însă trecerea de la arbore la graful eulerian se face în alt mod. Fiind un arbore, este sigur că are noduri impare (cel puțin pentru că are frunze) și suma tuturor gradelor nodurilor rămâne un număr par. Din această afirmație, se ajunge la concluzia că există în arborele de acoperire un număr par de noduri de grad impar. În *Figura 3.5.a.* sunt îngroșate nodurile de grad impar din arbore.

Fie O mulțimea nodurilor de grad impar: $|O| = 2k$ pentru care $k > 0$. Acum trecerea de la arbore la graf eulerian se face prin alegerea unor muchii care să unească nodurile din O două câte două, folosind algoritmul de determinare a unui cuplaj de cost minim (minimum-weight perfect matching). Muchiile alese de acest algoritm aplicat peste graful induș de mulțimea O se notează cu M .

Definiție 3.3.

Un cuplaj este o mulțime de muchii neadiacente (care nu au extremități în comun). De exemplu, dacă este aleasă muchia (x, y) , x sau y nu mai poate fi legat de un alt nod z (pentru fiecare nod se alege o singură muchie cu care este incident).

Un cuplaj se numește perfect dacă orice nod este extremitatea unei muchii din cuplaj (orice nod este legat de un singur alt nod printr-o muchie din cuplaj).

Într-un graf ponderat, un cuplaj este perfect de cost minim (min-cost perfect matching) dacă suma ponderilor muchiilor selectate este minimă.

Următorul pas este adăugarea noilor muchii din M ale cuplajului cu cost optim la arbore și se determină ciclul eulerian (trecerea prin toate muchiile o singură dată, fără a se accepta dublarea muchiilor) folosind parcurgerea 1, 3, 2, 6, 5, 7, 8, 5, 4, 2, 1 cum este afișat în *Figura 3.5.b*. Ciclul eulerian obținut are distanță totală $l(T) + l(M)$, unde T este arborele parțial de cost minim și M este mulțimea muchiilor din cuplajul perfect minim. Se aplică tehnica de shortcut-uri pe ciclul eulerian din *Figura 3.5.b*. pentru a crea ciclul hamiltonian: 1, 3, 2, 6, 5, 7, 8, 4, 1, iar distanța acestuia nu va depăși $l(T) + l(M)$ (*Figura 3.5.c*).

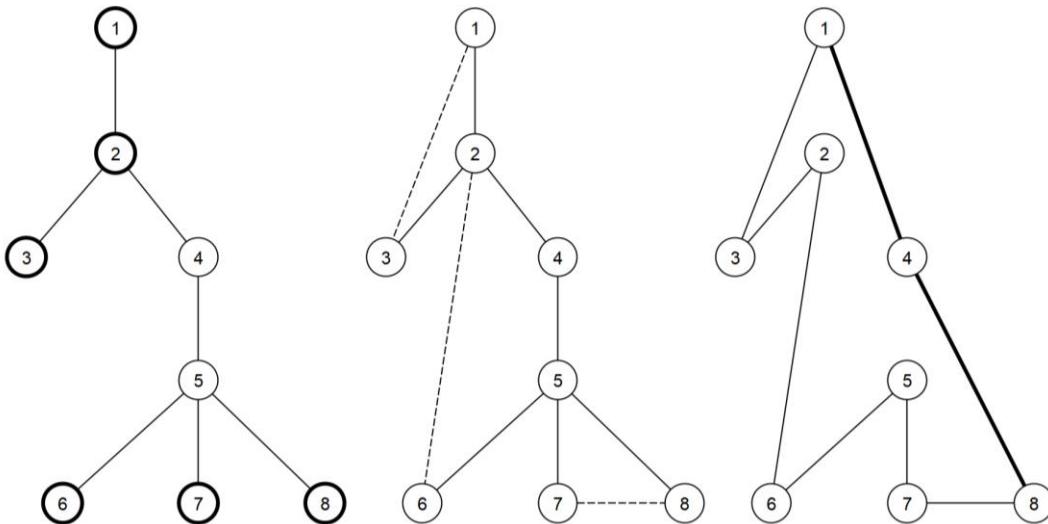


Figura 3.5. Continuarea exemplificării algoritmului lui Christofides pe arborelui din Figura 3.5.a. În primul arbore sunt îngroșate nodurile de grad impar, iar în cel din centru (b) sunt adăugate la arbore muchiile rezultate de matching, formându-se graful eulerian.

Figura 3.5.c. prezintă ciclul hamiltonian obținut din ciclul euclidian prin aplicarea metodei de shortcuting.

Teorema 3.7. [26]

Algoritmul lui Christofides are factorul de aproximare $\frac{3}{2}$.

Demonstrație:

Lungimea ciclului determinat de algoritmul lui Christofides nu depășește $l(T) + l(M)$ unde T este arborele parțial de cost minim pentru care a fost enunțat în *Teorema 3.4.* că $l(T) \leq OPT$ și M este mulțimea muchiilor alese de algoritmul de cuplaj. Acum ne axăm atenția pe $l(M)$.

Fie H un ciclu optimal care conține doar nodurile din O . La trecerea prin H , se colorează pe rând o muchie cu roșu și se adaugă în H_1 , apoi următoarea cu albastru și se adaugă în H_2 . H_1 și H_2 sunt cuplaje perfect. M este cuplajul perfect de cost minim, așadar:

$$\begin{array}{ll} l(M) \leq l(H_1) & \\ l(M) \leq l(H_2) & \Rightarrow 2l(M) \leq l(H) \quad (\text{a}) \\ l(H_1) + l(H_2) = l(H) & \end{array}$$

Din *Teorema 3.3*: $l(H) \leq OPT$ pentru că $H \subset G$ (b)

Din (a) și (b) rezultă că $2l(M) \leq OPT$, deci $l(M) \leq \frac{1}{2} OPT$.

Algoritmul este $l(T) + l(M) \leq \frac{3}{2} OPT$, așadar algoritmul lui Christofides este $\frac{3}{2}$ aproximativ.

Pseudocod

Sunt prezentate pașii de creare a algoritmului și pentru fiecare pas este creată o funcție separată. În construcție se va face analogie cu algoritmul arborelui dublu prezentat în *Subcapitolul 3.2.* deoarece în multe privințe seamănă, diferența fiind modul de determinare a grafului eulerian.

- Algoritmul lui Prim pentru a crea arborele de acoperire de cost minim

Diferența față de detaliile prezentate și implementarea în *Subcapitolul 3.2.* este că în cazul de față pe lângă reținerea arborelui în lista de adiacență, este momentul oportun să se determine gradul fiecărui nod pentru ca mai apoi să se afle care sunt acelea care sunt de grad impar.

Pseudocod Prim(*nod_start*)

```

1.   distanța  $\leftarrow$  vector de lungime  $n$  inițializat cu  $\infty$ 
2.   arbore  $\leftarrow$  listă de  $n$  liste ce reprezintă lista de adiacență
3.   grade  $\leftarrow$  vector de lungime  $n$  inițializat cu  $0$ 
4.   vizitat  $\leftarrow$  vector de lungime  $n$  inițializat cu  $0$ 
5.   distanța[nod_start]  $\leftarrow 0$ 
6.   h  $\leftarrow [[0, nod\_start, -1]]$  // rădăcina nu are ascendent
7.   pentru pași  $\leftarrow 1, n$  execută
8.       nod, tata  $\leftarrow$  nodul nevizitat cu distanță minimă din h, respectiv tatăl acestui nod
9.       adaugă la lista arbore[nod] nodul tata
10.      adaugă la lista arbore[tata] nodul nod
11.      grade[nod]  $\leftarrow grade[nod] + 1$ 
12.      grade[tata]  $\leftarrow grade[tata] + 1$ 
13.      vizitat[nod]  $\leftarrow 1$ 
14.      pentru vecin  $\leftarrow 0, n - 1$  execută
15.          dacă  $d(nod, vecin) \neq \infty$  și vizitat[vecin] = 0 și distanța[vecin] > d(nod, vecin) atunci
16.              distanța[vecin]  $\leftarrow d(nod, vecin)$ 
17.              adaugă la h tripletul [distanța(vecin), vecin, nod]
18.      returnează arbore, grade

```

- Determinarea mulțimii O de noduri de grad impar din arbore

Având la dispoziție rezultatul funcției Prim, se creează o listă în care se inserează nodurile care în vectorul *grade* au grad impar.

- Cuplajul perfect de cost minim pe nodurile din O , determinând mulțimea M de muchii, varianta cu factorul de aproximare de $\frac{3}{2}$. [20, 24]

Se amintește din *Definiția 3.3.* că un cuplaj este perfect dacă toate nodurile ajung să aibă câte o pereche. Metoda care creează cuplajul perfect de cost minim este una complexă ce se bazează pe algoritmul blossom a lui Karp.

În continuare se va prezenta pe scurt modul de gândire pentru algoritmul de determinare dacă există un cuplaj perfect într-un graf. Folosind idei similare, se poate afla și cel de lungime minimă. Următoarele idei sunt preluate din cursul “CSE202, Lecture 2: Edmond’s blossom algorithm” susținut de profesorul Seshadhri [20].

Definiție 3.4.

Fie M un cuplaj. O cale de alternanță (alternating path) într-un graf este o succesiune alternantă de muchii care aparțin din M și care nu aparțin.

Definiție 3.5.

Calea de creștere (augmenting path) este o cale de alternanță în care primul și ultimul nod sunt neacoperite (nu există în M nicio muchie incidentă cu el).

În acest caz, se trece prin calea de creștere găsită și se schimbă statusul muchiilor, obținându-se un cuplaj cu mai multe muchii.

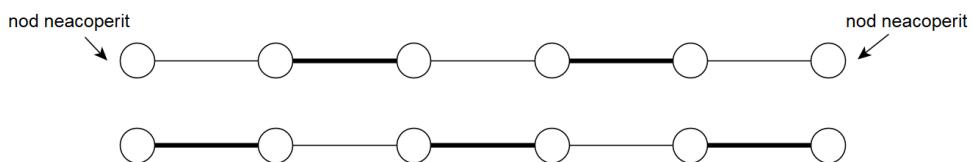


Figura 3.6. Prezentare a unei căi de creștere și a modului în care de modifică M .

Pentru a determina cuplajul perfect minim știind mulțimea O se construiește graful G_o pentru care există muchie între nodurile i și j dacă și numai dacă nu există muchia (i, j) în arborele parțial de cost minim.

Pentru găsirea căilor de creștere se construiește un arbore de alternanță T din graful G_o . Se alege ca rădăcină un nod neacoperit și drumurile de la nodul de start la frunze sunt căi de alternanță. Scopul este găsirea unor căi de creștere (frunze neacoperite) pentru a mări un cuplaj M .

Muchiile incluse în cuplaj sunt cele care sunt poziționate în T la o distanță impară față de rădăcină. Cu alte cuvinte, aceste muchii sunt formate din două noduri adiacente ce se găsesc în T pe două nivele consecutive, primul nivel impar și celălalt par. Din acest motiv nodurile de pe nivelurile impare au un singur fiu.

Pentru ușurătate, se colorează în *Figura 3.7* nodurile de pe niveluri pare cu albastru și cele de pe niveluri impare cu roșu.

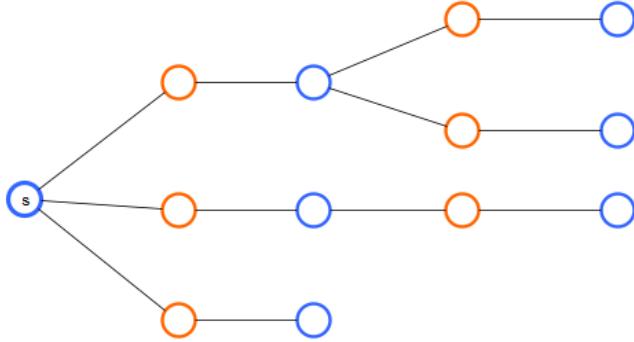


Figura 3.7. Reprezentarea unui arbore de alternanță (alternating tree).

Privind la arborele din *Figura 3.7*, se pornește dintr-un nod neacoperit și se parcurge arborele în căutarea unui căi de creștere. Dacă dintr-un nod albastru se poate ajunge într-un nod neacoperit înseamnă că s-a descoperit o cale de creștere, se schimbă statusul muchiilor și se obține un cuplaj cu mai multe muchii.

Așadar, se pornește de la o mulțime M fără elemente care va fi populată prin găsirea de căi de augmentare.

Algoritm

- se începe dintr-un nod neacoperit b și se consideră muchia (b, c) pentru care nodul $c \notin V(T)$
- *Caz 1:* c nu este acoperit de M , rezultând că s-a găsit o cale de creștere. Se inversează statusurile și se crește M
- *Caz 2:* c este deja în M . Așadar există un nod d pentru care $(c, d) \in M$ și $(c, d) \notin T$. În acest caz se crește arborele T adăugând (b, c) și (c, d) .

Algoritmul se repetă până când, la un moment dat, nu se mai face niciun progres. Pentru orice nod b de pe un nivel par, toate nodurile vecine cu b din G_o se află deja în T .

Definiție 3.6.

Un arbore se numește frustrat (frustrated) dacă pentru orice nod b de pe un nivel par, vecinii lui b din G_o se află deja în T , dar doar pe niveluri impare.

Dacă T este frustrat, nu există cuplaj perfect pentru graful G_o .

Construirea ciclului cu număr impar de muchii

În cazul în care arborele nu este frustrat înseamnă că modul în care s-a creat T a dus la blocaj, dar există o muchie în G_o între două noduri aflate pe niveluri pare. Pentru a continua căutarea de cuplaje, este folosit algoritmul blossom care căuta cicluri cu număr impar de noduri formate de o muchie de tipul (b, b') și drumul din arbore de la b la b' , $(b, b') \in G_o$, dar $(b, b') \notin T$ unde b și b' sunt două noduri de pe niveluri pare (*Figura 3.8*).

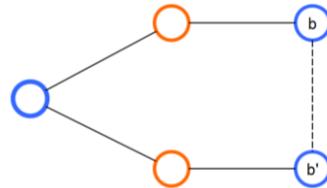
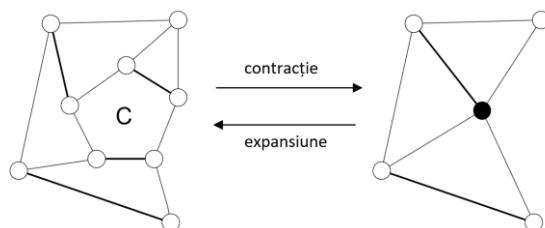


Figura 3.8. Reprezentarea unui ciclu cu număr impar de muchii găsit de algoritmul blossom.

Contractarea ciclului cu număr impar de noduri

Fie un graf G în care avem deja niște muchii cuplate și un ciclu C care trebuie contractat. Prin contractie se creează un alt graf G' în care se păstrează toate muchiile din G în afară de cele care formează ciclul C care se înlocuiește cu un singur nod numit C' . Toate nodurile incidente cu nodurile din C în G devin incidente cu nodul C' în G' .



Prin contractia unui ciclu cu număr impar de noduri, se poate construi cuplajul în noul graf G' , iar apoi se întoarce în graful inițial G și extinde nodul C' , păstrându-se numărul de noduri care nu sunt cuplate.

Folosind arborele de alternanță T din Figura 3.7, se presupune că există o muchie care formează în arbore un ciclu cu număr impar de noduri și se observă cum se contractă:

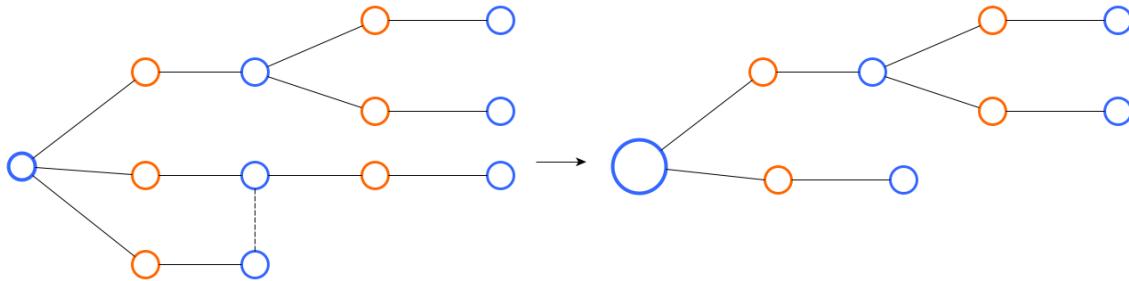


Figura 3.9. Prezentare a aplicării unei contracții pe arborele T din Figura 3.7.

Algoritm

1. Se începe cu un cuplaj. Dacă este perfect, algoritmul se oprește.
2. Se construiește un arbore de alternanță T . Se extinde până când nu se mai poate face niciun progres. Dacă s-a găsit o cale de creștere, modifică M și reîncepe arborele.
3. Dacă arborele este frustrat înseamnă că nu se poate construi un cuplaj perfect și algoritmul se oprește.
4. Alfel arborele are blossom-uri, contractează și mergi din nou la pasul 2.

După cum se observă, algoritmul de determinare dacă se poate construi un cuplaj perfect este unul complex. În ajutorul celor care vor să determine răspunsul la această problemă, și, desigur, să afle cuplajul de cost minim, biblioteca *NetworkX* s-a dovedit utilă, implementând aceste funcții.

Pentru a determina muchiile cuplajului perfect de cost minim există funcție predefinită *max_weight_matching* care dându-i-se distanțele negate dintre nodurile de grad impar, returnează o mulțime de perechi de noduri. Nu trebuie să se verifice dacă se poate crea un cuplaj perfect pentru că se știe deja că există pentru că avem un număr par de noduri în M .

Pentru că algoritmul de determinare a cuplajului este complex și dificil de implementat, se propune și o altă variantă pe baza strategiei de tip greedy care să determine un cuplaj însă nu garantează că costul cuplajului este minim. Aceste variante vor fi comparate în *Capitolul 5*.

- Cuplajul perfect pe nodurile din O , determinând mulțimea M de muchii folosind strategia greedy

Pentru a avea o variantă alternativă de a determina cuplajul se va crea o altă metodă care nu folosește algoritmul de determinare a cuplajului de cost minim pentru găsirea muchiilor, ci muchiile sunt alese greedy. Factorul de aproximare $\frac{3}{2}$ este legat de folosirea cuplajului perfect de cost minim după cum s-a demonstrat în *Teorema 3.7.*, aşadar înlocuind acea metodă cu un cuplaj ce se bazează pe o alegere greedy nu mai poate fi garantat factorul de aproximare. Totuși, pentru a oferi un cuplaj de pondere cât mai mică, se repetă de un număr mai mare de ori algoritmul greedy, la final păstrându-se varianta pentru care cuplajul are lungimea cea mai mică.

Pseudocod Alegere_GREEDY(arbore, noduri_grad_impar, repetări)

1. $copie \leftarrow$ copie a vectorului de noduri de grad impar
2. $soluție_{minimă} \leftarrow \infty$
3. $muchii \leftarrow$ listă vidă
4. $muchii_{alese} \leftarrow$ listă vidă
5. pentru $i \leftarrow 0, repetări - 1$ execută
 6. $cuplaj \leftarrow Fals$
 7. cât timp $cuplaj = Fals$ execută
 8. $cuplaj \leftarrow Adevărat$
 9. se amestecă nodurile din *noduri_grad_impar*
 10. $soluție \leftarrow 0$
 11. cât timp există noduri în *noduri_grad_impar* execută
 12. $u \leftarrow$ extrage un nod din *noduri_grad_impar*
 13. $lungime_{minimă} \leftarrow \infty$
 14. pentru fiecare nod v rămas *noduri_grad_impar* execută
 15. dacă $u \neq v$ și $d(u, v) < lungime_{minimă}$ și u și v nu erau deja vecini în *arbore* atunci
 16. $lungime_{minimă} \leftarrow d(u, v)$
 17. $pereche_u \leftarrow v$
 18. dacă s-a găsit o pereche lui u atunci
 19. adaugă muchia $(u, pereche_u)$ la lista *muchii*

```

20.           soluție  $\leftarrow$  soluție + lungime_minimă
21.           se scoate și nodul pereche_u din lista noduri_grad_impar
22.           altfel
23.           cuplaj  $\leftarrow$  Fals // ultimele noduri sunt deja vecine în
   arbore, nu se pot cupla din nou
24.           dacă soluție < soluție_minimă atunci
25.               soluție_minimă  $\leftarrow$  soluție
26.               muchii Alese  $\leftarrow$  muchii
27.           returnează muchii Alese

```

În variabilele *soluție_minimă* și *muchii Alese* se păstrează suma minimă a muchiilor unui cuplaj din cele *repetări* cuplaje determinate în mod greedy și muchiile cuplajului respectiv.

Strategia greedy este simplistă. Se extrage un nod *u* din lista de noduri de grad impar și se verifică care este cel mai apropiat nod *j* cu care se poate cupla dintre nodurile necuplate din listă. Acest nod *j* se setează ca pereche a nodului *u* (*pereche_u*). Se repetă modul de alegere până când nu mai există noduri de grad impar.

Ne dorim rularea de mai multe ori a algoritmului. Prin urmare, pentru a nu se determină de fiecare dată același cuplaj, lista nodurilor de grad impar se amestecă (linia 9 din pseudocod) adică se dorește schimbarea poziției nodurilor la fiecare încercare de a determina cuplajul perfect minim.

Chiar dacă există un număr par de noduri peste care se dorește realizarea unui cuplaj perfect, pot exista noduri pentru care cuplarea este imposibilă deoarece acele noduri sunt deja legate printr-o muchie în arborele parțial de cost minim. Se observă în pseudocod o variabilă *cuplaj* care este de tip boolean. Se pornește de la presupunerea că se poate crea un cuplaj perfect (variabila *cuplaj* ia valoarea *True*) și se ține evidența dacă cuplajul determinat este perfect sau nu. Dacă nu se poate găsi printre nodurile necuplate o pereche pentru nodul curent, *cuplaj* devine *False*. Nu se acceptă mulțimea de muchii găsite și se pornește din nou de la capăt căutarea, instrucțiune realizată în structura repetitivă *cât timp* de la linia 7. Dacă în schimb cuplajul este perfect se ieșe din structura repetitivă și se verifică dacă suma muchiilor cuplajului determinat este minimă. Această metodă se repetă de *repetări* ori după care se returnează lista de muchii (*muchii Alese*) pentru care suma muchiilor a fost minimă.

- Adăugarea muchiilor din M la arbore

Funcția *Adăugare_la_arbore* are ca scop introducerea în matricea *arbore* a nodurilor cuplate.

Pseudocod Adăugare_la_arbore

1. pentru fiecare muchie de forma (u, v) din *matching* execută
2. adaugă pe v la lista *arbore*[u]
3. adaugă pe u la lista *arbore*[v]

- Determinarea ciclului eulerian și aplicarea tehnicii de utilizare a scurtăturilor rezultând ciclul hamiltonian dorit

Ciclul hamiltonian nu se mai poate crea la fel ca cel din *Subcapitolul 3.2*. pentru că acum muchiile nu mai sunt dublate prin urmare nu se mai poate întoarcere în nodul precedent.

Pentru a determina ciclul eulerian se va folosi algoritmul lui Hierholzer [18] care este specializat în găsirea unui ciclu eulerian în graf fiind necesară însă specificația menționată în *Definiția 3.2*. că graful trebuie să fie conex și toate nodurile de grad par. Logica algoritmului este următoarea: se alege un nod de start u și se creează un drum din noduri până când se ajunge din nou în u mergând mereu pe muchii nevizitate. Nu este posibil să rămână blocat în orice alt nod în afară de u pentru că se respectă condiția ca toate nodurile să fie de grad par, deci există certitudinea că dacă drumul ajunge la un nod w trebuie să existe o muchie nevizitată prin care să iasă din w . Drumul format este de fapt un ciclu dar poate să nu acopere toate nodurile și muchiile grafului. Cât timp există un nod v inclus în drumul curent pentru care există o muchie incidentă nevizitată, trebuie să se creeze un alt drum care să pornească din v și să se dezvolte din muchii nevizitate până când se întoarce înapoi în v , adăugându-se apoi noul drum la cel inițial pe poziția în care se găsea nodul v .

Pentru a nu fi nevoie de crearea unui alt drum care să se insereze în cel inițial pe o poziție specifică, vom crea un drum care se va comporta ca o stivă în care se adaugă noduri după același criteriu discutat mai sus. Apoi când se ajunge din nou la nodul u și nu mai există nicio muchie neacoperită incidentă cu u , înseamnă că s-a provocat un blocaj (indiferent dacă s-a creat ciclul eulerian sau nu). Se scot noduri din drum și se păstrează într-o altă variabilă în ordinea în care se scot până când se ajunge la un nod v care încă mai are muchii incidente nevizitate și se continuă

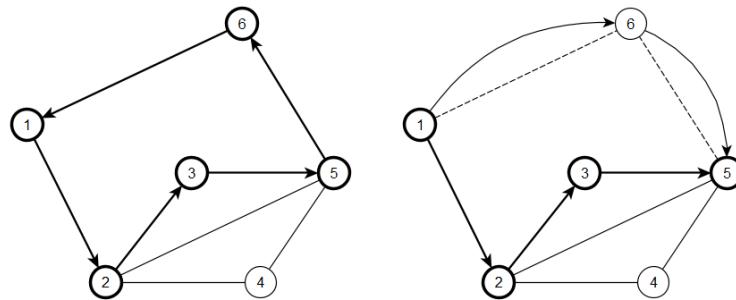
drumul plecând din v până când se provoacă un alt blocaj. Cât timp stiva nu ajunge goală înseamnă că mai există muchii nevizitate.

Pași algoritmului lui Hierholzer sunt următorii fiind apoi respectați în Figura 3.10.:

PAS 1: Se alege în mod aleator un nod de start u și se introduce ca prim nod la drum.

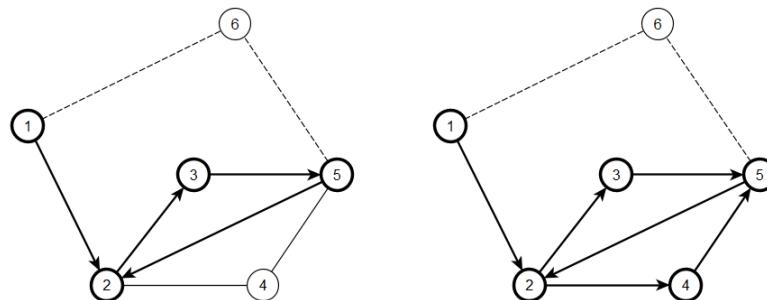
PAS 2: Se adaugă noduri la drum mergând mereu doar pe muchii nevizitate până se întâlnește un nod din care nu se mai poate pleca.

PAS 3: Se merge de la final spre început și se elimină din coadă noduri care se inserează într-o variabilă separată notată *ciclu* până când se ajunge într-un nod care mai are muchii adiacente nevizitate. Se repetă *PAS 2* și *PAS 3* până când nu se mai găsesc noduri nevizitate și toate nodurile sunt mutate în *ciclu*, creându-se astfel ciclul care conține toate muchiile.

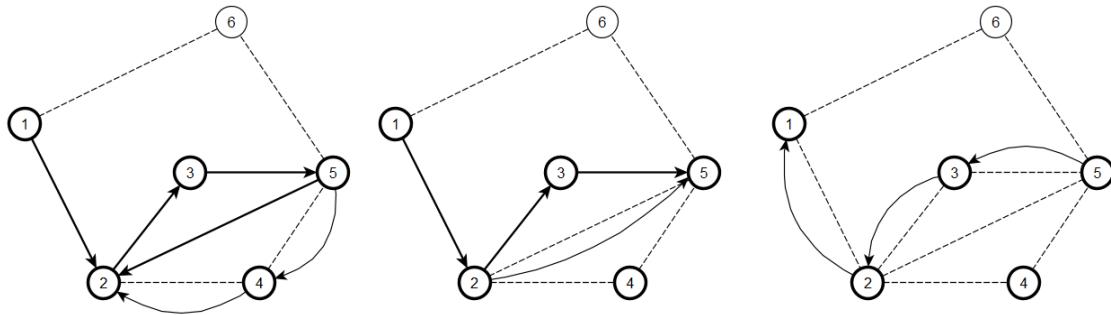


PAS 1, PAS 2, PAS 3: Se pleacă din nodul 1 și se parcurge un drum până când se ajunge din nou în nodul 1 unde se blochează nemaivând muchii nevizitate incidente. Se face 2 pași în spate până la nodul 5 care mai are vecini nevizitați de el și se păstrează în variabila „ciclu” nodurile

1 și 6.



Repetare PAS 2 și PAS 3: Din nodul 5 se merge pe muchia (5, 2) în nodul 2, apoi pe traseul 2-4-5 până când se blochează din nou.



Repetare PAS 3: Se merge înapoi reținându-se în continuare nodurile care se extrag din drum în lista „ciclul” în care au fost reținute și celelalte noduri: 5-4-2-5-3-2-1. Ajungând la primul nod și extrăgându-l și pe acela, s-a terminat parcursarea. Ciclul eulerian este [1, 6, 5, 4, 2, 5, 3, 2, 1].

Figura 3.10. Exemplificare algoritmul lui Hierholzer respectând pașii specificați. Se pleacă de la un graf eulerian și de la un nod de start ales, rezultând în urma algoritmului un ciclu eulerian.

Pseudocod ciclu_eulerian

1. fie k nodul de start
2. $drum \leftarrow [k]$
3. $ciclu \leftarrow []$
4. cât timp există noduri în $drum$ execută
 5. $nod_current \leftarrow$ ultimul nod din $drum$
 6. dacă $nod_current$ nu are muchii neparcurse incidente atunci
 7. adaugă $nod_current$ la $ciclu$
 8. scoate $nod_current$ din $drum$
 9. altfel
 10. $vecin \leftarrow$ vecin al lui $nod_current$ între care muchia este nevizitată
 11. inserează vecin în $drum$
12. returnează $ciclu$

Având ciclul eulerian returnat de metoda *ciclu_eulerian* ar trebui să se creează o nouă metodă care verifică dacă nodurile sunt distincte. În loc să se eliminate din ciclu orice reapariție a unui nod (excepție ultimul nod din ciclu care trebuie să fie același cu primul), se alege să se

modifice funcția precedentă (*ciclu_eulerian*) pentru a nu accepta în variabila *ciclu* noduri care sunt deja vizitate. Modalitatea propusă scade complexitatea pentru că eliminarea unui element de pe o poziție specificată din listă costă $O(n)$, în favoarea adăugării la coada listei care are complexitatea $O(1)$ și care se face deja în metoda existentă. Pe lângă aceasta, evită inserări inutile care urmau să fie eliminate anterior. Verificarea dacă nodul există în ciclu se realizează în $O(1)$ deoarece se creează la început lista *vizitat* care ține evidența nodurilor, iar verificarea constă în a vedea dacă *vizitat[nod_current]* este 0 sau 1. Așadar funcția *ciclu_eulerian* se transformă în funcția *ciclu_hamiltonian*:

Pseudocod ciclu_hamiltonian

1. fie k nodul de start
2. $drum \leftarrow [k]$
3. $ciclu \leftarrow []$
4. $vizitat \leftarrow$ listă de lungime n inițializată cu 0 pe fiecare poziție
5. cât timp există noduri în *drum* execută
 6. $nod_current \leftarrow$ ultimul nod din *drum*
 7. dacă *nod_current* nu are muchii neparcurse incidente atunci
 8. dacă *vizitat[nod_current]* = 0 atunci
 9. adaugă *nod_current* la *ciclu*
 10. *vizitat[nod_current]* $\leftarrow 1$
 11. dacă *drum* are un singur element atunci
 12. adaugă *nod_current* la *ciclu*
 13. scoate *nod_current* din *drum*
 14. altfel
 15. $vecin \leftarrow$ vecin al lui *nod_current* între care muchia este nevizitată
 16. inserează vecin în *drum*
 17. returnează *ciclu*

- Determinare soluție având ciclul: la fel ca la algoritmul arborelui dublu.

Complexitate

Teoremă 3.8. [13]

Algoritmul lui Christofides are complexitatea $O(n^3)$.

Demonstrație:

Așa cum a fost precizat în *Subcapitolul 3.2.*, algoritmul lui Prim are complexitatea $O(m \log n)$, determinarea mulțimii O de noduri impare costă $O(n)$, adăugarea muchiilor la arbore $O(k)$ unde k este numărul de muchii din cuplaj. Fie $t = k + n - 1$ numărul de muchii din graful rezultat.

Pentru a crea ciclul hamiltonian și privind la pseudocod, linia 5 se repetă de t pasi și în interiorul ei dacă se intră pe prima ramură a condiției adăugarea la finalul listei ciclu și ștergerea ultimului element din drum costă fiecare $O(1)$, iar dacă se intră pe a doua ramură, preluarea primului vecin prin eliminarea lui din lista de vecini a lui *nod_current* și apoi ștergerea lui *nod_current* din lista lui *vecin* sunt operații ce durează $O(n)$. Așadar complexitatea determinării ciclului hamiltonian este $O(t n)$. Cum $t = k + n - 1 < 2n - 1$, se ajunge la concluzia că complexitatea este $O(n^2)$.

La final, apelarea funcției *Valoare_ciclu* costă $O(n)$ după cum a fost precizat în *Subcapitolul 3.2.*

Dacă se folosește funcția predefinită din biblioteca *NetworkX*, determinarea muchiilor care creează cuplajul perfect de cost minim costă $O((2k)^3) = O(8 k^3) = O(k^3) = O(n^3)$.

Varianta care folosește greedy repetă algoritmul de alegere de *repetări* ori. Iar în sine algoritmul se plimbă prin noduri și cauță nodul disponibil care este cel mai aproape de sine, operație ce se execute în $O(k^2)$. Prin urmare, varianta ce folosește greedy durează $O(repetări k^2) = O(n^2)$. Dacă numărul de repetări este mic, este neglijabil.

În concluzie, complexitatea algoritmului lui Christofides este $O(n^3)$. Iar dacă se alege strategia greedy, complexitatea este $O(n^2 + m \log n)$.

4. Algoritmi euristici de tip greedy [6, 11, 12, 19]

În acest capitol se discută despre un TSP a cărui instanță este reprezentată de un graf complet G cu ponderi ne-negative a cărei matrice corespunzătoare este simetrică.

Algoritmii euristici de tip greedy sunt algoritmi buni ca performanță chiar dacă nu asigură corectitudinea soluției găsite și nu garantează neapărat un anumit factor de aproximare constant.

Euristicile TSP-ului sunt proceduri de construire a unui ciclu hamiltonian și de optimizare a acestuia.

4.1. Construirea unui ciclu hamiltonian

În această secțiune vom prezenta mai mulți algoritmi de determinare a unui ciclu hamiltonian de cost cât mai mic. Vom aduce în discuție următoarele euristică: algoritmul de inserție a celui mai îndepărtat nod (farthest insertion), algoritmul de inserție a celui mai apropiat nod (nearest insertion), algoritmul de inserție cu cel mai mic cost (cheapest insertion) și algoritmul cel mai apropiat vecin (nearest neighbor). Acești algoritmi de tip greedy construiesc un ciclu hamiltonian prin adăugarea treptată a câte un nod la drumul curent construit, strategia de alegere fiind sugerată de numele algoritmului.

Urmărirea implementării acestor algoritmi în Python se poate face prin accesarea linkului din [27].

Algoritmii vor fi explicați vizual plecând de la graful din *Figura 4.1.a.* și se va alege că nod de start nodul notat cu 1.

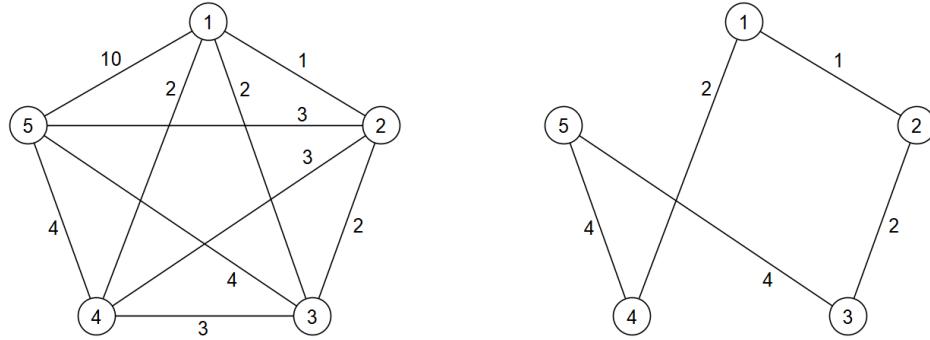


Figura 4.1. În Figura 4.1.a. este prezentat un graf neorientat complet ponderat ce va fi utilizat a pentru exemplificarea algoritmilor pas cu pas. Figura 4.1.b. afișează ciclul hamiltonian care oferă soluția optimă. Această soluție are valoarea 13.

4.1.1. Algoritmul de inserție a celui mai îndepărtat nod

Algoritmul de inserție a celui mai îndepărtat nod (farthest insertion) este un algoritm de tip greedy în care se caută, pentru a extinde ciclul construit până la pasul curent, cel mai îndepărtat nod j față de orice nod i din drum. Acesta se inserează în drum pe poziția care avantajează lungimea drumului, evitându-se astfel adăugarea unor muchii de lungime mare la drum. Matricea este completă și simetrică.

Algoritm

Algoritmul începe cu un nodul de start ales în mod aleator, apoi se repetă următorii pași până când ciclul conține *toate* noduri:

- Dintre toate nodurile nevizitate, se alege acela a cărui distanță minimă față de nodurile din ciclu este maximă comparativ cu distanța celorlalte noduri nevizitate. Fie ciclul curent de forma $(i_1, i_2, \dots, i_{k-1}, i_k, i_1)$ și distanța(j) = $\min \{d(i_1, j), d(i_2, j), \dots, d(i_k, j)\}$, se alege j cu distanța(j) maximă.
- Se inserează în ciclu nodul j pe poziția q în care ar crește cel mai puțin lungimea drumului, adică poziția q care are valoarea $d(i_q, j) + d(j, i_{q+1}) - d(i_q, i_{q+1})$ minimă.

Pseudocod FARTHEST(D matrice de distanțe)

1. se alege un nod random notat *nod_start*
2. *soluție* $\leftarrow 0$
3. *ciclu* $\leftarrow [nod_start]$ // listă în care se formează ciclul de distanță minimă, inițializată cu nodul de start; de reținut ca între nodurile indexate pe poziții consecutive se află muchie
4. *h* \leftarrow listă de lungime *n* în care se rețin distanțele de la nodurile care mi fac parte din ciclu la ciclu
5. pentru *i* $\leftarrow 0, n - 1$ execută
 6. dacă *i* $\neq nod_start$ atunci
 7. *h[i]* $\leftarrow d(nod_start, i)$
 8. cât timp ciclul nu are încă *n* noduri execută
 9. reține în *k* nodul atribuit valorii maxime din *h* știind că se iau în calcul doar nodurile nevizitate
 10. dacă ciclul este format dintr-un singur nod atunci
 11. introdu în ciclu nodul *k*
 12. marchează *k* ca vizitat
 13. *soluție* $\leftarrow 2 * h[k]$
 14. altfel
 15. *creștere_minimă* $\leftarrow \infty$
 16. *m* \leftarrow dimensiune ciclu
 17. pentru fiecare muchie (*ciclu[i]*, *ciclu[(i+1) % m]*) din ciclu execută
 18. *a* $\leftarrow ciclu[i]$
 19. *b* $\leftarrow ciclu[(i+1) \% m]$
 20. *creștere* $\leftarrow d(a, k) + d(k, b) - d(a, b)$
 21. dacă *creștere_minimă* $>$ *creștere* atunci
 22. *creștere_minimă* $\leftarrow creștere$
 23. *poziție_inserare* $\leftarrow i$
 24. introdu în ciclu nodul *k* pe poziția *poziție_inserare* + 1
 25. marchează *k* ca vizitat

```

26.            $soluție \leftarrow soluție + creștere\_minimă$ 
27.       pentru  $i \leftarrow 0, n - 1$  execută
28.           dacă  $i$  nu este vizitat și  $d(i, k) < h[i]$  atunci
29.                $h[i] \leftarrow d(i, k)$ 
30.       returnează  $soluție$  și  $ciclu$ 

```

La început se generează nodul de start nod_start și se inițializează h cu distanțele de la nod_start la orice alt nod din graf. La fiecare pas, valoarea soluției și ciclul se actualizează, primind încă un nod, nod ce corespunde celui mai îndepărtat nod de ciclu (linia 9). După ce se determină nodul de introdus notat cu k , trebuie să se decidă poziția optimă, adică aceea care să crească valoarea soluției cel mai puțin. După actualizări (liniile 10-26), se necesită și modificarea listei h , pentru a o aduce la starea curentă a ciclului, verificând dacă, prin introducerea nodului k , distanța de la nodurile nevizitate la ciclu se poate micșora.

Algoritmul se încheie când ciclul creat conține toate nodurile.

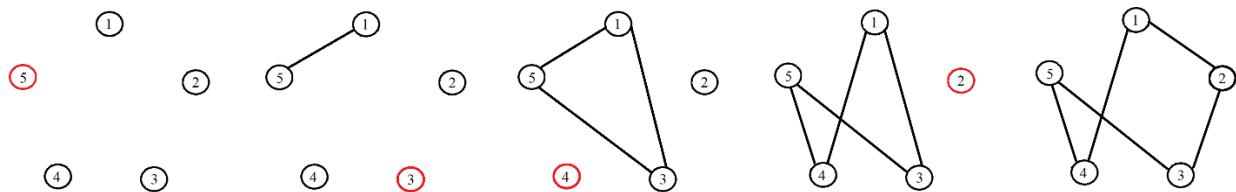


Figura 4.2. Pașii de rulare ai algoritmului de inserție a celui mai îndepărtat nod. La fiecare pas este înroșit nodul care urmează să fie introdus în ciclu. Pentru nodul initial setat 1, se obține ciclul hamiltonian $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$ și soluția este 13.

Golden, Bodin, Doyle și Stewart au specificat în articolul „Algoritmi aproximativi pentru problema comis-voiajorului” [6] că algoritmul este $2 \ln n + 0.16$ aproximativ, aşadar chiar și în cel mai rău caz:

$$\frac{\text{soluția FARTHEST}}{OPT} \leq 2 \ln n + 0.16$$

Complexitate

Teoremă 4.1.[6]

Algoritmul de inserție a celui mai îndepărtat nod rulează în $O(n^2)$.

Demonstrație:

Structura repetitivă ce începe la linia 8 din pseudocod execută n pași și în interiorul ei luarea informației din h (linia 9) și introducerea unor alte valori (liniile 27-29) costă $O(n)$, structura repetitivă aflată la liniile 17-23 se execută în maxim $O(n)$, introducerea nodului în ciclu tot $O(n)$ (linia 24).

Așadar, per total, interiorul structurii de la linia 8 se realizează în $O(n)$ și împreună cu cele n repetări, rezultă complexitatea dorită $O(n^2)$.

4.1.2. Algoritmul de inserție a celui mai apropiat nod

Algoritmul de inserție a celui mai apropiat nod (nearest insertion algorithm) este un algoritm de tip greedy care are ca scop determinarea unui ciclu hamiltonian de lungime minimă. Ciclul hamiltonian se construiește treptat pornind de la un singur nod, iar la fiecare pas dintre toate nodurile nevizitate se adaugă acela care este cel mai apropiat ca distanță față de orice nod din ciclu, acesta fiind apoi introdus în ciclu pe poziția care aduce cea mai mică creștere a lungimii ciclului.

După cum se observă, seamană cu algoritmul de inserție a celui mai îndepărtat nod, diferența fiind modul de selectare a noului nod care va fi introdus în ciclu.

Algoritm

Algoritmul începe cu un nodul de start ales în mod aleator, apoi se repetă următorii pași până când ciclul conține *toate* noduri:

- Dintre toate nodurile nevizitate, se alege acela a cărui distanță minimă față de nodurile din ciclul este minimă comparativ cu distanța celorlalte noduri nevizitate. Fie ciclul curent de forma $(i_1, i_2, \dots, i_{k-1}, i_k, i_1)$ și distanța(j) = $\min \{d(i_1, j), d(i_2, j), \dots, d(i_k, j)\}$, se alege j cu distanța(j) minimă.
- Se inserează în ciclu nodul j pe poziția q în care ar crește cel mai puțin lungimea drumului, adică poziția q care are valoarea $d(i_q, j) + d(j, i_{q+1}) - d(i_q, i_{q+1})$ minimă.

Așadar, linia 9 din pseudocodul algoritmului de inserție a celui mai îndepărtat nod se înlocuiește astfel:

9'. reține în k nodul atribuit valorii *minime* din h știind că se iau în calcul doar nodurile nevizitate

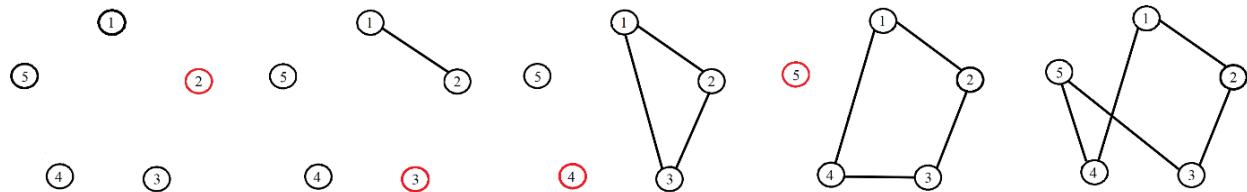


Figura 4.3. Pașii de rulare a algoritmului de inserție a celui mai apropiat nod. La fiecare pas este înroșit nodul care urmează să fie introdus în ciclu. Pentru nodul inițial setat 1, se obține ciclul hamiltonian $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$ și soluția este 13.

Golden, Bodin, Doyle și Stewart au specificat în articolul „Algoritmi aproximativi pentru problema comis-voiajorului” [6] că algoritmul este 2 aproximativ, aşadar chiar și în cel mai rău caz:

$$\frac{\text{soluția NEAREST}}{\text{OPT}} \leq 2$$

Complexitate

Teoremă 4.2.[6]

Algoritmul de inserție a celui mai apropiat nod are complexitatea în $O(n^2)$.

Demonstrație:

În afară de această modificare, algoritmul funcționează la fel, prin urmare și acesta se încadrează în $O(n^2)$ conform *Teoremei 4.1.* în care se analizează pseudocodul.

4.1.3. Algoritmul de inserție cu cel mai mic cost

Algoritmul de inserție cu cel mai mic cost (cheapest insertion algorithm) este un algoritm greedy care are ca scop determinarea unui ciclu hamiltonian de cost cât mai mic. La fiecare pas, se introduce la ciclul curent nodul ce aduce distanței ciclului cea mai mică creștere, introducându-se nodul pe poziția care îl avantajează.

Algoritmul seamană cu algoritmii de la subsecțiunile 4.1.1. și 4.1.2., diferența fiind ca și în cazul precedent, modul de selectare a noului nod care va fi introdus în ciclu.

Algoritm

Algoritmul începe cu un nodul de start ales în mod aleator, apoi se repetă următorii pași până când ciclul conține *toate* noduri:

- Dintre toate nodurile nevizitate, se alege acela care aduce cea mai mică creștere a lungimii ciclului dacă este introdus între două noduri deja aflate în ciclu. Fie ciclul curent de forma $(i_1, i_2, \dots, i_{k-1}, i_k, i_1)$ și se alege j cu $d(i_p, j) + d(j, i_q) - d(i_p, i_q)$ minimă, unde i_p și i_q sunt noduri consecutive.
- Se inserează în ciclu nodul j între nodurile pentru care distanța adăugată este minimă.

Pseudocod CHEAPEST(D matrice de distanțe)

1. se alege un nod random notat *nod_start*
2. $ciclu \leftarrow [nod_start]$ // listă în care se formează ciclul de distanță minimă, inițializată cu nodul de start; de reținut că între nodurile alăturate se află muchie
3. $h \leftarrow$ listă în care se rețin triplete de forma
[creștere, poziție din stânga locului în care ar fi adăugat nodul k , k]
4. determină cel mai apropiat nod de *nod_start* și se notează cu k
5. introdu în ciclu nodul k
6. $soluție \leftarrow 2 * d(nod_start, k)$
7. pentru $i \leftarrow 0, n - 1$ execută
 8. dacă $i \neq ciclu[0]$ și $i \neq ciclu[1]$:
 9. $creștere \leftarrow d(ciclu[0], j) + d(j, ciclu[1]) - d(ciclu[0], ciclu[1])$
 10. adaugă la h tripletul [$creștere$, 0 , i]
11. cât timp ciclul nu are încă n noduri execută
12. extragă din h tripletul cu creșterea minimă și reține valorile în *creștere_minimă*, *poziție_inserare*, k
13. introdu în ciclu nodul k pe poziția *poziție_inserare* + 1
14. marchează k ca vizitat
15. $soluție \leftarrow soluție + creștere_minimă$

```

16.       $m \leftarrow$  dimensiune ciclu
17.      pentru orice triplet din  $h$  de forma creștere, poziție, nod execută
18.          dacă poziție = poziție_inserare sau nod =  $k$  atunci
19.              elimină tripletul din  $h$ 
20.          dacă poziție > poziție_inserare și nod nu este vizitat atunci
21.              poziție  $\leftarrow$  (poziție + 1) %  $m$ 
22.          pentru  $i \leftarrow 0, n - 1$  execută
23.              dacă  $i$  nu este vizitat atunci
24.                   $a \leftarrow ciclu[poziție\_inserare]$ 
25.                   $b \leftarrow ciclu[(poziție\_inserare + 2) \% m]$ 
26.                  adaugă la  $h$  tripletul [ $d(a, i) + d(i, k) - d(a, k)$ , poziție_inserare,  $i$ ]
27.                  adaugă la  $h$  tripletul [ $d(k, i) + d(i, b) - d(k, b)$ ,
28.                               (poziție_inserare+2) %  $m$ ,  $i$ ]
29.      returnează soluție și ciclu

```

Inițial se dorește crearea unui ciclu din două noduri cu scopul de a-l extinde treptat. Așadar, după ce a fost generat nodul de start *nod_start*, se caută cel mai apropiat nod de acesta, numit k (linia 4 din pseudocod). Pentru că este un ciclu, costul acestuia este de două ori costul distanței de la *nod_start* la k , iar mai apoi se introduce în lista h opțiunile din care se va alege în viitor creșterea minimă posibilă.

Algoritmul rulează cât timp ciclul nu are încă n noduri, la fiecare pas extragându-se din h creșterea minimă posibilă, respectiv locul în care se dorește inserarea nodului în ciclu și nodul k . Este nevoie să se actualizeze valoarea soluției și ciclul prin introducerea nodului găsit pe poziția în care avantajează distanța ciclului (liniile 13, 15). Având în vedere că nodul k devine parte din ciclu, trebuie să se modifice lista h , eliminându-se din ea tripletele care trătau nodul k precum un nod nevizitat și acele care considerau că între i_p și i_q nu a fost introdus un alt nod. Adică tripletele pentru care distanța calculată nu mai reflectă starea actuală a ciclului (liniile 18, 19). De asemenea, trebuie să se actualizeze poziția de inserare pentru tripletele care vizează poziții ce se află după poziția extrasă din h . Aceasta deoarece, inserând nodul pe poziția *poziție_inserare*, toate celelalte noduri de după el se mută mai la dreapta cu o poziție (liniile 20, 21). La final, se

completează lista h cu triplete în care sunt calculate distanțele ce îl includ și pe k la nodurile ciclului (liniile 20-25).

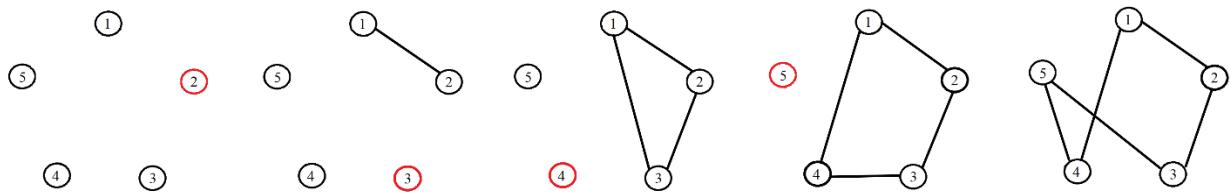


Figura 4.4. Pașii de rulare a algoritmului de inserție cu cel mai mic cost. La fiecare pas este înroșit nodul care urmează să fie introdus în ciclu. Pentru nodul inițial setat 1, se obține ciclul hamiltonian $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow 1$ și soluția este 13.

Golden, Bodin, Doyle și Stewart au specificat în articolul „Algoritmi aproximativi pentru problema comis-voiajorului” [6] că algoritmul este 2 aproximativ, la fel ca și algoritmul a celui mai apropiat nod. Așadar chiar și în cel mai rău caz:

$$\frac{\text{soluția CHEAPEST}}{OPT} \leq 2$$

Complexitate

Teoremă 4.3. [6]

Algoritmul de inserție cu cel mai mic cost rulează în $O(n^2 \log n)$.

Demonstratie:

Pentru a eficientiza determinarea minimului din lista h , se lucrează cu o listă de priorități, operațiile de adăugat și de scoatere a unui element costând $O(\log n)$. Determinarea celui mai apropiat nod de nodul de start (linia 4) costă $O(n)$, iar inserarea în h a primelor triplete (liniile 7-10) rulează în $O(n \log n)$. Timpul de execuție $O(n^2 \log n)$ este dat însă de structura repetitivă care începe la linia 11. Aceasta se rulează de n ori și în interiorul ei extragerea din h a tripletului minim necesită $O(\log n)$, actualizarea ciclului $O(n)$, eliminarea și adăugare de triplete în lista de priorități $O(n \log n)$. Concluzionând, interiorul structurii repetitive rulează în $O(n \log n)$ și se repetă de n ori, rezultând complexitatea $O(n^2 \log n)$.

4.1.4. Algoritmul cel mai apropiat vecin

Cel mai simplu algoritm heuristic este algoritmul cel mai apropiat vecin (nearest neighbor algorithm, NN). Dându-se un graf complet cu ponderi ne-negative, se determină un ciclu hamiltonian în care nodurile sunt alese în funcție de apropierea dintre ele. Așadar, la fiecare pas se introduce la drumul curent nodul care nu face încă parte din el și care este cel mai aproape ca distanță de ultimul nod din drum.

Algoritm

Algoritmul începe cu un nodul de start ales în mod aleator, apoi se repetă următorii pași până când drumul conține *toate* noduri:

- Dintre toate nodurile nevizitate, se alege acela care este cel mai aproape de ultimul nod din drum. Fie drumul curent de forma $(i_1, i_2, \dots, i_{k-1}, i_k)$ și se alege j cu distanța $d(i_k, j)$ minimă.
- Se inserează nodul j la finalul drumului.

Pseudocod NN(D matrice de distanțe)

1. se alege un nod random *nod_start* și se setează ca primul nod din ciclu
2. *valoare_ciclu* $\leftarrow 0$
3. *i* \leftarrow *nod_start*
4. cât timp există noduri nevizitate execută
 5. pentru nodul curent *i* introdu în ciclu nodul *j* nevizitat cu $d(i, j)$ minim
 6. *valoare_ciclu* \leftarrow *valoare_ciclu* + $d(i, j)$
 7. marchează *j* ca vizitat
 8. *i* \leftarrow *j*
9. *valoare_ciclu* \leftarrow *valoare_ciclu* + $d(i, nod_start)$
10. returnează ciclul creat și *valoare_ciclu*

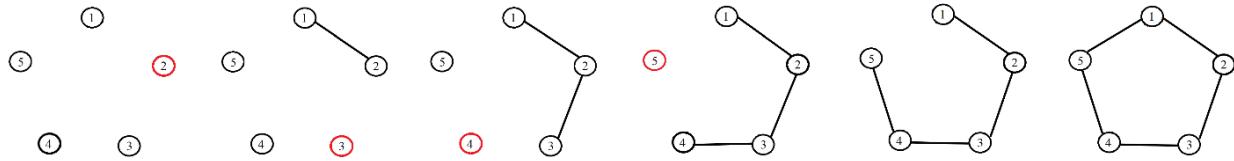


Figura 4.5. Pașii de rulare a algoritmului cel mai apropiat vecin.

Pentru nodul inițial setat 1, se obține ciclul hamiltonian $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$ și soluția este 20.

Ciclul determinat de algoritmul NN prezentat în Figura 4.5. nu oferă soluția optimă, iar dacă s-ar modifica valoarea muchiei de întoarcere (pasul 9 din pseudocod, muchia (4,5) care are valoarea 10) și nodul de start ar fi același, algoritmul greedy ar fi ales același ciclu. De asemenea, se observă problema că, dacă nodul de start ar fi altul, nu s-ar mai ajunge la ciclul prezentat în Figura 4.5. (un exemplu fiind Figura 4.6.). Prin urmare, și valoarea ciclului ar fi diferită. Așadar, din cauza faptului că folosește o strategie greedy, algoritmul cel mai apropiat vecin nu este unul favorabil în starea sa actuală pentru a rezolva TSP.

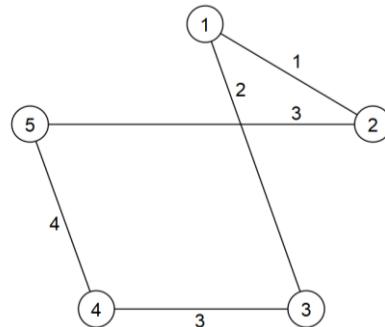


Figura 4.6. Ciclul hamiltonian obținut prin metoda NN plecând din nodul de start 2 și în care la pasul $i = 1$ se alege $j = 3$. Valorile muchiilor (1, 3) și (1, 4) sunt ambele egale cu 2. Se presupune alegerea primei muchii de lungime minimă. Se ajunge la soluția lungimii ciclului 13.

Golden, Bodin, Doyle și Stewart au specificat în articolul „Algoritmi aproximativi pentru problema comis-voiajorului” [6] că algoritmul este $\frac{1}{2}\ln n + \frac{1}{2}$ aproximativ. Așadar chiar și în cel mai rău caz:

$$\frac{\text{soluția NN}}{\text{OPT}} \leq \frac{1}{2}\ln n + \frac{1}{2}$$

Complexitate

Teoremă 4.4.[6]

Complexitatea algoritmului NN este $O(n^2)$.

Demonstrație:

Algoritmul este simplist, complexitatea fiindu-i determinată de necesitatea repetării structurii repetitive de la liniile 4-8 de n ori și de faptul că trebuie aflat nodul cel mai apropiat de nodul curent, operație care se efectuează în $O(n)$.

Algoritmii prezentați nu sunt exacti, factorul de aproximare fiind menționat pentru fiecare algoritm în parte, în subcapitolul respectiv. Ne dorim o valoare cât mai apropiată de cea optimă, prin urmare ciclul hamiltonian determinat este modificat în algoritmii 2-OPT și 3-OPT. Scopul acestor algoritmi este îmbunătățirea soluției.

4.2. Micșorarea costului unui ciclu hamiltonian

Pentru crearea unui ciclu hamiltonian se poate folosi orice algoritm prezentat în *Subcapitolul 4.1.*, dar pentru a exemplifica pas cu pas se va alege în acest subcapitol ciclul rezultat din algoritmul cel mai apropiat vecin (NN). Având ciclul hamiltonian, următorul pas este îmbunătățirea costului lui până când nu se mai poate. La final nu se poate garanta că ciclul obținut este minim. Reluarea algoritmului de mai multe ori folosind cicluri posibil diferite duce la creșterea probabilității găsirii soluției minime.

Definiție 4.1.

Fie F mulțimea tuturor ciclurilor hamiltoniene. O vecinătate este o mapare $N : F \rightarrow 2^F$ în care N mapează fiecare $f \in F$ în vecinătatea $N(f)$. Lin propune vecinătățile pentru un TSP pe K_n . Pentru f un ciclu și $k \in \{2, \dots, n\}$, $N_k(f)$ este mulțimea tuturor ciclurilor hamiltoniene g care sunt obținute din f prin eliminarea a k muchii și înlocuirea lor cu k muchii (nu neapărat toate muchiile adăugate trebuie să fie diferite de cele eliminate). $N_k(f)$ poartă numele de vecinătate k -schimburi (k -change neighborhood), iar oricare g a cărui distanță este minimă dintre toate celelalte cicluri din $N_k(f)$ se numește k -optimal.

Pseudocod k-OPT(D)

1. alege un ciclu inițial f
2. cât timp există $g \in N_k(f)$ cu $d(g) < d(f)$ execută
3. alege $g \in N_k(f)$ cu $d(g) < d(f)$
4. $f \leftarrow g$

Explicație algoritm

Se consideră o mulțime de soluții $N_k(f)$ care sunt vecine cu soluția curentă f și se caută, din această mulțime, optimul local, ciclul g care aduce cea mai însemnată îmbunătățire. Algoritmul se oprește când nu se mai găsește niciun optim local. Așadar, dacă ciclul hamiltonian curent nu are vecini care să ofere o soluție mai bună se consideră că nu se mai poate optimiza.

În practică este propusă varianta $k = 3$ pentru un rezultat cât mai apropiat de soluția optimă. Înainte de a vorbi de aceasta, vom discuta despre varianta $k = 2$ pentru a înțelege conceptele și a ușura explicația pas cu pas a algoritmului.

Complexitate

Chiar dacă există multe cicluri posibile, numărul lor este finit. După fiecare iterație a algoritmului k-OPT, se produce un ciclu hamiltonian cu lungime strict mai mică ca cel precedent, prin urmare ciclurile nu se pot repeta.

Timpul de rulare este dat de numărul de iterări a ciclului principal „când timp” înmulțit cu numărul de operații realizate de o iterărie. Având în vedere că sunt n noduri, există $O(n^k)$ k-schimburi care trebuie verificare la fiecare iterărie (k reprezintă numărul de muchii care vor fi înlocuire, prin urmare variază în funcție de problema pentru care se dorește rezolvare). Problema apare la numărul total de iterări care poate ajunge să fie exponential, acesta depinzând de ciclul hamiltonian inițial și de setul de date primit ca input.

4.2.1. Algoritmul 2-OPT

Algoritmul presupune optimizarea algoritmului prin aplicare de 2-schimburi, prin urmare ciclul hamiltonian creat la primul pas continuă să fie modificat prin schimbarea a 2 muchii, producând un ciclu nou g care este mai bun sau mai puțin optim ca cel original.

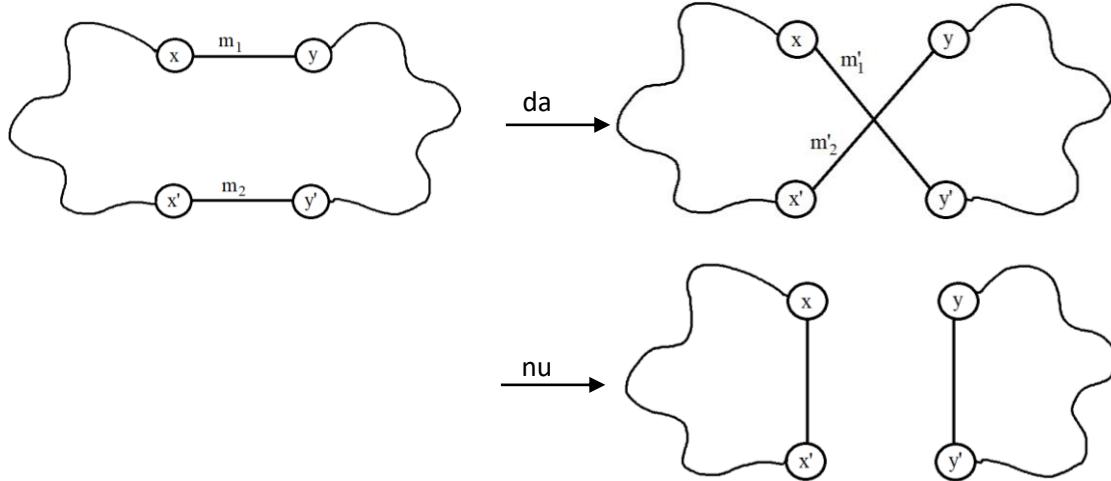


Figura 4.7. Modalitatea de a realiza 2-schimburi (2-change). Prima figură prezintă un ciclu ce conține cele două muchii (x, y) și (x', y') . A doua figură modalitatea corectă de a conecta cele 4 noduri, creându-se un nou ciclu. A treia figură arată modalitatea greșită de a conecta întrucât se distrugе ciclul.

Fie $\delta(g)$ avantajul ciclului nou g comparativ cu ciclul anterior f . $\delta(g) = d(m_1) + d(m_2) - d(m'_1) - d(m'_2) > 0$ înseamnă că s-a găsit un ciclu de lungime mai mică adică s-a produs o îmbunătățire.

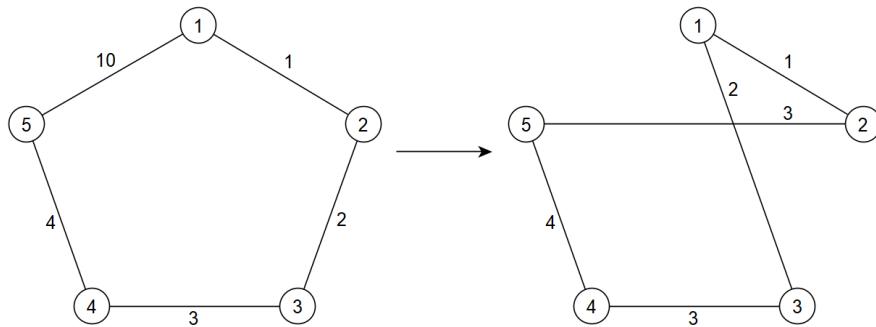


Figura 4.8. Graful din stânga este ultimul din Figura 4.5. reprezentând ciclul hamiltonian determinat de algoritmul NN. Cel din dreapta este realizat prin optimizarea celui inițial prin

înlocuirea muchiilor (1, 5) și (2, 3) cu muchiile (1, 3) și (2, 5). Avantajul ciclului nou creat este $\delta = 10 + 2 - 2 - 3 = 7 > 0$, rezultând că s-a produs o îmbunătățire.

Fie *calcul_avantaj* funcția care determină avantajul aplicării unui 2-schimb asupra unui ciclu. Îi sunt date ca parametri cele 2 muchii pentru care se dorește să se calculeze avantajul care se obține dacă s-ar aplica 2-schimbul peste f . Scopul acestei metode este să se verifice dacă este necesar să se creeze noul ciclu g .

Pseudocod calcul_avantaj((x, y), (x', y'))

1. calculează $\delta \leftarrow d(x, y) + d(x', y') - d(x, x') - d(y, y')$
2. returnează δ

Fie *2Schimburi* funcția care determină un ciclu g vecin cu f . Îi sunt date ca parametri ciclul f de la care se pornește și cele 2 muchii pentru care se dorește modificarea.

Pseudocod 2Schimburi(f, (x, y), (x', y'))

1. înlătură din f muchiile (x, y) și (x', y') care nu au niciun nod comun
2. adaugă în f muchiile (x, x') și (y, y') , pereche de muchii care duce la crearea unui nou ciclu g
3. returnează noul ciclu g

Din algoritmul *NN* și metodele *calcul_avantaj* și *2Schimburi* se creează algoritmul dorit *2-OPT* care are ca scop determinarea ciclului de lungime cât mai mică prin modificarea ciclului inițial.

Pseudocod 2-OPT

1. $f, soluție \leftarrow NN(D)$
2. repetă
3. $\delta \leftarrow 0$
4. $g \leftarrow f$
5. pentru fiecare m_1, m_2 din f , muchii care nu au niciun nod comun execută
6. $\delta_g \leftarrow calcul_avantaj(m_1, m_2)$
7. dacă $\delta_g > \delta$ atunci

8. $g \leftarrow 2Schimburi(f, m_1, m_2)$
9. $\delta \leftarrow \delta_g$
10. $f \leftarrow g$
11. $soluție \leftarrow soluție - \delta$
12. până când $\delta = 0$
13. returnează $soluție$

Algoritmul alege ciclul g care are avantajul cel mai mare adică cel care are lungimea ciclului cea mai mică.

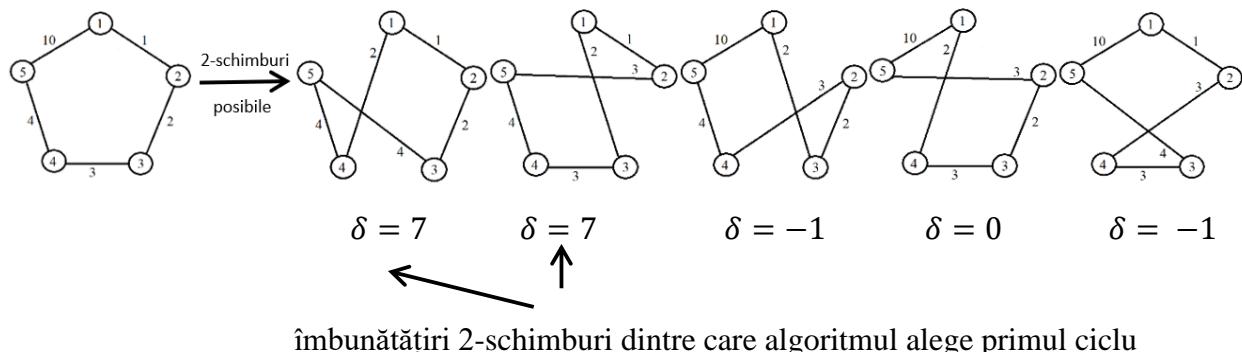


Figura 4.9. Prezentarea tuturor ciclurilor realizate prin 2-schimburi la primul pas al optimizării ciclului. După ce se creează un ciclu, se calculează avantajul adus, acesta fiind criteriu de alegere a ciclului cu care se merge la pasul următor.

La următoarea iterație a structurii repetitive cu test final, se obțin cele 5 cicluri de mai jos. Se observă că nu există niciun $\delta > 0$ ceea ce înseamnă că nu se mai pot crea optimizări, ciclul f găsit fiind cel final.

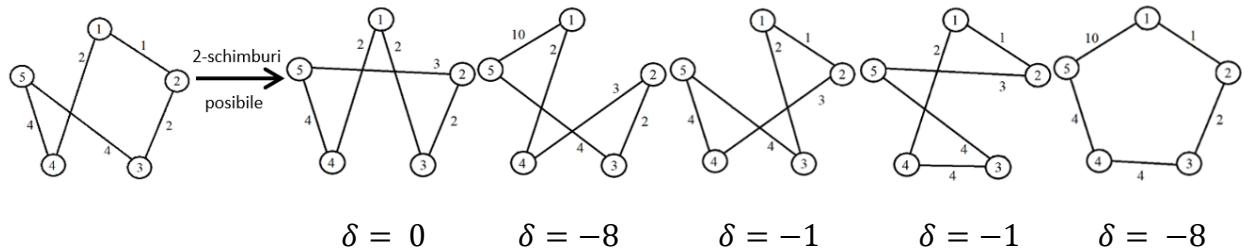


Figura 4.10. Prezentarea tuturor ciclurilor realizate prin 2-schimburi la al doilea pas al optimizării ciclului. Aceasta este și pasul final deoarece nu se creează niciun ciclu care să aducă avantaj ciclului curent.

Pentru a ajunge la o soluție cât mai bună, *2-OPT* este apelată de *repetări* ori cu scopul de a ajunge la soluția optimă prin optimizarea a mai multor cicluri obținute prin metoda *NN*. Prin urmare, soluția minimă din aceste *repetări* rulări ale algoritmului este considerată soluția optimă.

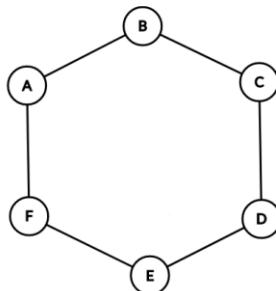
1. $soluție_minimă \leftarrow \infty$
2. pentru $i \leftarrow 1$, *repetări* execută
3. $soluție \leftarrow 2\text{-OPT}$
4. dacă $soluție < soluție_minimă$ atunci
5. $soluție_minimă \leftarrow soluție$

Codul corespunzător algoritmului 2-OPT este prezentat în [27] în limbajul Python.

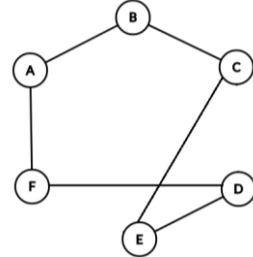
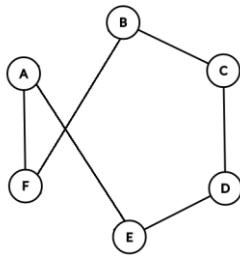
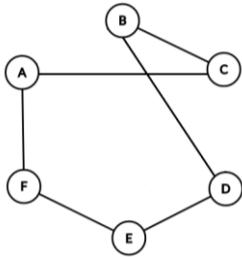
4.2.2. Algoritmul 3-OPT

Algoritmul 3-OPT are aceeași structură de rezolvare, diferența față de algoritmul 2-OPT este funcția care calculează avantajul realizării unui schimb, fiind adaptată pentru noul tip de înlocuire, și funcția în care se realizează cicluri prin 2-schimburi, acum prin 3-schimburi. Conceptul de înlocuire rămâne același: un 3-schimb înseamnă că se aplică 2-schimb pe un ciclu de unul sau mai multe ori pentru că sunt la dispoziție 3 muchii. Așadar dintr-un ciclu se creează 7 alte cicluri și dintre acestea se alege cel care are avantajul maxim.

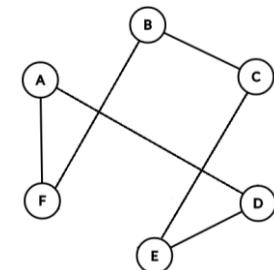
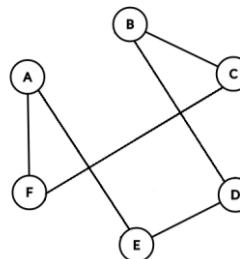
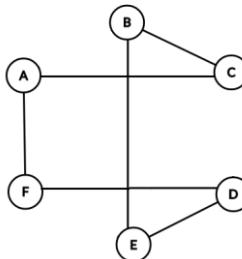
Cele 7 cicluri se creează astfel: 3 cicluri prin aplicarea 2-schimburi pe câte 2 din cele 3 muchii, 3 cicluri prin aplicarea repetată de 2 ori a două perechi de muchii și un ciclu prin aplicarea tuturor combinațiilor posibile din cele 3 muchii. Reprezentarea acestora este realizată în *Figura 4.11*.



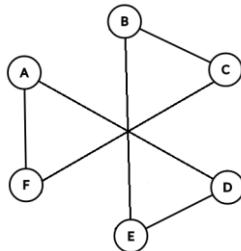
Ciclu inițial



Cicluri obținute prin aplicarea unui 2-schimb



Cicluri obținute prin aplicarea a două 2-schimburi



Ciclu obținut prin aplicarea a trei 2-schimburi

Figura 4.11. Cele 7 cicluri create printr-un 3-schimb între 3 muchii AB, CD și EF.

Exemplu

Fie f ciclul inițial de formă $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$. Se consideră că nodul A se află pe poziția i , nodul C pe poziția j și nodul E pe poziția k și se prezintă modul în care se realizează un 3-schimb prin aplicarea a două 2-schimburi.

Inițial se schimbă muchiile reținute pe poziții $(i, i + 1)$ și $(j, j + 1)$, iar apoi a muchiilor de pe pozițiile $(i, i + 1)$ și $(k, k + 1)$. Acești pași sunt reprezentați în Figura 4.12.

Se apelează funcția $2Schimburi$ cu parametri f , AB, CD și se obține ciclul g de forma $A \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow F$. După primul schimb pe pozițiile $(i, i + 1)$ se află muchia AC, pe pozițiile $(j, j + 1)$ muchia BD și pe $(k, k + 1)$ EF.

Apoi se apelează funcția $2Schimburi$ cu parametri g , AC, EF și se obține ciclul dorit h care arată astfel $A \rightarrow E \rightarrow D \rightarrow B \rightarrow C \rightarrow F$.

Plecându-se de la ciclul f și ajungând la $A \rightarrow E \rightarrow D \rightarrow B \rightarrow C \rightarrow F$, avantajul δ este $d(AB) + d(CD) + d(EF) - d(AE) - d(BD) - d(CF)$, ținând cont că pozițiile nodurilor din formulă sunt cele inițiale.

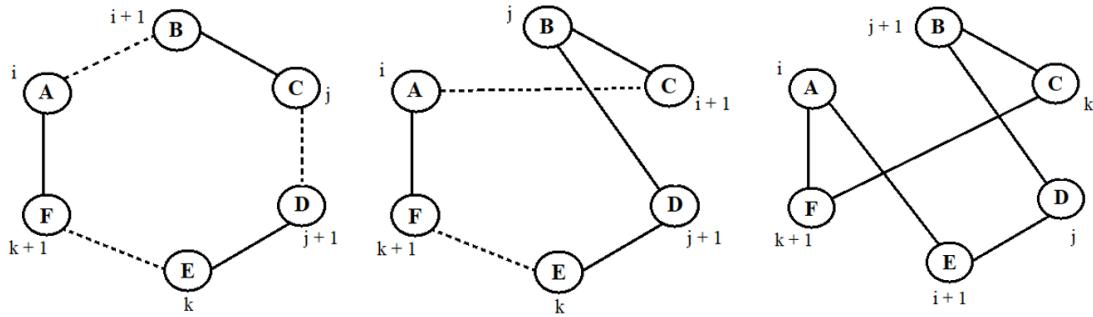


Figura 4.12. Realizarea unui 3-schimb prin două 2-schimburile. În Figura 4.12.a. este prezentat ciclul inițial și pozițiile muchiilor pentru care se dorește înlocuirea. În Figura 4.12.b. este afișat pasul intermediar în care se creează primul schimb, iar în Figura 4.12.c. este prezentat pasul final după ce se realizează și a doua schimbare.

Se creează funcția *calcul_avantaj* în analogie cu *Figura 4.11*, determinând avantajul fiecarui ciclu:

Pseudocod calcul_avantaj(AB, CD, EF)

1. $\delta_{A \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow F} \leftarrow d(AB) + d(CD) - d(AC) - d(BD)$
2. $\delta_{A \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow F} \leftarrow d(AB) + d(EF) - d(AE) - d(BF)$
3. $\delta_{A \rightarrow B \rightarrow C \rightarrow E \rightarrow D \rightarrow F} \leftarrow d(CD) + d(EF) - d(CE) - d(DF)$
4. $\delta_{A \rightarrow C \rightarrow B \rightarrow E \rightarrow D \rightarrow F} \leftarrow d(AB) + d(CD) + d(EF) - d(AC) - d(BE) - d(DF)$
5. $\delta_{A \rightarrow E \rightarrow D \rightarrow B \rightarrow C \rightarrow F} \leftarrow d(AB) + d(CD) + d(EF) - d(AE) - d(DB) - d(CF)$
6. $\delta_{A \rightarrow D \rightarrow E \rightarrow C \rightarrow B \rightarrow F} \leftarrow d(AB) + d(CD) + d(EF) - d(AD) - d(EC) - d(BF)$
7. $\delta_{A \rightarrow D \rightarrow E \rightarrow B \rightarrow C \rightarrow F} \leftarrow d(AB) + d(CD) + d(EF) - d(AD) - d(EB) - d(CF)$
8. se alege avantajul maxim și se notează cu δ
9. returnează δ și un identificator unic prin care să se poată recunoaște cărui schimb îi corespunde avantajul

Se consideră identificator ordinea nodurilor în ciclul rezultat. Prin urmare, funcția *3Schimburile* arată astfel:

Pseudocod 3Schimburi(f, AB, CD, EF, identifier)

1. dacă *identifier* = ‘ACBDEF’ atunci
2. $g \leftarrow 2\text{Schimburi}(f, AB, CD)$
3. dacă *identifier* = ‘AEDCBF’ atunci
4. $g \leftarrow 2\text{Schimburi}(f, AB, EF)$
5. dacă *identifier* = ‘AEDCBF’ atunci
6. $g \leftarrow 2\text{Schimburi}(f, CD, EF)$
7. dacă *identifier* = ‘ACBEDF’ atunci
8. $g_{aux} \leftarrow 2\text{Schimburi}(f, AB, CD)$
9. $g \leftarrow 2\text{Schimburi}(g_{aux}, AB, EF)$
10. dacă *identifier* = ‘AEDBCF’ atunci
11. $g_{aux} \leftarrow 2\text{Schimburi}(f, AB, CD)$
12. $g \leftarrow 2\text{Schimburi}(g_{aux}, CD, EF)$
13. dacă *identifier* = ‘ADECBF’ atunci
14. $g_{aux} \leftarrow 2\text{Schimburi}(f, AB, EF)$
15. $g \leftarrow 2\text{Schimburi}(g_{aux}, CD, EF)$
16. dacă *identifier* = ‘ADEBCF’ atunci
17. $g_1 \leftarrow 2\text{Schimburi}(f, AB, CD)$
18. $g_2 \leftarrow 2\text{Schimburi}(g_1, AB, EF)$
19. $g \leftarrow 2\text{Schimburi}(g_2, CD, EF)$
20. returnează ciclu g

În primele trei condiții din pseudocod se calculează cele trei 2-schimburi (liniile 1-6), liniile 7 – 15 cele trei cicluri obținute prin aplicarea a două 2-schimburi care se realizează în 2 etape și la liniile 16 - 19 ciclul construit prin trei 2-schimburi. Identifierul decide pe ce condiție se va intra creându-se un singur ciclu care la final se returnează.

Pseudocod 3-OPT

1. $f, soluție \leftarrow NN(D)$
2. repetă
3. $\delta \leftarrow 0$

```

4.       $g \leftarrow f$ 
5.      pentru fiecare  $m_1, m_2, m_3$  din  $f$ , muchii care nu au niciun nod comun execută
6.           $\delta_g, identifier \leftarrow calcul\_avantaj(m_1, m_2, m_3)$ 
7.          dacă  $\delta_g > \delta$  atunci
8.               $g \leftarrow 3Schimburi(f, m_1, m_2, m_3, identifier)$ 
9.               $\delta \leftarrow \delta_g$ 
10.          $f \leftarrow g$ 
11.          $soluție \leftarrow soluție - \delta$ 
12.         până când  $\delta = 0$ 
13.         returnează  $soluție$ 

```

Având în vedere că și algoritmul *3-OPT*, la fel ca algoritmul *2-OPT*, presupune micșorarea soluției unui ciclu determinat de un algoritm aproximativ a cărui nod de start este ales aleator, este posibil ca *3-OPT* să nu ofere o soluție bună din cauza aceluia ciclu inițial. Pentru a avea mai multe șanse, se poate rula de mai multe ori algoritmul și să se păstreze ciclul cu soluția cea mai mică.

```

1.       $soluție\_minimă \leftarrow \infty$ 
2.      pentru  $i \leftarrow 1$ , repetări execută
3.           $soluție \leftarrow 3-OPT$ 
4.          dacă  $soluție < soluție\_minimă$  atunci
5.               $soluție\_minimă \leftarrow soluție$ 

```

Codul acestui algoritm implementat în Python este atașat în [27] și este construit pe baza explicațiilor și pseudocodului anterior.

5. Comparații între algoritmi

Scopul acestui capitol este de a prezenta eficiența soluției și timpul de rulare ai algoritmilor prezentăți de-a lungul lucrării.

Graficele sunt realizate folosind biblioteca *matplotlib.pyplot* și au ca scop vizualizarea datele din tabele pentru a observa mai ușor calitatea soluțiilor și rapiditatea algoritmilor. Pe axa x sunt numărul de noduri al fiecărui set de date (pentru algoritmii exacti se adaugă pe axa x și numărul de noduri), iar pe axa y apare raportul dintre soluția propriu-zisă și cea optimă sau timpul de rulare.

Instanțele sunt preluate din biblioteca online *TSPLIB*, bibliotecă ce conține numeroase instanțe-exemplu de diferite tipuri pentru problema comis-voiajorului. S-au ales seturi a căror date sunt de două tipuri.

Primul tip folosește distanțe euclidiene 2D. Fișiere sunt de forma *număr nod, coordonata X, coordonata Y*, iar coordonatele pot fi atât numere naturale, cât și reale pozitive. Folosind formula din *Subcapitolul 1.2.2.* unde este prezentat cazul special de TSP metric numit TSP euclidian, se determină distanțele dintre oricare 2 noduri și sunt puse într-o matrice de distanțe simetrică și apoi sunt preluate de algoritmi.

Pentru o comparație amplă, s-au extras mai multe instanțe cu număr diferit de noduri *eil51.tsp, st70.tsp, lin105.tsp, ch150.tsp, kroA200.tsp, pr299.tsp, rd400.tsp, pcb442.tsp, rat575.tsp, rat783.tsp, pr1002.tsp, rl1323.tsp, pr2392.tsp, rl5915.tsp și pla7397.tsp*. Aceste seturile sunt folosite pentru a compara algoritmii aproximativi și euristici.

Al doilea tip de instanțe de pe TSPLIB este cel explicit în care sunt date valorile din matricea de distanțe de sub prima diagonală (*EDGE_WEIGHT_FORMAT*:

LOWER_DIAG_ROW). Preluarea datelor este simplistă: se introduc în matrice valorile știind că la final trebuie să se creeze o matrice simetrică. Instanțele folosite sunt *gr24.tsp* și *fri26.tsp*. Acestea sunt utilizate pentru comparațiile dintre algoritmii exacti care sunt mai lenți.

În plus bibliotecii TSPLIB, pentru algoritmii exacti sunt folosite și instanțe de pe *infoarena*: *grader_test1.in*, *grader_test4.in*, *grader_test6.in*, *grader_test11.in*, *grader_test16.in*, *grader_test19.in* și *grader_test20.in*. Acestea au un alt format, pe prima linie apar două numere, primul reprezentând numărul de noduri din graf, iar al doilea numărul de muchii. Important de precizat este că grafurile nu sunt complete și nici simetrice, motiv pentru care aceste instanțe nu sunt potrivite pentru algoritmii ce se bazează pe condiția că instanța trebuie să respectă inegalitatea triunghiului.

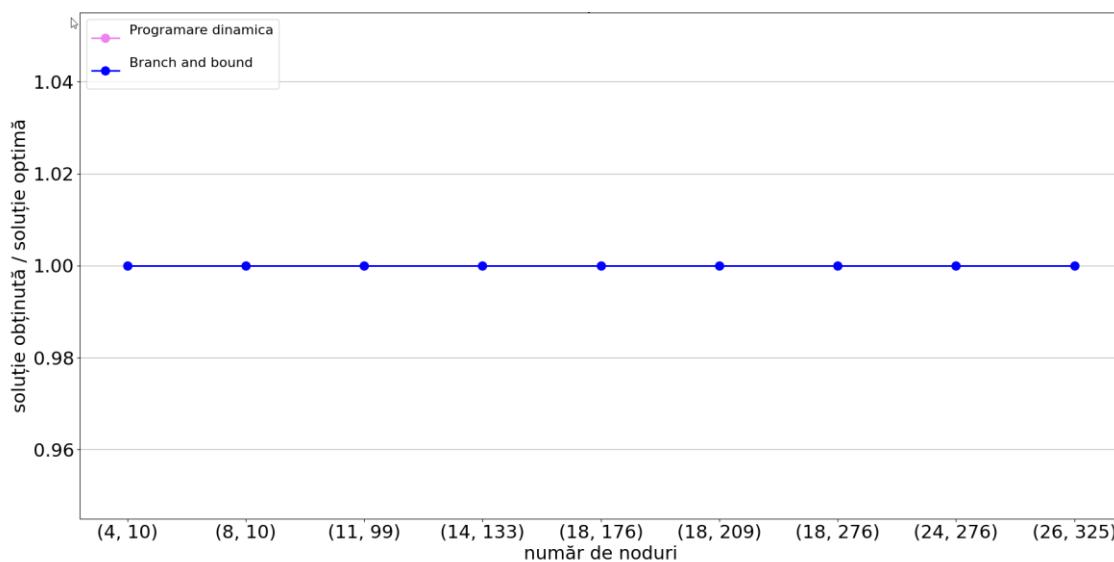
5.1. Algoritmi exacti

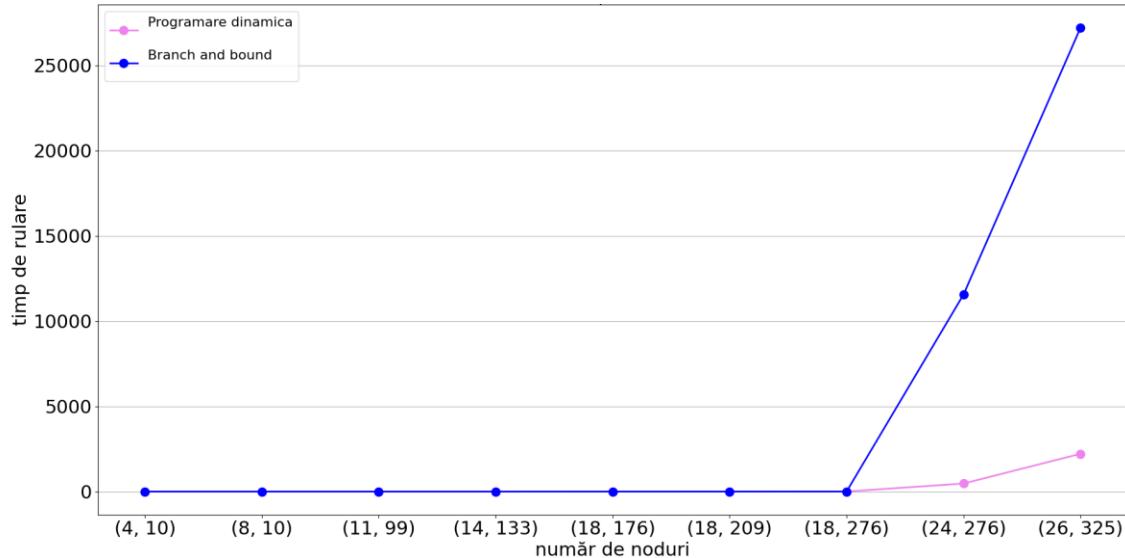
Algoritmii exacti sunt aceia care returnează soluția optimă indiferent de timpul de rulare. Prin urmare, din cauza timpului exponențial, acești algoritmi nu se pot folosi pe instanțe cu foarte multe noduri (de exemplu pe instanțele alese de pe *TSPLIB* pentru compararea algoritmilor aproximativi și euristică). Însă buna lor funcționare se observă prin rularea seturilor de date de pe *infoarena* și pe cele 2 cu 24 și 26 de noduri de pe *TSPLIB* (*Tabelul 5.1.* și *Graficul 5.1.a.*).

Algoritmul Branch and Bound este lent pentru că trebuie să parcurgă arborele și pentru a ajunge să elimine subarbori, trebuie să găsească o soluție cu scopul de a seta o limită superioară a soluției optime. O optimizare menționată și în *Subcapitolul 2.2.* este să obținem de la început o valoare pentru limită superioară. Această lucru este posibil pentru cazul în care algoritmul primește ca input un graf complet și simetric prin aplicarea unui algoritm heuristic, care va oferi o soluție aproximativă, aflată în aria factorului său de aproximare. Se alege algoritmul de inserție a celui mai apropiat nod cu 5 repetări (discutat în *Subcapitolul 4.2.*) care este 2-aproximativ. Valoarea rezultată este setată ca limită superioară inițială pentru algoritmul Branch and Bound.

Instanță				Programare dinamică		Branch and bound	
nume	noduri	muchii	OPT	soluție	timp	soluție	timp
grader_test1	4	10	2166782	2166782.0	0.0	2166782.0	0.002
grader_test4	8	10	2844250	2844250.0	0.001	2844250.0	0.011
grader_test6	11	99	1965875	1965875.0	0.013	1965875.0	0.017
grader_test11	14	133	3336377	3336377.0	0.171	3336377.0	0.0795
grader_test16	18	176	2682065	2682065.0	4.326	2682065.0	0.519
grader_test19	18	209	2516395	2516395.0	4.223	2516395.0	1.432
grader_test20	18	276	2171518	2171518.0	4.33	2171518.0	3.352
gr24	24	276	1272	1272.0	471.889	1272.0	11557.7467
fri26	26	325	937	937.0	2216.1889	937.0	27195.2738

Tabel. 5.1. Datele de ieșire obținute de algoritmii exacti care folosesc programare dinamică și branch and bound pe instanțele de pe infoarena.





Grafic 5.1. Reprezentarea vizuală a raportului soluție obținută – soluție optimă și a timpului de rulare a algoritmilor în funcție de numărul de noduri și de muchii ale instanțelor. Pentru algoritmii exacți $y = 1$ când vine vorba de raport.

Chiar și cu optimizarea aplicată, creșterea numărului de noduri și de muchii are un efect mai mare asupra algoritmului Branch and Bound decât a celui ce utilizează programare dinamică în ceea ce privește timpul de rulare. Se observă în *Graficul 5.1.b.* că timpul de execuție crește semnificativ când se ajunge la instanța *gr24* care corespunde unui graf complet. Diferența este de peste 11.000 secunde ≈ 3.05 ore și continuă să se mărească în cazul instanței următoare (peste 24.500 secunde ≈ 7.08 ore) ceea ce face ca algoritmul prim menționat să fie evitat în favoarea celui ce folosește programare dinamică.

5.2. Algoritmi aproximativi

Algoritmii aproximativi sunt aceeaia a căror soluție este încadrată într-un factor de aproximare față de soluția optimă (OPT). Amintind din *Capitolul 3* factorii de aproximare ai algoritmilor discutați:

- algoritmul arborelui dublu este 2-aproximativ
- algoritmul lui Christofides pentru care cuplarea nodurilor de grad impar se face pe baza algoritmul de determinare a cuplajului de cost minim este $\frac{3}{2}$ -aproximativ

- algoritmul lui Christofides pentru care cuplarea nodurilor de grad impar se face folosind o strategie greedy este imprevizibil

Pentru prezentarea eficienței algoritmilor se iau mai multe instanțe cu numărul de noduri diferit, cuprins între 51 noduri (minim) și 7397 noduri (maxim). În *Tabelul 5.2.* sunt afișate următoarele valori: numele instanței de pe *TSPLIB*, numărul de noduri, soluția optimă (OPT), soluția returnată de algoritmi și timpul de execuție a programelor. Se amintește că toți cei 3 algoritmi pornesc dintr-un nod de start ales în mod aleator care este folosit ca prim nod în algoritmul lui Prim și în funcția de determinare a ciclului hamiltonian. Așadar oricare două rulări ale aceluiași algoritm poate duce la două soluții distincte.

Pentru algoritmul lui Christofides care utilizează strategia greedy mai există o coloană *repetări greedy* care semnifică de câte ori a fost repetat algoritmul de căutare a cuplajului pentru a ajunge la acela care s-a adăugat arborelui și în mod direct a influențat valoarea soluției. S-au ales pentru numărul de repetări valorile 100, 300 și 500 și se va urmări și pentru aceste 3 cazuri cum se comportă algoritmul.

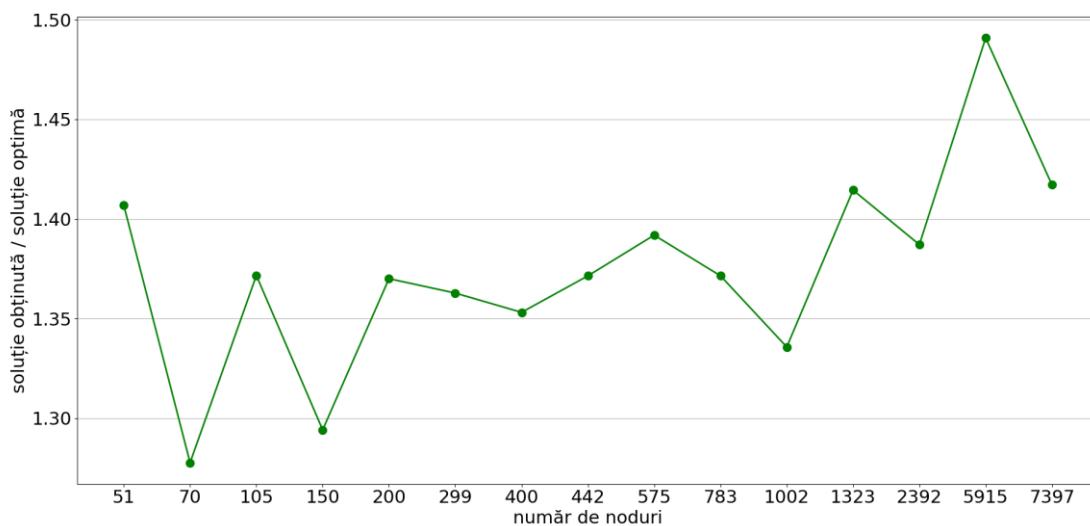
Instanță			Algoritmul arborelui dublu		Algoritmul lui Christofides folosind NetworkX		Algoritmul lui Christofides în cuplajul este realizat folosind o strategie greedy		
nume	noduri	OPT	soluție	timp	soluție	timp	soluție	timp	repetări greedy
eil51	51	426	599.3412	0.003	510.3363	0.01	495.3128	0.008	100
							512.8048	0.014	300
							524.8261	0.02	500
st70	70	675	862.4324	0.007	821.9486	0.033	835.636	0.013	20
							850.3106	0.024	300
							818.1597	0.036	500
lin105	105	14379	19723.3146	0.013	19649.3781	0.059	19602.2127	0.027	100
							19104.0615	0.04	300
							19409.8272	0.058	500
ch150	150	6528	8448.2644	0.027	7828.4241	0.094	7956.7989	0.037	100
							7998.2469	0.059	300
							7890.8512	0.083	500

kroA200	200	29368	40235.8742	0.044	35857.9237	0.357	36284.4624	0.075	100
							36231.2407	0.132	300
							36353.3166	0.187	500
pr299	299	48191	65675.8927	0.103	58946.1205	0.785	60595.5668	0.158	100
							61233.863	0.262	300
							60869.7781	0.334	500
rd400	400	15281	20677.5432	0.177	18531.0943	3.0409	19364.9793	0.258	100
							19365.0049	0.465	300
							19416.2942	0.638	500
pcb442	442	50778	69639.9937	0.21	62835.9889	2.821	65575.8182	0.321	100
							65504.7303	0.531	300
							64440.7244	0.725	500
rat575	575	6773	9427.3044	0.367	8305.8577	6.2811	8788.2248	0.523	100
							8667.6754	0.896	300
							8701.6105	1.209	500
rat783	783	8806	12077.412	0.646	10861.4532	15.9234	11340.3595	0.953	100
							11440.6579	1.699	300
							11364.7262	2.202	500
pr1002	1002	259045	346020.5302	1.027	309290.2367	23.4309	330840.885	1.693	100
							329565.9821	2.642	300
							330296.5174	3.5941	500
rl1323	1323	270199	382198.2234	1.7771	317183.4798	8.654	328168.3938	2.081	100
							331594.5974	2.377	300
							333308.3154	2.701	500
pr2392	2392	378032	524385.2662	5.747	465684.6573	245.111 8	508280.5864	8.4141	100
							499911.5945	14.2197	300
							502157.3341	19.197	500
rl5915	5915	565530	843138.1298	35.012 9	647474.7717	261.191	689692.4943	37.221	100
							689965.9214	42.4693	300
							687021.4495	47.0969	500
pla7397	7397	23260728	32964004.122 8	55.667 2	28582869.69 42	5462.59 52	31160608.4569	76.2858	100
							31162164.6414	120.5676	300
							30992700.7354	155.2564	500

Tabel 5.2. Datele de ieșire rezultate în urma rulării algoritmului arborelui dublu, algoritmului

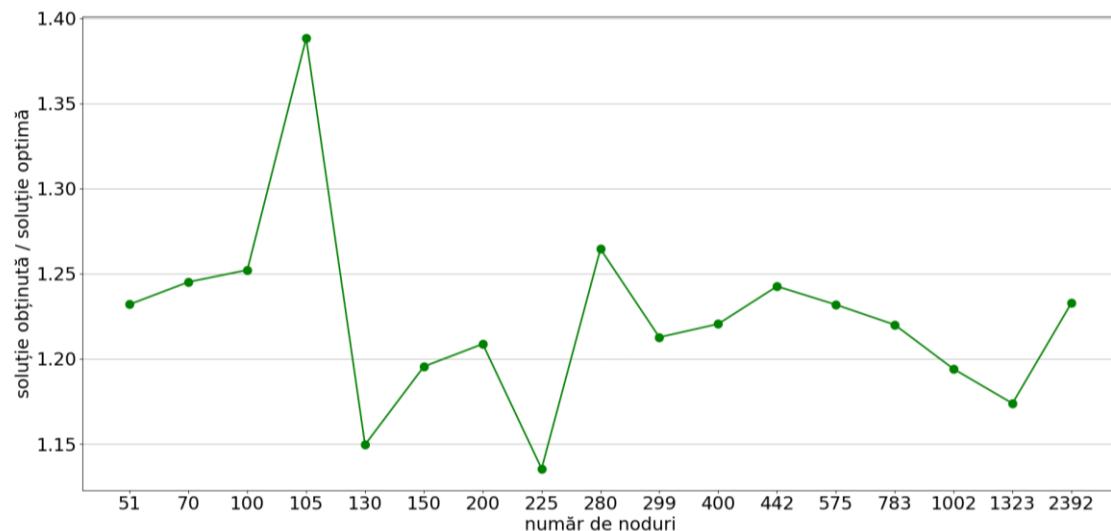
lui Christofides varianta $\frac{3}{2}$ aproximativ și cea în care cuplajul se determină în mod greedy.

Algoritmul arborelui dublu

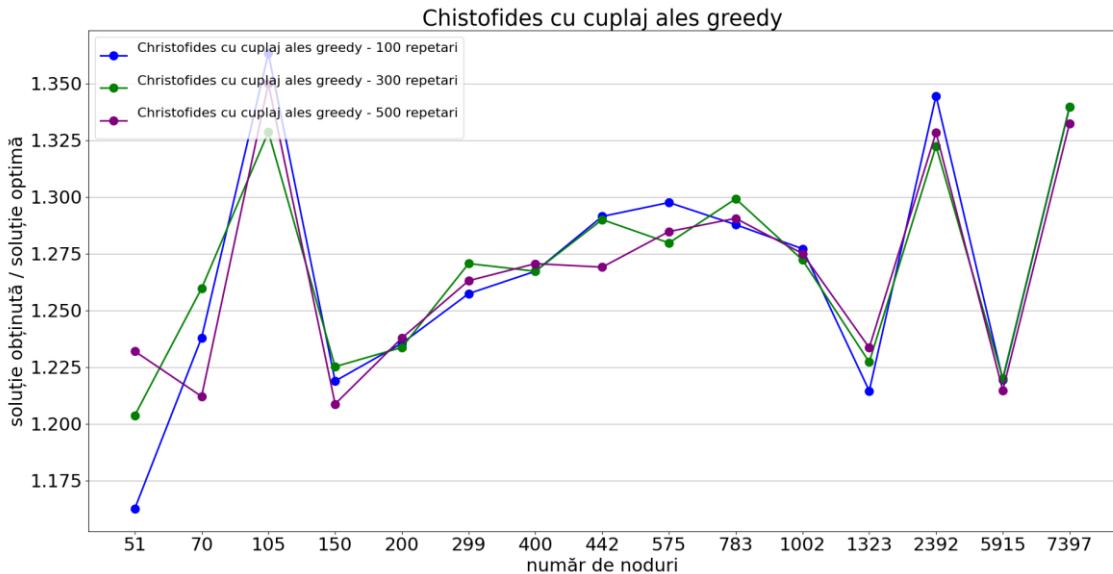


Grafic 5.2. Prezentarea vizuală a raportului soluție obținută – soluție optimă pentru algoritmul arborelui dublu.

Christofides cu NetworkX



Grafic 5.3. Prezentarea vizuală a raportului soluție obținută – soluție optimă pentru algoritmul lui Christofides.



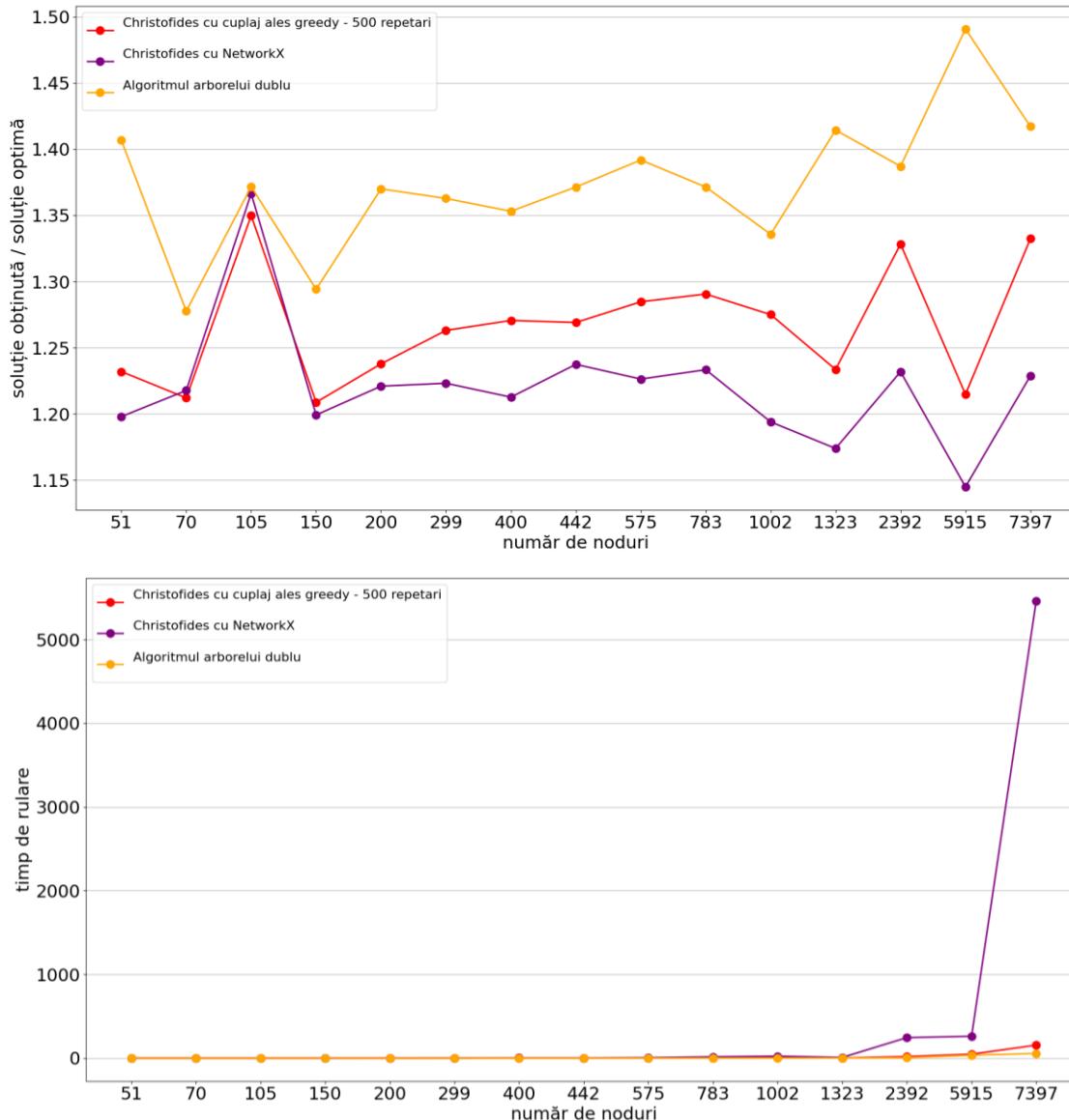
Grafic 5.4. Prezentarea vizuală a raportului soluție obținută – soluție optimă pentru algoritmul lui Christofides în care cuplajul se realizează greedy pentru care funcția în care se determină muchiile cuplajului perfect se repetă de 100, 300 și 500 ori.

Algoritmul arborelui dublu obține soluții bune, chiar dacă este 2-aproximativ. Se observă în *Tabelul 5.2.* și în *Graficul 5.2.* că soluțiile sunt mai mici de $1.5 * \text{OPT}$ și timpul de rulare este relativ mic, luând în calcul că ultima instanță *pla7397* conține 7397 de noduri, iar algoritmul reușește să dea un răspuns în 55.6672 secunde, adică mai puțin de 1 minut.

În comparație cu algoritmul arborelui dublu apare algoritmul lui Christofides care este mult mai lent din cauza algoritmului de determinare a cuplajului perfect de cost minim care durează $O(n^3)$ unde n este numărul de noduri de grad impar. Acest algoritm durează mult până ajunge să ofere o soluție pentru instanța *pla7397*, timpul fiind de 5462.5952 secunde ≈ 1.517 ore (vezi *Tabel 5.2.*). Dacă nu se ia în calcul timpul de execuție, algoritmul oferă soluții bune, după cum ne așteptam având în vedere că are un factor de aproximare de $\frac{3}{2}$ (*Grafic 5.3.*).

Varianta algoritmului în care cuplajul este realizat folosind o metoda greedy oferă soluții cu raport asemănătoare pentru cele 3 rulări cu număr diferit de repetări, diferența fiind sub 0.10 pentru orice instanță, cea maximă observându-se pentru cazul cu cele mai puține noduri (51). Prin creșterea lor se stabilizează rezultând o diferență mai mică de maxim 0.05 și ajunge până la punctul în care să se suprapună pe grafic (*Graficul 5.4.*). Cu toate acestea, pe alocuri, numărul de

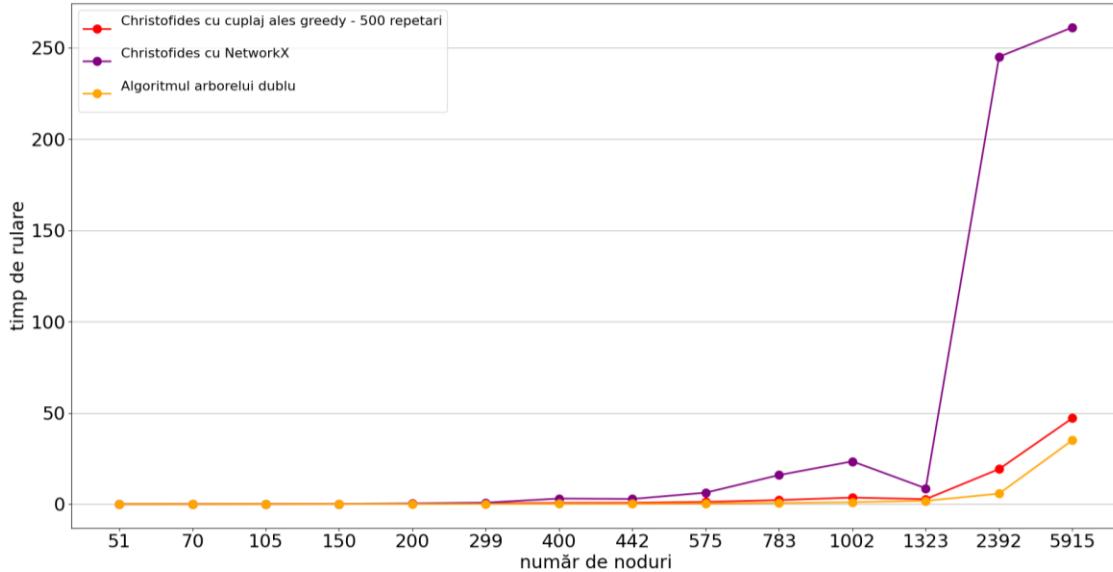
repetări mai mari duce la o soluție puțin mai bună, motiv pentru care pentru compararea celor 3 algoritmi se va folosi varianta cu 500 repetări:



Grafic 5.5. Prezentarea împreună a rezultatelor celor 3 algoritmi: algoritmul arborelui dublu, algoritmul lui Christofides și algoritmul lui Christofides în care cuplajul este realizat folosind o metodă greedy (500 repetări).

Expunerea timpurilor de execuție împreună accentuează diferența de peste 4.000 secunde (adică peste o oră) dintre algoritmul lui Christofides și ceilalți doi algoritmi (Graficul 5.5.b.) întâlnită pentru cazul cu 7397 noduri. Pentru a vizualiza mai bine și celealte diferențe de timp,

este nevoie de un grafic pentru care pe axa y să fie valori mai apropiate. Din acest motiv se atașează un alt grafic în care excludem instanța *pla7397*:



Grafic 5.6. Același ca Graficul 5.5.b. din care s-a eliminat verificarea instanței cu peste 7000 de noduri.

Din *Graficul 5.6.* se observă mai bine diferența de timp. Algoritmul lui Christofides continuă să dureze relativ mult comparativ cu ceilalți algoritmi, aproximativ 260 secunde (4.33 minute) pentru instanța *rl5915*, pe când ceilalți 2 au reușit să ofere răspunsul în mai puțin de 1 minut. Pentru instanța *pr2392* din nou diferența este mare (circa 240 secunde \approx 4 minute, în comparație cu sub 25 secunde). Prin urmare, când se ia în calcul timpul de rulare, nu se merită folosirea algoritmului lui Christofides pentru instanțe care au mai mult \approx 1400 noduri pentru că durata executării crește semnificativ.

Pe de altă parte, algoritmul arborelui dublu este rapid și pe cazuri cu foarte multe noduri, dar este scăzută performanța soluțiilor, aceasta nemaifiind la fel de apropiată de OPT comparativ cu cea a celorlalți 2 algoritmi chiar dacă se observă că se încadrează sub $1.5 * \text{OPT}$ (*Grafic 5.5.a.*).

Așadar, pentru instanțe cu număr de noduri < 1400 , este potrivit să se folosească ca raport calitate - durată de execuție algoritmul lui Christofides. Iar pentru instanțe mai mari consider că o varianta bună de folosit este algoritmul lui Christofides în care cuplarea nodurilor de grad impar se face greedy pentru că oferă un răspuns în mai puțin de 160 secunde pentru *pla7397* (când

cealaltă variantă mai lentă trece de o oră), iar calitatea soluțiilor este într-un interval de 1.35 * OPT pentru instanțe mari (*Grafic 5.5.a.*).

5.3. Algoritmi euristici

În *Capitolul 4* s-a discutat despre algoritmii 2-OPT și 3-OPT pentru care ciclul hamiltonian se poate determina folosind diferiți algoritmi euristici: algoritmul de inserție a celui mai îndepărtat nod (farthest insertion algorithm), algoritmul de inserție a celui mai apropiat nod (nearest insertion algorithm), algoritmul de inserție cu cel mai mic cost (cheapest insertion algorithm) și algoritmul cel mai apropiat vecin (nearest neighbor algorithm). Având în vedere că cele 4 euristici folosesc strategii greedy și nodul de start pentru construcția ciclului este ales aleator, rezultatele obținute nu sunt unele fixe, motiv pentru care la o altă rulare, acestea pot fi mai bune sau mai proaste decât cele prezentate în acest subcapitol.

Algoritmul 2-OPT este verificat în forma sa prezentată în *Capitolul 4* și în [27] schimbând doar valoarea variabilei *repetări* care specifică de câte ori a fost rulat pentru a obține soluția propriu-zisă. În acest subcapitol se vor folosi valorile 3 și 5, iar datele de ieșire obținute pot fi urmărite în *Tabelul 5.3.*, respectiv, *Tabelul 5.4.*

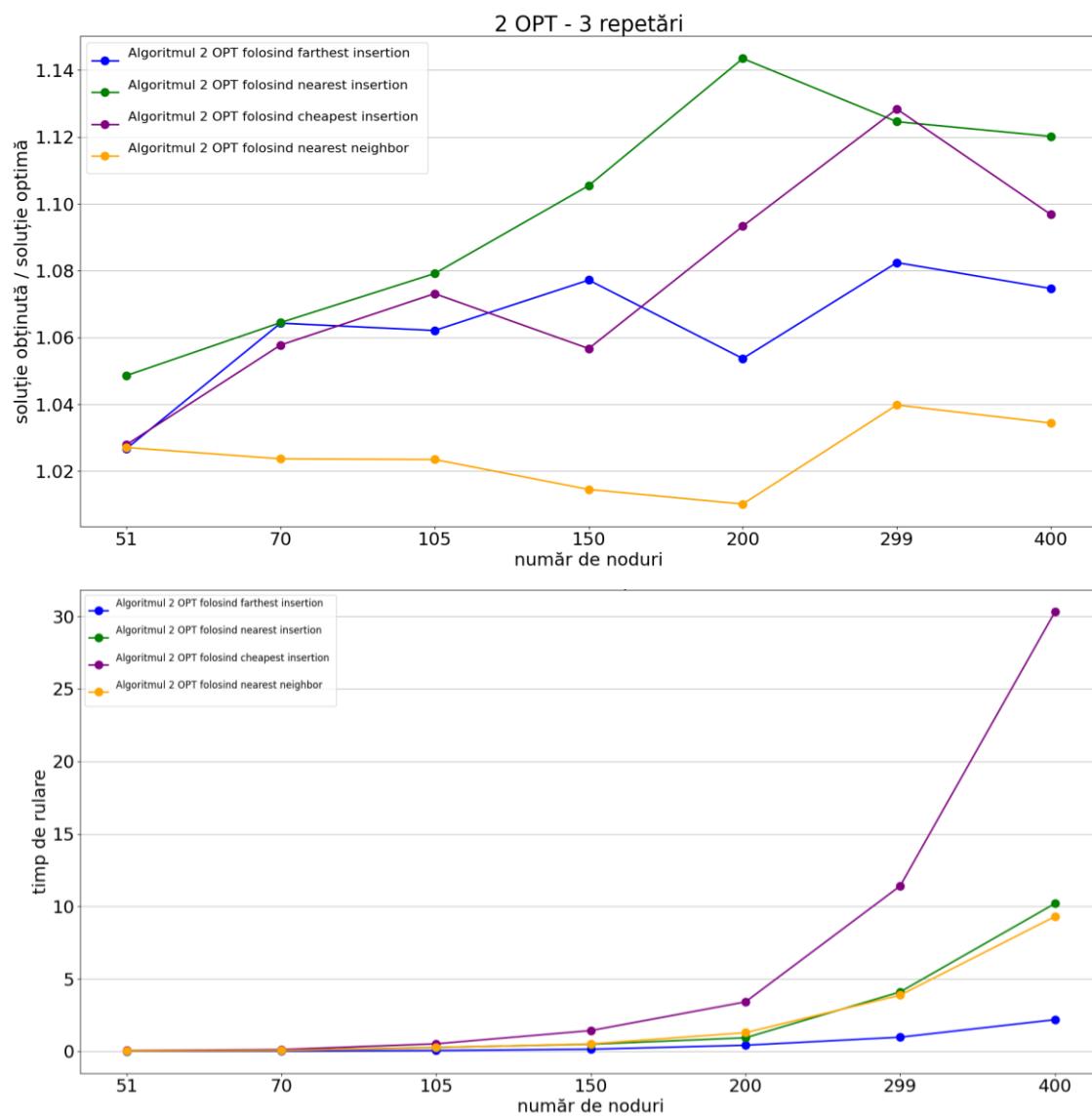
Algoritmul 3-OPT, indiferent de euristică folosită, este mai lent decât algoritmul 2-OPT pentru că are mai multe verificări de făcut și schimburile pot fi mai costisitoare (se ajunge la 3-schimuri comparativ cu algoritmul 2-OPT unde per iterație se face doar un singur 2-schimb). Din acest motiv, algoritmului 3-OPT i s-a permis rularea doar cu o singură repetare.

În tabelele următoare sunt prezentate rezultatele rulării celor doi algoritmi specificând pentru fiecare instanță folosită, numărul de noduri, soluția optimă preluată de pe site-ul *TSPLIB*, soluția obținută de algoritm și timpul de rulare.

			2 OPT – 3 repetări							
Instanță			Farthest insertion		Nearest insertion		Cheapest insertion		Nearest neighbor	
nume	noduri	OPT	soluție	timp	soluție	timp	soluție	timp	soluție	timp
eil51	51	426	437.3909	0.013	446.7084	0.027	437.893	0.058	437.5297	0.033
st70	70	675	718.3929	0.022	718.5129	0.075	713.9737	0.129	690.9967	0.075
lin105	105	14379	15271.8255	0.066	15517.0192	0.276	15430.5059	0.524	14716.6269	0.2616
ch150	150	6528	7031.8444	0.151	7216.4196	0.493	6898.0217	1.437	6622.8248	0.517

kroA200	200	29368	30945.2157	0.425	33581.9017	0.939	32108.1085	3.4166	29666.4024	1.291
pr299	299	48191	52162.4602	0.98	54194.9917	4.105	54377.4265	11.411	50108.799	3.881
rd400	400	15281	16421.8622	2.1854	17117.5742	10.1958	16760.0345	30.305	15806.7446	9.3002

Tabel 5.3. Datele de ieșire rezultate în urma rulării algoritmului 2-OPT cu 3 repetări împreună cu farthest insertion, nearest insertion, cheapest insertion și nearest neighbor. Date de intrare sunt luate de pe TSPLIB.



Grafic 5.6. Prezentarea vizuală a raportului soluție obținută – soluție optimă și a timpului de rulare per instanță, comparând astfel calitatea celor 4 algoritmi care determină ciclul hamiltonian care va fi îmbunătățit de algoritmul 2-OPT rulat de 3 ori.

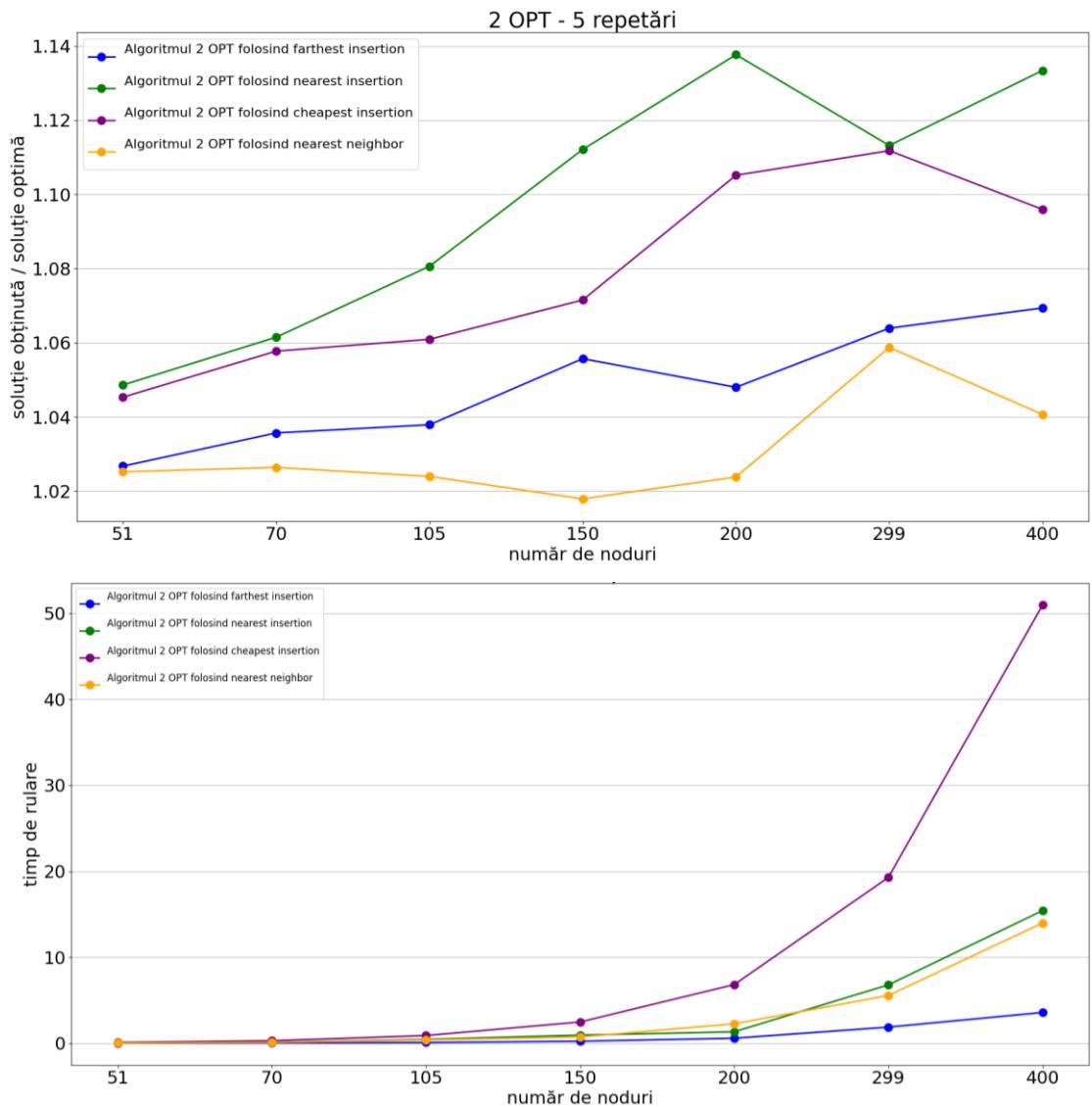
Privind la raportul soluție – timp de execuție din *Graficul 5.6.*, algoritmul nearest neighbor este cel care a determinat ciclul hamiltonian ce a pus bazele celei mai bune variante de 2-OPT pentru cazul în care algoritmul este repetat de 3 ori. Soluțiile pentru acesta sunt într-un factor de 1.04 de aproximare, iar timpul de rulare este de maxim 10 secunde.

Chiar dacă restul algoritmilor au performat mai prost, privind graficul se observă că rezultatele sunt mai mici de $1.15 * \text{OPT}$. De aceea, algoritmii sunt totuși buni, dar nu mai buni decât cel menționat anterior. În raport cu timpul de rulare (*Graficul 5.6.b.*), euristica cheapest insertion se îndepărtează de timpul celorlalți, pentru 400 de noduri ajungând la un timp de execuție de cca 31 secunde.

Dar având în vedere că încă sunt decenti ca timp de rulare, se stabilește un număr mai mare de iterații (*repetări* = 5) și se execută din nou aceeași instanță cu scopul de a vedea dacă prin creșterea numărului de repetări soluția devine mai apropiată de cea optimă. Experimentul arătat în *Tabelul 5.4.* și *Graficul 5.7.* arată că soluția cea mai mică continuă să fie dată de euristica nearest neighbor, dar timpul de execuție a crescut la 15 secunde pentru instanța care a rulat cel mai lent. Per total performanța algoritmilor a crescut, acum raportul rezultat - soluție optimă este puțin sub 1.14. Referitor la timp, acesta s-a mărit, dar nu semnificativ mai mult, decât la algoritmul cheapest insertion se observă o diferență mai mare. Pentru 400 noduri acum ajunge la 51 secunde.

			2 OPT – 5 repetări							
Instanță			Farthest insertion		Nearest insertion		Cheapest insertion		Nearest neighbor	
nume	noduri	OPT	soluție	timp	Soluție	timp	soluție	timp	soluție	timp
eil51	51	426	437.3909	0.016	446.7084	0.039	445.3099	0.112	436.7422	0.047
st70	70	675	699.0884	0.029	716.5133	0.111	713.9737	0.299	692.8246	0.108
lin105	105	14379	14923.8614	0.091	15538.0693	0.464	15255.0968	0.902	14723.6704	0.403
ch150	150	6528	6891.6498	0.241	7259.9259	0.952	6994.9842	2.476	6644.9871	0.736
kroA200	200	29368	30777.5906	0.589	33411.011	1.34	32456.3286	6.825	30067.1443	2.2642
pr299	299	48191	51272.038	1.881	53645.6085	6.7924	53577.711	19.2825	51022.4757	5.5542
rd400	400	15281	16341.234	3.581	17319.5756	15.425	16747.285	50.9503	15902.3418	13.9892

Tabel 5.4. Datele de ieșire rezultate în urma rulării algoritmului 2-OPT împreună cu farthest insertion, nearest insertion, cheapest insertion și nearest neighbor știind ca algoritmul este repetat de 5 ori.



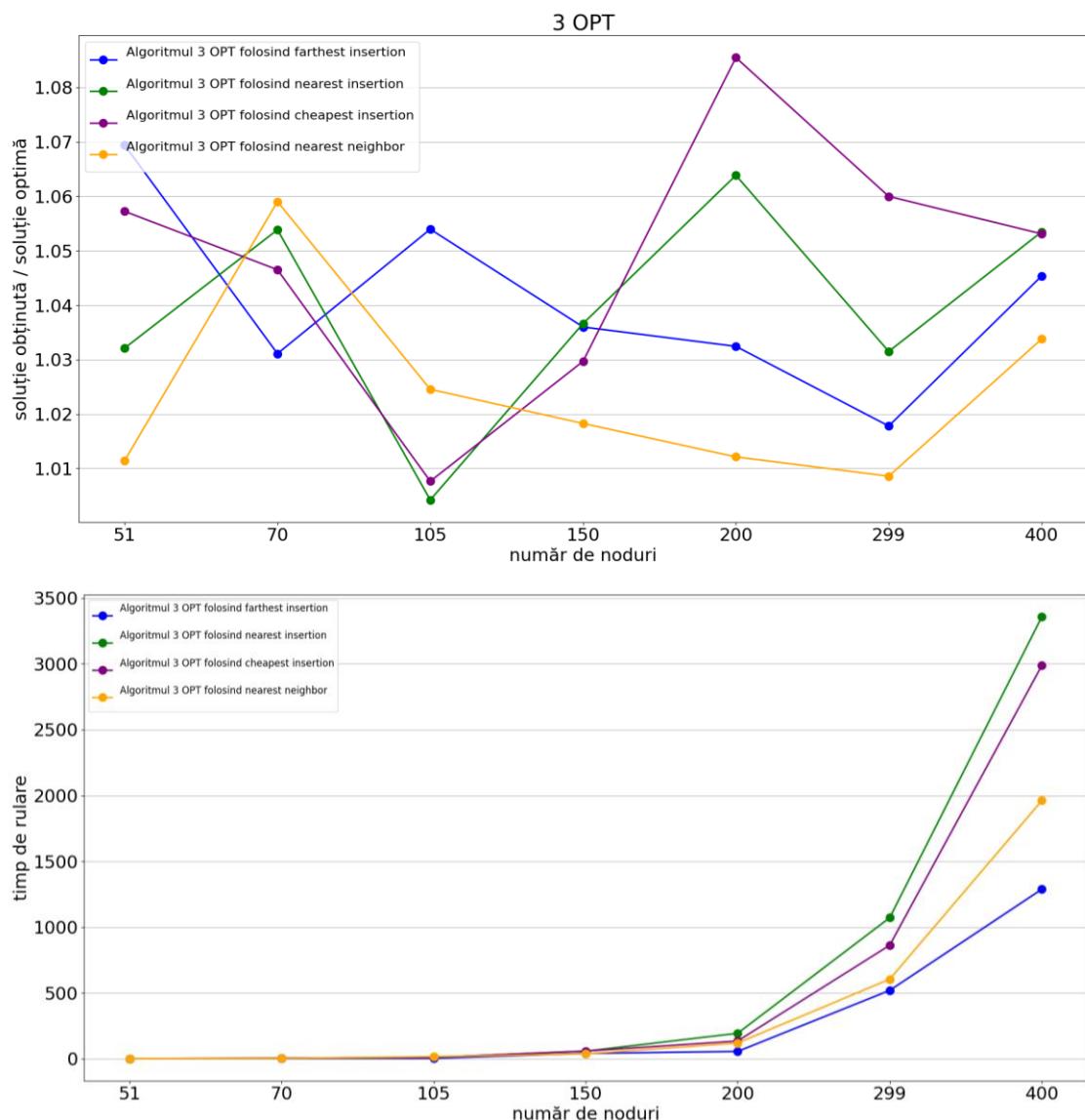
Grafic 5.7. Prezentarea vizuală a datelor din Tabelul 5.4. pentru a ușura înțelegerea informațiilor.

În continuare, se trece la algoritmul 3-OPT cu o singură repetare a programului și se compară datele între ele dorind să se afle care este cel mai bun algoritm pentru instanțele folosite.

		3 OPT							
Instanță		Farthest insertion		Nearest insertion		Cheapest insertion		Nearest neighbor	
nume	OPT	soluție	temp	soluție	temp	soluție	temp	soluție	temp
eil51	426	455.562	0.343	439.6815	0.6772	450.38	0.5461	430.883	0.747
st70	675	695.9915	1.5143	711.3482	3.0846	706.392	3.7033	714.8343	2.152

lin105	14379	15154.8176	3.2037	14439.7698	10.1705	14489.0793	10.373	14731.8591	15.7181
ch150	6528	6762.9538	40.1589	6767.0965	57.0279	6721.8022	57.8137	6647.2672	37.119
kroA200	29368	30320.4393	55.1369	31243.2815	192.9551	31878.2022	134.4868	29724.1723	119.0981
pr299	48191	49048.8177	518.7884	49708.9198	1071.2798	51081.5582	861.8909	48604.7511	604.0316
rd400	15281	15973.8463	1287.3048	16097.8453	3356.6178	16092.4633	2986.9031	15797.0347	1960.5541

Tabel 5.5. Datele de ieșire rezultate în urma rulării algoritmului 3-OPT împreună cu farthest insertion, nearest insertion, cheapest insertion și nearest neighbor știind ca algoritmul nu se repetă.



Grafic 5.8. Prezentarea vizuală a datelor din Tabelul 5.5. privind execuția algoritmului 3-OPT.

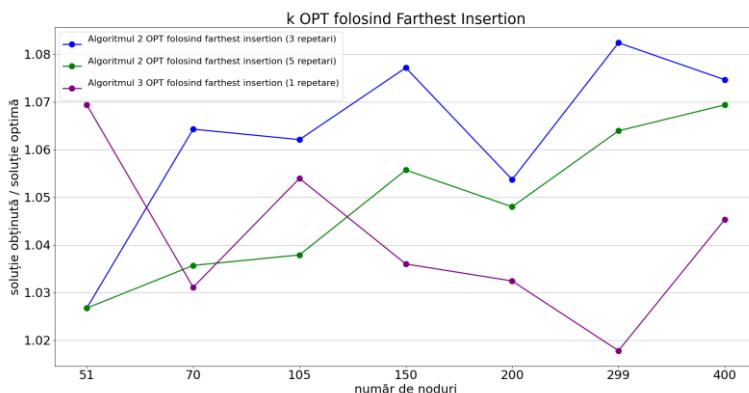
3-OPT este un algoritm bun, rezultatele obținute fiind apropiate ca valoare de soluția optimă (soluțiile găsite sunt mai mici ca $1.09 * \text{OPT}$) prin aceasta demonstrând că este un algoritm potrivit de folosit dacă avem un număr relativ mic de noduri în graf.

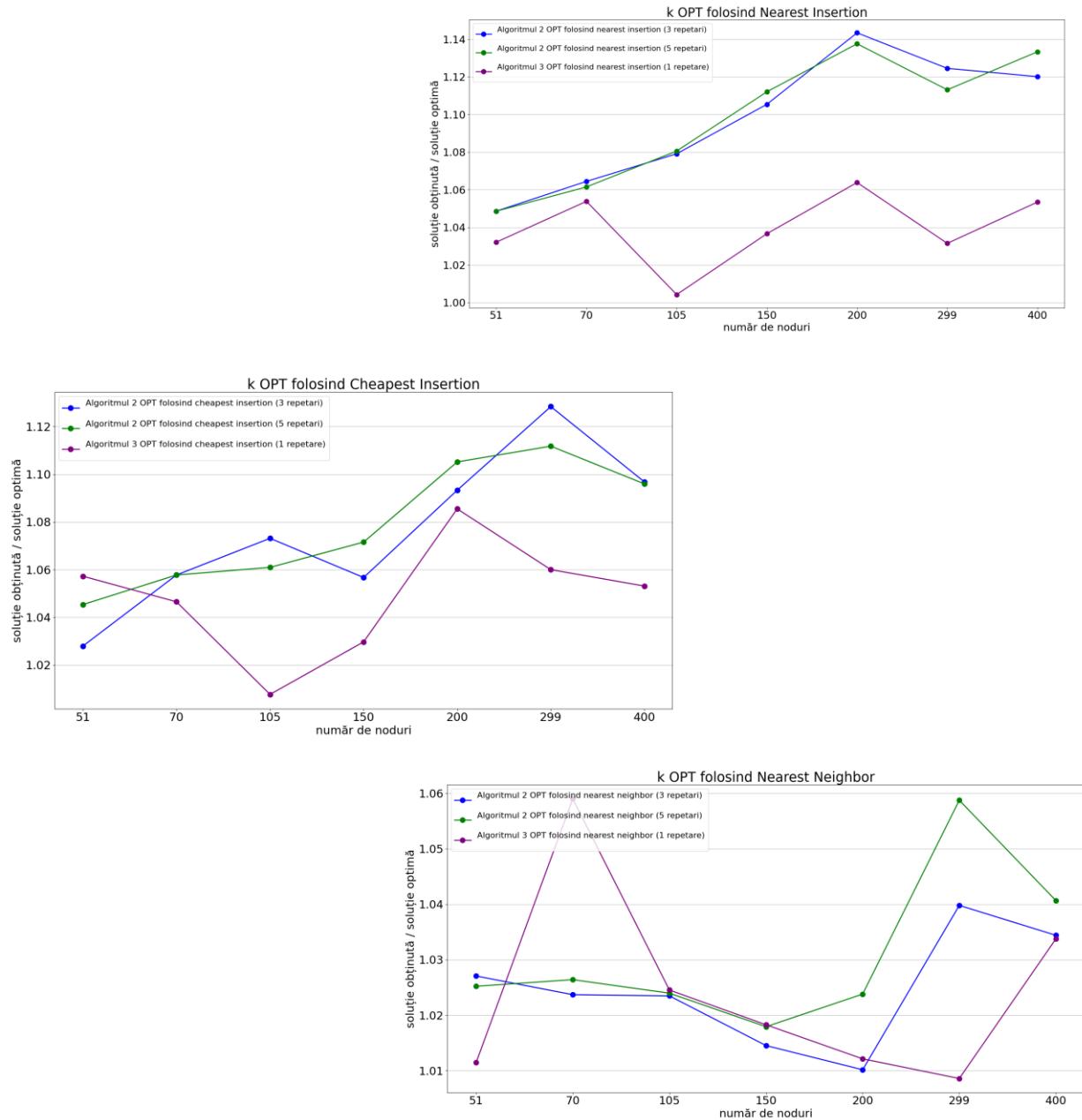
Timpul de rulare se observă că crește semnificativ odată cu numărul de noduri (*Graficul 5.8.b.*). Pentru 400 noduri, algoritmul ce folosește strategia nearest insertion este cel mai lent, reușind să dea un răspuns în aproximativ 3500 secunde ≈ 58 minute. Comparându-l cu cea mai rapidă strategie, farthest insertion, care răspunde în mai puțin de 1500 secunde (25 min), performanța celui de al doilea este mai bună per total, aşadar nu este avantajos să folosim nearest insertion. Alta varianta care merită amintită este cea care utilizează algoritmul nearest neighbor care se remarcă cu rezultatele cele mai scăzute pentru instanțele cu număr mai mare de noduri și cu timpul care crește doar până la 33 de minute.

Privind la toate datele, euristică nearest neighbor este cea mai avantajoasă arătat prin raportare la valorile calitate – durata că este mai eficientă ca celelalte euristici discutate.

În concluzie, 3-OPT este prea costisitor când vine vorba de timpul de rulare, iar în locul său se poate folosi 2-OPT cu 3 (sau mai multe) repetări folosind euristică nearest neighbor care a returnat soluții foarte apropiate de cele optime într-un timp scurt (< 20 secunde) după cum se observă în *Graficul 5.8.* și *Tabelul 5.5.*

Pentru a revizui datele comparând performanțele fiecărui algoritm euristic cu el însuși când este introdus în 2-OPT (3 repetări și 5 repetări) și 3-OPT (o repetare) se urmărește *Graficul 5.9.* care este compus din 4 grafice, câte unul pentru fiecare euristică de determinare a ciclului hamiltonian inițial.





Grafic 5.9. Reprezentarea celor 4 algoritmi euristici de determinare a ciclului hamiltonian în comparație cu sine integrat în 2-OPT cu 3 repetări, 2-OPT cu 5 repetări și 3-OPT cu o repetare.

Grafcile expuse arată la algoritmii care folosesc farthest insertion și nearest neighbor oferă rezultate cele mai bune (comparând valoarea maximă a raportului soluție obținută – soluție optimă). La celelalte 2 euristici, se diferențiază algoritmul 3-OPT, dar s-a discutat că acesta este lent comparativ cu algoritmul 2-OPT, indiferent de numărul de repetări a acestuia.

5.4. Compararea algoritmilor aproximativi și euristici

Algoritmii exacti sunt lenți și rulează doar pe instanțe cu un număr mic de noduri. Prin urmare, acest tip de algoritmi nu sunt luați în calcul în comparația din acest subcapitol.

În comparația realizată se iau face referire doar la instanțele menționate de pe *TSPLIB* cu noduri cuprinse între 51 și 400.

Algoritmii euristici sunt eficienți din punct de vedere a soluției rezultate, obținând valori mai mici de $1.15 * \text{OPT}$, dar timpul de rulare este crescut, maximul întâlnit fiind de 3500 secunde (≈ 58.3 min) pentru algoritmul 3-OPT. De partea cealaltă, execuția algoritmilor de aproximare durează maxim 3.0409 secunde și se obțin rezultate mai mici de $1.42 * \text{OPT}$.

În *Subcapitolul 5.2* s-a ajuns la concluzia că algoritmul lui Christofides este cel mai eficient dintre cei aproximativi în cazul în care instanțele au mai puțin de 1400 de noduri, iar pentru acest algoritm maximul de timp atins este este cel precizat mai sus (3.0409) și soluțiile sunt mai mici de $1.37 * \text{OPT}$. 1

În *Subcapitolul 5.3* s-a ales algoritmul 2-OPT. Varianta cu 3 repetări a rulat în maxim de 10 sec și valori mai mici de $1.4 * \text{OPT}$, iar la varianta cu 5 repetări timpul crește la 15 secunde și valori mai mici de $1.6 * \text{OPT}$. Aceste rezultate sunt obținute din cauza strategiei greedy care în cazul celei de a doua variante a dus la formarea unor cicluri hamiltoniene de lungime mai mare chiar dacă a avut 5 repetări ale algoritmului. Iar ciclul hamiltonian influentează soluția finală.

Prin urmare, algoritmul 2-OPT este mai eficient când privim la soluție, iar algoritmul lui Christofides este mai rapid. În plus, acesta din urmă este stabil având un factor de aproximare de $\frac{3}{2}$ indiferent de instanță. Algoritmul neighbor insertion este $\frac{1}{2} \ln n + \frac{1}{2}$ – aproximativ, neputând însă fi aproximată soluția algoritmului care îl apelează (2-OPT).

Concluzionând, consider că algoritmul lui Chistofides este cel mai potrivit pentru a oferi o soluție pentru problema TSP.

6. Aplicație web

Prin această aplicație propusă și implementată am încercat să înglobez funcționalitățile necesare înțelegерii în profunzime a problemei comis-voiajorului prin prezentarea acesteia și a algoritmilor cuprinși în lucrare (*capitolele 2, 3 și 4*). Informațiile sunt prezentate într-o manieră intuitivă și detaliată, utilizatorul având acces, din intermediul aplicației, la explicații scrise, poze sugestive, pseudocod care urmărește pașii algoritmilor, cod atașat care poate fi rulat în aplicație pentru diverse tipuri de instanțe primind ca răspuns soluția și ciclul hamiltonian obținut reprezentat grafic, plus alte date folositoare utilizatorului.

6.1. Tehnologii utilizate

În această secțiune sunt descrise tehnologiile folosite pentru a implementa aplicația și detalii legate de acestea.

6.1.1. Flask [5, 17, 25]

Flask este un microframework (termen folosit pentru un framework web minimalist care nu necesită anumite biblioteci sau tool-uri) scris în Python care facilitează acceptarea unui *HTTP request*, rutează cererea către o funcție și prin intermediul ei returnează un *HTTP response*.

Flask oferă sugestii, dar nu obligă folosirea unor dependențe sau a unui stil de proiect (project layout). Lăsă pe seama programatorului să aleagă tool-urile și bibliotecile pe care vrea să le folosească, oferind multe extensii de comunitate care face ușoară adăugarea unei funcționalități.

Referitor la aplicația creată, am ales să folosesc Flask în implementarea back-end-ului pentru a putea rula prin intermediul acestuia algoritmii implementați în Python. Fiind o aplicație informativă, nu este nevoie de autentificare, atribuire de roluri și interacțiune cu baza de date, activitatea făcută de Flask este să primească request-uri trimise din front-end, să acceseze funcțiile din clase și să returneze codul (preluare de cod pentru a fi afișat în pagină) sau răspunsul (obținut prin rularea codului).

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app, origins="http://localhost:4200")
```

CORS (Cross-Origin Resource Sharing) este un mecanism implementat de browsere care blochează site-urile web să ceară date din alt URL. Când se face un request în browser, se adaugă un header de origine la mesajul request-ului și, în cazul în care serverul are aceeași origine, este acceptat, fiind o modalitatea a browser-ului de a asigura securitate.

```
✖ Access to XMLHttpRequest at 'http://127.0.0.1:5000/getCodeBranchAndBound' from localhost:1
origin 'http://localhost:4200' has been blocked by CORS policy: No 'Access-Control-Allow-Origin'
header is present on the requested resource.
```

Pentru a rezolva această problemă, Flask conține un pachet *flask-cors* care ne permite să accesăm URL-uri din front-end.

6.1.2. TypeScript [23]

TypeScript este un limbaj de programare care derivă din JavaScript, adăugându-se la acesta permisiunea de a folosi tipuri de date obiectelor. Această permisiune vine cu avantaje ale utilizării TypeScript-ului precum:

- parametrii funcțiilor și variabilele din JavaScript nu oferă nicio informație cu privire la valori, îngreunând astfel citirea codului pentru un necunoscător fără a accesa documentația
- parametrii funcțiilor din JavaScript nu raportează eroarea dacă se introduce un tip de date care nu este potrivit, în schimb TypeScript-ul raportează, atenționând programatorul că este o problemă cu datele inserate

6.1.3. Angular [1, 15]

Angular este un framework bazat pe componente, fiecare componentă incluzând o clasă TypeScript (pentru care trebuie să se specifică prin decoratorul `@Component`), un template de html și o pagină pentru stiluri. Decoratorul specifică:

- template-ul HTML care spune cum se randează pagina
- selectorul CSS care definește cum este componenta folosită într-un template (componenta poate fi folosită în alte template-uri prin apelarea selectorului)
- pagina de stiluri CSS care specifică cum să arate elementele din template-ul HTML (optional)

```
@Component({
  selector: 'app-approximation-algorithms',
  templateUrl: './approximation-algorithms.component.html',
  styleUrls: ['./approximation-algorithms.component.scss']
})

export class ApproximationAlgorithmsComponent {
```

În plus, oferă o colecție de biblioteci integrate care acoperă o multitudine de feature-uri (comunicare client-server, form-uri) care mi-au fost necesare pentru a crea funcționalități cu scopul de a oferi utilizatorului pagini interactive și stilizate.

În cele ce urmează, se prezintă o listă de biblioteci pe care le-am folosit și care conține module pentru aplicații create în Angular:

- a. Angular Material este biblioteca care se folosește în Google UI. Este construită folosind TypeScript and are ca scop implementarea aplicației pe stilul design-ului de la Google.
 - MatIconModule
 - MatToolbarModule
 - MatButtonModule
 - MatInputModule
 - MatFormFieldModule
 - MatSidenavModule
 - MatTableModule
 - MatPaginatorModule
 - MatSelectModule

- b. NGX Bootstrap este o bibliotecă de componente pentru Angular care extinde capabilitățile framework-ului Bootstrap și ajută programatorii să le folosească în aplicațiile Angular mai ușor. Comparativ cu Bootstrap, NGX Bootstrap este modular, extensibil și ușor de adaptat.
- TranslateLoader
 - TranslateModule
 - TranslateHttpLoader
 - TranslateService
- c. NG Bootstrap este o bibliotecă de componente cunoscută pentru Angular UI inspirată din Bootstrap. Componentele NG Bootstrap-ului sunt construite folosind doar Bootstrap 5 CSS împreună cu API-uri specific realizate pentru Angular. NG Bootstrap oferă programatorilor posibilitatea de a utiliza componente Bootstrap precum carusele, bară de progres, modale și multe altele având cunoștințe reduse de Angular.
- NgbModule
 - ngb-carousel
 - ng-container
- d. NG Stack este o bibliotecă Angular nu foarte cunoscută datorită utilizării minime în proiecte, printre care am descoperit componenta CodeEditorModule ce oferă aspectul unui fișier deschis în Visual Studio. Printre utilitățile acesteia se numără posibilitatea de a oferi textelor atribuția de read-only (nu se pot face modificări asupra obiectului cu această atribuție), diferite teme vizuale ale IDE-ului Visual Studio Code și alte opțiuni care îmbunătățesc experiența de folosire a acestui modul [4]. În aplicația creată, este folosit pentru a expune într-un mod cât mai ușor de vizualizat algoritmii creați.
- e. NG Image Slider este un glisor de imagini receptiv pentru Angular [22]. Feature-uri ale utilizării acestui glisor:
- săgeți înainte-înapoi pentru navigare ce devin vizibile la interacțiune
 - permite apăsarea pe imagine cu scopul de a o mări pentru o mai bună vizualizare
 - acceptă setarea unui titlu și a unei secțiuni cu detalii în interiorul glisorului

6.1.4. Graphology [7] și SigmaJs [21]

Graphology este o bibliotecă multifuncțională de manipulare a grafurilor în JavaScript și TypeScript. Scopul ei este să permită folosirea unei multitudini de grafuri care împart aceeași interfață. Un graf realizat cu Graphology poate fi orientat, neorientat sau mixt și poate emite evenimente ce oferă utilizatorului posibilitatea de a interacționa în timp real cu acesta.

SigmaJs este o bibliotecă JavaScript modernă a cărei scop este reprezentarea de grafuri care pot ajunge să aibă un număr mare de noduri și muchii. Este folositoare pentru randarea și interacțiunea cu grafuri în browser întrucât funcționează în simbioză cu Graphology, reprezentând partea de front-end, iar Graphology back-end-ul.



Figura 6.1. Imagine preluată de pe site-ul oficial [21]

În aplicația dezvoltată, biblioteca a fost folosită pentru a expune în pagină grafuri în mod dinamic pe baza informațiilor ce sunt primite în urma rulării de către utilizator a unui program.

6.2. Arhitectura aplicației create

În această secțiune voi prezenta detalii despre modul în care a fost construită aplicația. Se vor lua separat cele două părți ale proiectului și se va discuta pe baza modului în care sunt structurate, precum și motivele din spatele alegerilor făcute.

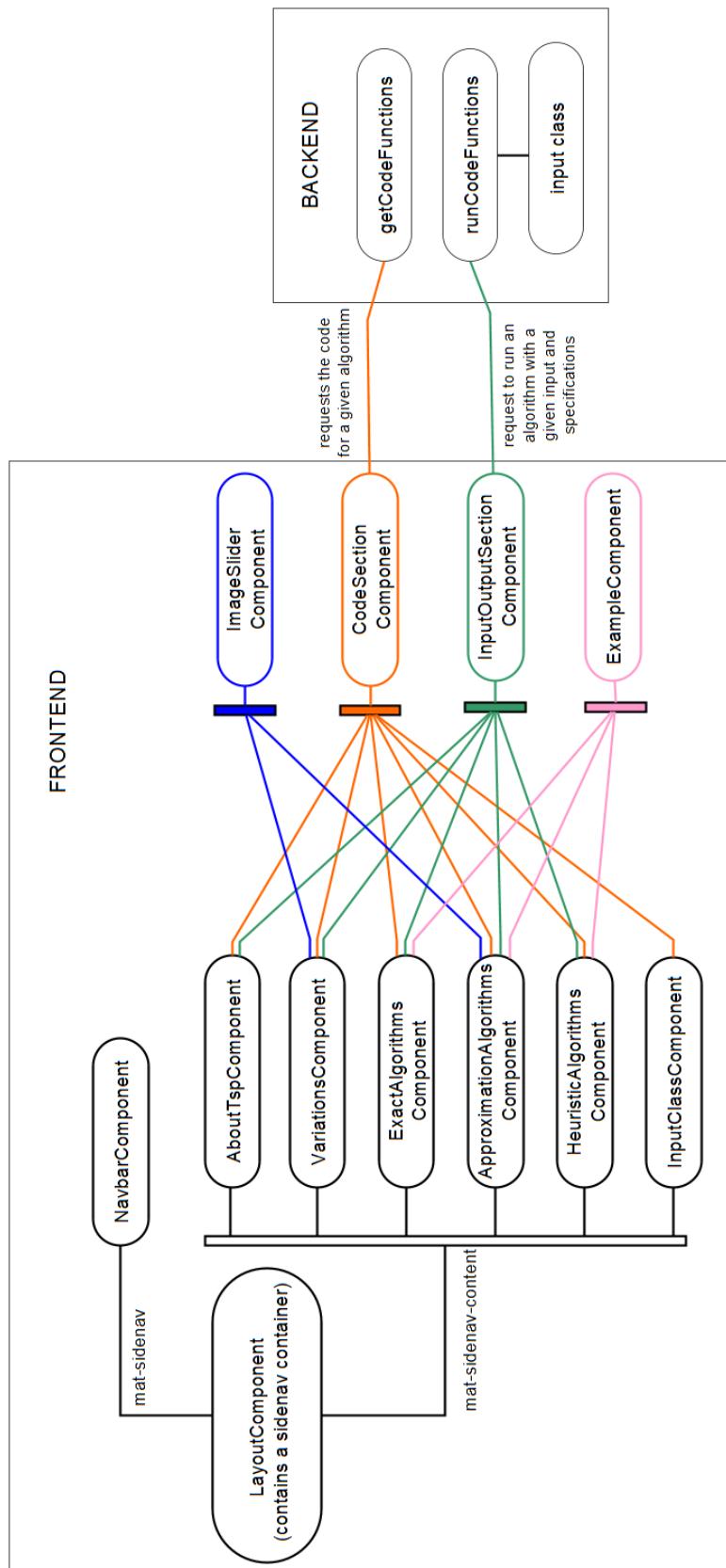


Figura 6.2. Structura aplicației

6.2.1. Back-end-ul aplicației

Având în vedere că aplicația se bazează mai mult pe partea vizuală, backend-ul este unul simplist. Există un fișier principal *main.py* care trebuie să ruleze permanent pentru a avea acces la funcțiile din back-end, acesta fiind acela care comunică cu front-end-ul de la care primește HTTP request-uri și căruia îi răspunde prin HTTP response-uri.

Fiecare algoritm prezentat în lucrare în *capitolele 2, 3 și 4* sunt stocați local, fiind reținuți în fișiere pentru a fi accesibili când utilizatorul accesează diferite pagini din aplicație. Fiecare algoritm necesită două funcții, una de preluare a clasei ce o face vizibilă pentru utilizator și alta de rulare care primește un input dat cu scopul de a oferi utilizatorului un răspuns.

```
53  @app.route('/getCodeChristofides', methods=["GET"])
54  def getCodeChristofides():
55      f = open("algorithm_christofides.py", mode="r")
56      cod = f.read()
57      f.close()
58      return jsonify(cod), 200
59
171
172  @app.route('/runCodeChristofides', methods=["POST"])
173  def runCodeChristofides():
174      data = request.get_data(as_text=True)
175      decoded_data = json.loads(data)
176
177      inputData = decoded_data["input"]
178      inputType = decoded_data["inputType"]
179      algoritm = decoded_data["algoritm"]
180      noRepetitions = decoded_data["noRepetitions"]
181      try:
182          input = Input(inputData, inputType)
183          matrix = input.createMatrix()
184          input.isComplete()
185          input.isMetric()
186          christofides = runner_christofides.Christofides(input.n, matrix)
187          response = christofides.TSP(algoritm, noRepetitions)
188
189      except Exception as e:
190          return jsonify(e.args), 500
191
192      return jsonify(response), 200
193
```

Când se rulează codul, sunt apelate, în general, alte clase decât cele accesibile utilizatorului deoarece, chiar dacă conțin același cod, este nevoie de diferite procesări pentru a putea trimite informații utilizatorului în pagină, uneori mai multe decât cele returnate în clasa vizibilă. Fișierele în care se găsesc clasele acestea sunt notate folosind prefixul *runner*, sugerând astfel că scopul lor este rularea, nu afișarea codului.

Reamintesc algoritmii discutați și numărul de fișiere necesare pentru fiecare, specificând și necesitatea acestora:

Număr algoritm	Algoritm	1 fișier (se completează cu informațiile care sunt returnate dacă s-ar apela)	2 fișiere (se completează cu informații suplimentare ce sunt returnate când se apelează, pe lângă cele specificate în coloana 1 fișier)
1	Algoritmul bazat pe programare dinamică	<ul style="list-style-type: none"> • soluția 	<ul style="list-style-type: none"> • ciclul hamiltonian • matricea M în care sunt reținute distanțele) • înlocuirea soluției cu -1 dacă pentru inputul dat nu există soluție
2	Algoritmul bazat pe branch and bound	<ul style="list-style-type: none"> • soluția • ciclul hamiltonian 	<ul style="list-style-type: none"> • matricea redusă corespunzătoare rădăcinei • matricele reduse ale primului pas (copiii direcți ai rădăcinei) • înlocuirea soluției cu -1 dacă pentru inputul dat nu există soluție
3	Algoritmul arborelui dublu	<ul style="list-style-type: none"> • soluția • ciclul hamiltonian 	<ul style="list-style-type: none"> • arborele returnat de algoritmul lui Prim
4	Algoritmul lui Christofides	<ul style="list-style-type: none"> • soluția • ciclul hamiltonian 	<ul style="list-style-type: none"> • arborele returnat de algoritmul lui Prim • lista de noduri de grad impar • muchiile returnate de matching
5	Algoritmul farthest insertion	<ul style="list-style-type: none"> • soluția • ciclul hamiltonian 	-
6	Algoritmul nearest insertion	<ul style="list-style-type: none"> • soluția • ciclul hamiltonian 	-
7	Algoritmul cheapest insertion	<ul style="list-style-type: none"> • soluția • ciclul hamiltonian 	-
8	Algoritmul	<ul style="list-style-type: none"> • soluția 	-

	nearest neighbor	<ul style="list-style-type: none"> ciclul hamiltonian 	
9	Algoritmul 2- OPT	<ul style="list-style-type: none"> soluția ciclul hamiltonian 	<ul style="list-style-type: none"> ciclul hamiltonian determinat de euristică ciclul hamiltonian determinat de primul <i>2Schimb</i> realizat
10	Algoritmul 3- OPT	<ul style="list-style-type: none"> soluția ciclul hamiltonian 	<ul style="list-style-type: none"> ciclul hamiltonian determinat de unul dintre euristici ciclul hamiltonian determinat de primul <i>3Schimb</i> realizat

6.2.2. Front-end-ul aplicației

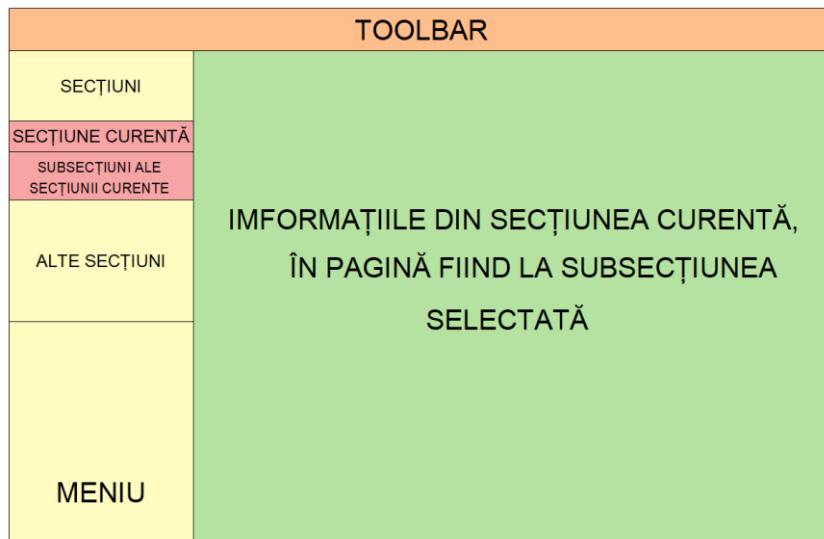


Figura 6.3. Structura front-end-ului

Front-end-ul este realizat cu ajutorul framework-ului Angular. Prin urmare, are structura specifică acestuia, fiecare componentă conținând 3 fișiere:

- *nume_fișier.ts* - păstrează logica componentei;
- *nume_fișier.html* - conține codul html ce va fi interpretat de browser pentru a reprezenta vizual datele;
- *nume_fișier.css* - are scopul de a stiliza elementele vizuale găsite în fișierul cu extensia *.html*

```

    < app
      < components
        < about-tsp
          < about-tsp.component.html
          < about-tsp.component.scss
          TS about-tsp.component.ts
        > approximation-algorithms
        > code-section
        > exact-algorithms
        > example-component
        > heuristic-algorithms
        > image-slider
        > input-class
        > input-output-section
        > layout
        > navbar
        > variations
        > models
        > services
      TS app-routing.module.ts
      < app.component.html
      < app.component.scss
      TS app.component.spec.ts
      TS app.component.ts
      TS app.module.ts

```

Referitor la modul în care este structurată vizual aplicația, am oferit o soluție ușor de dedus și navigat: în stânga sunt prezentate secțiunile cu subsecțiunile aferente lor, iar în partea dreaptă se afișează conținutul secțiunii selectate. Pentru fiecare secțiune a fost creată o componentă după cum urmează:

- Secțiunii *Prezentare generală* îi corespunde AboutTspComponent
- Secțiunii *Variatii* îi corespunde îi corespunde VariationsComponent
- Secțiunii *Algoritmi exacti* îi corespunde ExactAlgorithmsComponent
- Secțiunii *Algoritmi de aproximare* îi corespunde ApproximationAlgorithmsComponent
- Secțiunii *Algoritmi euristicici* îi corespunde HeuristicAlgorithmsComponent
- Secțiunii *Clasa Input* îi corespunde InputClassComponent

Componentele ImageSliderComponent, CodeSectionComponent, InputOutputComponent și ExampleComponent sunt reutilizabile adică sunt apelate din interiorul mai multor alte componente. Acest lucru este făcut fiindcă este nevoie de ele în mai multe secțiuni și nu este rentabil să se creeze mereu câte o altă componentă asemănătoare cu una deja existentă.

```
201
202 <app-code-section [currentSubsection]=christofidesAlgorithm></app-code-section>
203 <app-input-output-section [currentSubsection]=christofidesAlgorithm></app-input-output-section>
204
```

Servicii

Partea de front-end a proiectului folosește două servicii. Primul este cel prin care se trimit cereri care back-end și se primesc răspunsuri, iar al doilea reține informații ce sunt necesare pentru proiect (denumirile secțiunilor și subsecțiunilor) și preia datele reținute în *storage.json*. Acest fișier conține date ce nu se modifică precum informații despre secțiuni din meniu și despre pozele din glisor pentru a oferi utilizatorului o experiență cât mai plăcută.

6.3. Funcționalități aplicație

6.3.1. Validarea inputului

Trimitera datelor către back-end de un utilizator se realizează prin intermediul câmpurilor de input aferente secțiunilor de cod rulabile. De aceea, este nevoie de validări pentru a ne asigura că formatul și informația poate fi interpretată de algoritm. Verificările se realizează în back-end, înainte de apelarea algoritmul care rezolvă TSP. Utilizatorul are posibilitatea să aleagă dintre 3 modele de input când dorește să ruleze un algoritm exact și 2 tipare de input în restul cazurilor.

Există anumite verificări universale, precum:

- verificare dacă este introdus numărul necesar de date. În caz contrar se afișează mesajul "*The entered input does not have the correct form!*"
- verificare dacă a fost introdus un număr negativ. În caz contrar este aruncată următoarea eroare: "*You entered at least one negative number!*"
- verificare dacă numerotarea nodurilor este corectă. Nu este permisă numerotarea nodurilor în alt fel decât de la 0 la $n - 1$, unde n este numărul de noduri specificat pe primul rând din input. Dacă este introdus un număr mai mare se aruncă următoarea eroare "*Nodes are numbered from 0 to {0} , but node {1} was found!*".format(self.n-1, i) (pe prima poziție se introduce numărul de noduri, iar pe a doua valoarea găsită).

În continuare sunt prezentate cele 3 forme de input și erorile ce sunt tratate pentru fiecare în parte.

1. Inputul ce corespunde unui graf neorientat

Prima linie trebuie să conțină două valori, prima semnificând numărul de noduri și a doua numărul de muchii. Pe următoarele linii se cer triplete de forma *primul-nod al-doilea-nod distanță*. În cazul acesta se fac următoarele verificări:

- numărul de triplete este același ca numărul de muchii specificate, altfel se aruncă eroarea: *"The specified number of edges ({0}) is different from the number of lines of the type: node 1 node 2 value ({1})".format(self.m, len(self.data))* unde pe prima poziție se introduce numărul de muchii, iar pe a doua numărului de triplete.
- între datele introduse se află două tupluri identice de noduri, se aruncă eroarea: *"Already set the distance from node {0} to {1}!"*.format(i, j) fiind specificare nodurilor între care distanța a fost deja stabilită
- algoritmii aproximativi și euristicii necesită ca datele introduse să corespundă unui graf complet și să respecte inegalitatea triunghiului (rezolvă TSP în cazul metric). Așadar dacă inputul este introdus pentru unul din aceste tipuri de algoritmi, se verifică dacă este specificată o pondere între oricare două noduri și se cere ca distanțele din matricea creată să respecte inegalitatea triunghiului. În caz contrar, pentru prima verificare se aruncă eroarea: *"The given graph is not complete!"*, iar pentru a doua eroarea: *"The given set of data does not transform to a matrix that respects the triangle inequality!"*

```
# verify if the graph is complete (needed for approximation and heuristic algorithms)
def isComplete(self):
    # the matrix created using euclidean distances is already complete
    if self.type != 3:
        if np.count_nonzero(self.D == np.inf) != self.n:
            raise Exception("The given graph is not complete!")

# just for input type == 1
# the matrix is symmetric: D[i, j] = D[j, i] and D[i, k] + D[k, j] = D[j, k] + D[k, i]
def isMetric(self):
    if self.type == 1:
        for i in range(self.n - 1):
            for j in range(i + 1, self.n):
                for k in range(self.n):
                    if i != k and j != k:
                        if self.D[i, k] + self.D[k, j] < self.D[i, j]: # does not hold the triangle inequality
                            raise Exception("The given set of data does not transform to a matrix that respects the triangle inequality!")
```

2. Inputul ce corespunde unui graf orientat

Acest tip de input este acceptat doar în cazul algoritmilor exacti. Prima linie trebuie să conțină două valori, semnificând numărul de noduri și numărul de muchii ce trebuie introduse. Pe următoarele linii se cer triplete de forma *primul-nod al-doilea-nod distanță*. În cazul acesta se verifică:

- dacă numărul de triplete este același ca numărul de muchii specificate, altfel se aruncă eroarea următoare: "*The specified number of edges ({0}) is different from the number of lines of the type: node 1 node 2 value ({1})*".format(*self.m*, len(*self.data*)) unde pe prima poziție se completează cu numărul de muchii, iar pe a doua cu numărului de triplete.
- o altă verificare este pentru cazul în care utilizatorul dorește să seteze de două ori distanța între aceleași noduri, se aruncă eroarea: "*Already set the distance from node {0} to {1}!*".format(*i, j*) fiind specificare nodurile între care distanța a fost deja stabilită

3. Inputul dat corespunde unui graf neorientat și complet care respectă inegalitatea triunghiului. Se specifică distanțele euclidiene pentru orice nod din graf.

Pe prima linie se introduce o singură valoare *n* semnificând numărul de noduri din graf, iar pe următoarele *n* linii triplete de forma *nod coordonată-X coordonată-Y*. Datele corespund unui TSP euclidian care este un caz special de TSP metric (*Secțiunea 1.2.4*). În cazul acesta se verifică:

- dacă numărul de triplete este același cu numărul de noduri specificate, altfel se aruncă eroarea următoare: "*The number of nodes is different than specified!*"
- dacă indexele nodurilor introduse sunt unice, în caz contrar se ridică eroarea: "*Already set the coordinates for node {0}!*".format(int(*inputList*[*index*])) fiind specificat nodul care se repetă
- dacă utilizatorul dorește să insereze două noduri diferite cu aceleași coordonate, se aruncă eroarea "*Can not have the same coordinates for 2 different nodes!*"

6.3.2. Schimbarea limbii afişate

Pentru face aplicația accesibilă cât mai multor persoane, am adăugat opțiunea de a selecta una din limbile disponibile. În implementarea selectorului de limbă s-au utilizat modulele de translate din NGX Bootstrap. Schimbarea unei limbi se face ușor prin traducerea și salvarea textelor prezente în aplicație într-un fișier *.json*.

```
7   <div class="title">
8     <h2>{{ 'approximations-algorithms.double-tree-algorithm.title' | translate }}</h2>
9   </div>
10
11  <p>{{ 'approximations-algorithms.double-tree-algorithm.header' | translate }}</p>
12  <div>
13    <ul>{{ 'approximations-algorithms.double-tree-algorithm.eulerian.0' | translate }}
14      <li *ngFor="let i of [1,2,3]">
15        {{ 'approximations-algorithms.double-tree-algorithm.eulerian.' + i | translate }}
16      </li>
17    </ul>
18  </div>
19
20  <div class="theorem">
21    <p class="highlight">{{ 'general.theorem' | translate }} </p>
22    <p>{{ 'approximations-algorithms.double-tree-algorithm.theorem-subset-hamiltonian-cycle' | translate }}</p>
23  </div>
```

Schimbarea limbii se face automatizat apelând funcția următoare:

```
44  switchLanguage(language: string) {
45    this.translate.use(language);
46    localStorage.setItem('language', language);
47    window.location.reload();
48  }
```

Pentru traduceri avem posibilitatea de a adăuga ulterior diferite limbi făcând aplicația creată cât mai accesibilă pentru o audiență mai mare. Tot ce trebuie făcut pentru această modificare este să se adauge un fișier de tip *json* cu traducerile textelor în limba dorită în lista de fișiere corespunzătoare limbilor reprezentate din directorul *assets/i18n*.

```
▽ assets
  ▽ i18n
    { en.json
    { ro.json
```

```

export function HttpLoaderFactory(http: HttpClient): TranslateHttpLoader {
  return new TranslateHttpLoader(http);
}

export function ApplicationInitializerFactory(translate: TranslateService, injector: Injector) {
  console.log(translate);
  return () => new Promise<any>((resolve: any) => {
    const locationInitialized = injector.get(LOCATION_INITIALIZED, Promise.resolve(null));
    locationInitialized.then(() => {
      translate.addLangs(['ro', 'en']);
      let selectedLanguage = localStorage.getItem('language');
      if (selectedLanguage) {
        translate.setDefaultLang(selectedLanguage);
      }
      else {
        translate.setDefaultLang('ro');
        selectedLanguage = 'ro';
        localStorage.setItem("language", 'ro');
      }

      translate.use(selectedLanguage).subscribe(() => {
        console.info(`Successfully initialized '${selectedLanguage}' language.`);
      }, err => {
        console.error(`There was a problem with '${selectedLanguage}' language initialization.`);
      }, () => {
        resolve(null);
      });
    });
  });
}

```

6.3.3. Integrarea de grafuri dinamice

Grafurile sunt create folosind SigmaJS împreună cu Graphology și sunt dinamice, mulându-se pe datele primite din back-end. Grafurile se creează într-un fișier de TypeScript prin construirea lor folosind datele primite și se afișează în pagină în zona de output a componentei.

A fost destul de greu de interactionat cu ele deoarece necesită multe variabile folosite. Un graf trebuie să fie instantiat, să aibă un container în html și un renderer care să afișeze graful în pagină prin apelarea funcției `renderer.refresh()`. Cum sunt cazuri în care este nevoie de mai multe grafuri în secțiunea output pentru un algoritm rulat, este nevoie de acele inițializări pentru fiecare graf în parte.

Ca funcționalitate grafurile sunt interactive, adică se poate selecta un nod și se poate mări și micșora graful.

```

<div *ngIf="currentSubsection === doubleTreeAlgorithm">
  <div *ngIf="responseDoubleTree.solution !== -1">
    {{'code-section.hamiltonian-cycle' | translate}}: {{responseDoubleTree.cycle.join(" -> ")}}
  </div>
  <div id="graph-double-tree" class="cont"></div>
</div>

```

```

graphDoubleTreeCycle = new Graph();
graphDoubleTreePrim = new Graph();
containerDoubleTreeCycle: any;
containerDoubleTreePrim: any;

runDoubleTree(input: string): void {
    this.responseDoubleTree.solution = -1;
    this.solution = -1;

    if (this.graphDoubleTreeCycle.size != 0) {
        this.cleanGraph(this.graphDoubleTreeCycle);
        this.renderer.refresh();
        this.cleanGraph(this.graphDoubleTreePrim);
        this.rendererTree.refresh();
    }
    this.appService.runDoubleTree(input, Number(this.inputType)).subscribe(
        (response: any) => {
            this.responseDoubleTree.solution = response[0];
            this.responseDoubleTree.tree = response[1];
            this.responseDoubleTree.cycle = response[2];

            if (this.responseDoubleTree.solution !== -1) {
                this.solution = response[0];

                this.containerDoubleTreeCycle = document.getElementById("graph-double-tree") ?? new HTMLElement();
                this.containerDoubleTreePrim = document.getElementById("graph-double-tree-prim") ?? new HTMLElement();

                this.containerDoubleTreeCycle.innerHTML= "";
                this.containerDoubleTreePrim.innerHTML= "";

                this.renderer = new Sigma(this.graphDoubleTreeCycle, this.containerDoubleTreeCycle, {
                    renderEdgeLabels: true,
                    allowInvalidContainer: true,
                });
                this.rendererTree = new Sigma(this.graphDoubleTreePrim, this.containerDoubleTreePrim, {
                    renderEdgeLabels: true,
                    allowInvalidContainer: true,
                });

                this.createNodes(this.graphDoubleTreeCycle, this.responseDoubleTree.cycle.length - 1);
                this.createNodes(this.graphDoubleTreePrim, this.responseDoubleTree.cycle.length - 1);

                this.createCycle(this.graphDoubleTreeCycle, this.responseDoubleTree.cycle);
                this.renderer.refresh();

                this.createEdges(this.graphDoubleTreePrim, "grey", this.responseDoubleTree.tree);
                this.rendererTree.refresh();
            } else {
                this.hasResponse = false;
            }
        }, (error) => {
            console.log(error);
            this.solution = error.error;
            this.hasResponse = false;
            this.error = true;
        }
    );
}
}

```

6.3.4. Integrarea code-editorului

Code-editorul este un modul nice-to-have pentru că aduce un aspect profesional aplicației oferind în același timp un mediu familiar utilizatorului de vizualizare a codului.

Chiar dacă are opțiuni interesante, eu l-am folosit strict pentru afișarea codului. Ideea inițială era diferită: ca orice cod să poată fi modificat de utilizator, iar varianta sa de cod să poată fi rulată. Astfel acesta nu mai necesita folosirea unei alte aplicații sau unui editor online. Dar această ideea nu a fost posibilă din diferite motive înșiruite mai jos:

- dacă se acceptă modificări în timp real (fără să existe un buton de submit) înseamnă că în back-end se modifică codul și se rulează chiar dacă este incomplet. Din aceasta cauză, primește inevitabil erori care pot duce ușor la picarea back-end-ului aplicației. Repornirea se realizează doar manual, deci nu este eficientă modificarea în timp real
- dacă modificările se aplică doar la cererea utilizatorului prin apăsarea pe un un buton se submit apar alte probleme:
 - nu se poate prezice formatul rezultatelor. De exemplu, nu se poate trimite valoarea infinit (∞) în front-end și fiind vorba de o problemă de minimizare, inevitabil vor apărea în anumite cazuri și valori de infinit. Un alt exemplu este că nu se pot întoarce liste din clasa *numpy*, iar acestea trebuie convertite la liste obișnuite întâlnite în Python
 - nu se pot insera mesaje în secțiunea de output pentru că nu se știe conținutul acestuia prin urmare ar trebui afișat aşa cum este trimis din back-end (nu este estetic)
 - din nou, dacă utilizatorul apasă pe buton înainte de a termina de scris, pot apărea erori la rularea back-end-ului

Din aceste motive am ales să ofer codului atribuția `read-only`, astfel încât validarea inputului să fie realizată înainte de rularea codului și în cazul erorilor să poată fi afișate mesajele corespunzătoare fără a întâmpina erori de rulare.

Code-editorul se află într-o componentă reutilizabilă deoarece afișează codul-text în funcție de identificatorul transmis.

```

1 <div id="container">
2   <ngs-code-editor
3     theme="vs-dark"
4     [codeModel]="model"
5     [readOnly]="readOnly"
6     [options]="options"
7   >
8   </ngs-code-editor>
9 </div>
10

```

6.3.5. Crearea secțiunii de input-output

Secțiunea de input-output este cea mai complexă din aplicație deoarece este porțiunea de interacțiune cu utilizator. Aceasta va fi abordată în două părți, discutând pe rând despre partea de inserare de date și cea de afișare a rezultatelor:

1. Input

Secțiunea de input este reprezentată printr-un *textarea* în care utilizatorul inserează date și alege opțiuni referitoare la cod (de exemplu, numărul de repetări a algoritmului, tipul de heuristică) sau la tipul de date inserate. Inputul se trimită în back-end sub formatul unui .json și la primire, se fac validările specifice algoritmului și tipului de input selectat. În urma verificărilor, se trimit utilizatorului mesaje de eroare sugestive pentru a repara greșelile făcute sau este apelat algoritmul dorit.

2. Output

După ce inputul este procesat cu succes, algoritmul respectiv este rulat pe setul de date introdus. Răspunsurile trebuie să fie transformate înainte de a fi trimise pentru a evita problemele menționate în Secțiunea 6.3.3.:

- evitarea infinitului prin înlocuirea lui `-1` care, având în vedere că ponderile grafului sunt nenegative pentru problema TSP, este un număr ce nu are cum să apară în altă situație. Pentru afișaj, în front-end transformarea trebuie să se trateze în sens invers
- transformarea listelor din clasa numpy în liste din Python folosind funcția predefinită `.tolist()`

Outputul este ulterior preluat și afișat în pagină în funcție de formatul ales pentru algoritmul respectiv, fiecare rulare conținând două informații comune, soluția și ciclul care au

determinat soluția respectivă care este reprezentată vizual printr-un graf. La acestea, se adaugă diverse pași în funcție de algoritm testat:

- Algoritm bazat pe programare dinamică: matricea M rezultată în urma aplicării recursiei
- Algoritm bazat pe Branch and bound: matricele corespunzătoare reducerii nodului de start și a copiilor acestora
- Algoritmul arborelui dublu: arborele obținut în urma rulării algoritmului lui Prim
- Algoritmul lui Christofides: arborele obținut în urma rulării algoritmului lui Prim, arborele în care sunt evidențiate nodurile de grad impar și graful obținut în urma matchingului
- Algoritmii 2-OPT și 3-OPT: ciclul obținut de euristică aleasă și, dacă este cazul, prima optimizare a rezultatului prin modificarea ciclului inițial

6.3.6. Integrarea exemplelor pas cu pas

Exemplele pas cu pas sunt făcute pentru a-l ajuta pe utilizator să înțeleagă mai bine informația transmisă. Aplicația conține 6 exemple pas cu pas prin care sunt exemplificate:

- strategia folosită pentru algoritmul exact care se bazează pe Branch and Bound
- algoritmul lui Hierholzer reinterpretat pentru a determinarea un ciclu hamiltonian în loc de unul eulerian
- algoritmului de inserție a celui mai îndepărtat nod
- algoritmul de inserție a celui mai apropiat nod
- algoritmul de inserție cu cel mai mic cost
- algoritmul cel mai apropiat vecin

Exemplele sunt afișate în pagina folosind un carousel.

6.3.7. Integrarea sliderului pentru poze

Pentru a prezenta conceptele vizuale, informația bazându-se mai mult pe imagini, este folosit un glisor pentru poze.

```

1  <div #container id="container-slider">
2    <ng-image-slider
3      [images]=>images
4      [infinite]="false"
5      [autoSlide]="1"
6      [imageSize]={`${width: '100%', height: 400, space: 0}`}
7      (imageClick)=>closeSidenav()
8    </ng-image-slider>
9  </div>

```

6.4. Prezentarea aplicației

Aplicația este ușor de utilizat și funcționalitățile sunt intuitive pentru a oferi cea mai bună experiență utilizatorului.

În momentul în care accesează site-ul, utilizatorul este întâmpinat cu secțiunea prezentării generale a problemei comis-voiajorului.

Ușor de observat este bara de navigație care conține 4 elemente: primul este iconița de meniu (burger menu) care deschide și închide meniul. Utilizatorul are două posibilități, să îl închidă când nu interacționează cu el sau să îl lase deschis. A doua variantă vine cu avantaje întrucât vede în timp real subsecțiunea în care este în pagină și se poate muta ușor de la o secțiune la alta sau de la o subsecțiune la alta de același tip.



Figura 6.4. Meniu

Al doilea element din bara de navigație este un dropdown pentru schimbarea limbii aplicației. Opțiunile din care utilizatorul poate alege sunt limba română și limba engleză. Pentru a păstra consistența, prezentarea aplicației în scris și vizual se va face pe varianta în limba română. Apoi urmează titlul secțiunii curente și titlu problemei discutate.

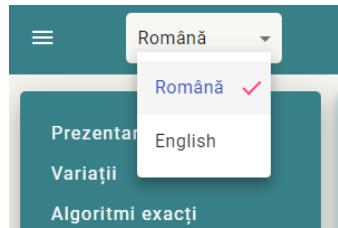


Figura 6.5. Dropdown pentru schimbarea limbii

Prezentare generală

Prezentarea problemei

Construire

$d(\pi) = \sum d(i, \pi(i))$ cu $i = \{1, \dots, n\}$

Așadar problema TSP este următoarea: *dat un graf neorientat complet, să se determine ciclul hamiltonian minim*. Această formă a problemei poartă numele de problema comis-viajorului simetrică, prescurtat sTSP, sau de TSP general.

Alte variații ale problemei se găsesc în [Sectiunea Variatii](#).

TSP general

Graful fiind neorientat, $d(i, j) = d(j, i)$ pentru orice $i, j \in V$, iar matricea este simetrică (egală cu transpusa sa). În analogie cu viața reală, problema poate fi văzută în felul următor: nodurile sunt destinații, iar muchiile sunt străzi cu specificația că nu există străzi cu sens unic (distanța este egală indiferent de direcția de mers). Se dorește drumul cel mai scurt.

The Traveling Salesman Problem (TSP) is one of the most well-known computational optimization problems, a problem of interest due to the fact that it is encountered in practice in various forms. It has a long history of attempts to solve it, finding the most efficient algorithm possible even though we will see that it is quite possible not to be able to create an algorithm that gives a solution in polynomial time, having shown that the problem is NP-complete.

Problem presentation

TSP involves finding the optimal route by which the travel agent can visit n cities, passing through each one only once and finally returning to the city from which he left. It can be considered as the optimal route the minimum distance or the minimum cost of the road (the problem does not change).

Construction

Let $G = (V, E)$ be an undirected graph for which V is the set of nodes (cities), E is the set of edges between nodes (roads between cities) and each edge has its own length provided the weight is non-negative. A weighted undirected graph can be viewed as complete by creating dummy edges between non-adjacent nodes of infinite length.

Any weighted graph can be stored using the distance matrix D of size $(n \times n)$ where for each i and j we have $d(i, j) = \text{edge length}(i, j)$. The distance from a node to itself is defined to be zero ($d(i, i) = 0$ for any $i \in V$). In the algorithms described in the following chapters, we will prefer to consider $d(i, i) = \infty$ to eliminate the need to test that the current vertices are distinct.

Presentation of the notion of Hamiltonian cycle and introduction to the traveling salesman problem

- A Hamiltonian cycle in a complete graph G is a cyclic permutation $(1, \pi(1), \dots, \pi^{n-1}(1))$ on the set of nodes $\{1, \dots, n\}$ in the graph: $1 \rightarrow \pi(1) \rightarrow \dots \rightarrow \pi^{n-1}(1) \rightarrow 1$.
- A minimal Hamiltonian cycle is a Hamiltonian cycle π such that the total length of the cycle is minimal:

$$d(\pi) = \sum d((i, \pi(i))) \text{ with } i = \{1, \dots, n\}$$

So the TSP problem is as follows: given a complete undirected graph, determine the minimal Hamiltonian cycle. This form of the problem is called the symmetric traveling salesman problem, abbreviated sTSP, or general TSP.

Other variations of the problem are found in [Variations section](#).

General TSP

The graph being undirected, $d(i, j) = d(j, i)$ for any $i, j \in V$, and the matrix is symmetric (equal to its transpose). In analogy to real life, the problem can be seen as follows: nodes are destinations and edges are streets with the specification that there are no one-way streets (the distance is equal regardless of the direction of travel). The shortest path is wanted.

Figura 6.6. Prezentarea unei secțiuni din aplicație în cele două limbi disponibile

Utilizatorul selectează, de exemplu, secțiunea *Algoritmi de aproximare*. În aceasta găsește text scris reprezentând teoria algoritmilor aduși în discuție, dar, de asemenea, informația este prezentată vizual prin grafuri numerotate pentru a ști la ce parte din teorie se referă și un ghid cu poze prin care este prezentat modul în care algoritmul parcurge arborele creând la final un ciclu hamiltonian.

Pași algoritm

- Primul pas al rezolvării problemei TSP este determinarea arborelui de acoperire de cost minim (MST, minimum-cost spanning tree) T . Acesta poate fi afiat, de exemplu, cu algoritmul lui Prim sau Kruskal.

Teoremă

Valoarea arborelui de acoperire de cost minim este cel mult valoarea ciclului TSP, adică $l(T) \leq OPT$.

Demonstrație: Dacă din ciclul hamiltonian OPT se elimină o muchie se obține un arbore parțial care are costul mai mare sau egal cu $l(T)$.

- Pentru a parcurge toate muchile din arborele de acoperire dublat în căutarea unui ciclu eulerian, se propune următoarea strategie cunoscută sub numele de parcurgere în adâncime (DFS, depth-first search):
 - Se ia un nod i din arbore
 - Dacă există o muchie de forma (i, j) nevizitată, se merge pe acea muchie, iar $i := j$. Se repetă pasul 2.
 - Dacă toate muchile care pornesc din i sunt vizitate, se dorește să se facă un pas în spate. Dacă i este nodul ales pasul 1, i nu are unde să se întoarcă, deci stop; altfel $i := k$ unde k este nodul în care se întoarce și se repetă pasul 2.
- Se folosește tehnica scurtăturilor (shortcutting) prin care din parcurgerea determinată anterior (în care muchile se repetă de două ori), se păstrează doar prima apariție a fiecărui nod, adică se sare peste nodurile deja vizitate

Figura 6.7. Pașii de rezolvare ai algoritmului arborelui dublu.

DFS care nu reține nodurile deja vizitate pentru a sări peste pasul cu shortcutting

```

Pseudocod Ciclu_hamiltonian(arbore)
se alege random un nod de start și se notează cu k
vizitat ← vector de lungime n inițializat cu 0
ciclu ← listă inițială vidă în care se va reține ciclul hamiltonian
Parcurgere(k, arbore)
    adaugă la ciclu nodul k      // muchia de întoarcere

Pseudocod Parcurgere(i, arbore)
vizitat[i] ← 1
adaugă pe i la ciclu
pentru fiecare j din lista arbore[i] adică fiecare j vecin al lui i execută
    dacă vizitat[j] = 0 atunci
        Parcurgere(j, arbore)
    
```

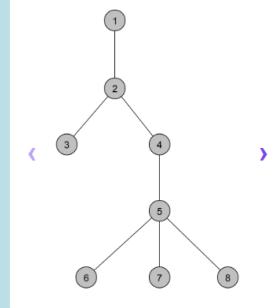


Figura 6.8. Pseudocodul metodei de determinare a unui ciclu hamiltonian împreună cu glisorul pentru exemplificare grafică.

În plus, în această secțiune este integrat un videoclip de pe YouTube ajutător utilizatorului pentru a înțelege modul de creare și conceptele discutate.

Pentru o bună înțelegere a modului de gândire și a pașilor prezentați, se recomandă vizualizarea următorului curs :)

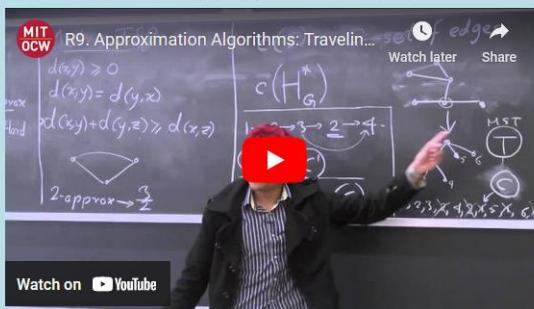
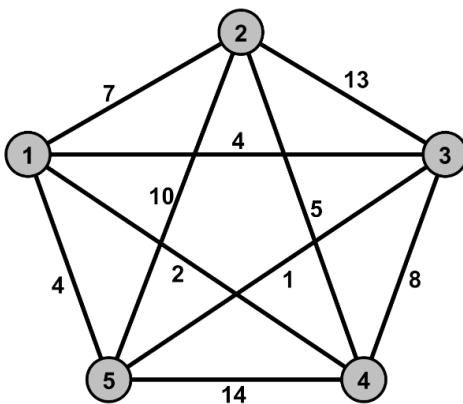


Figura 6.9. Integrarea în apliicație a cursului de algoritmi aproximativi [2]

Interesante sunt exemplele pas cu pas realizate pe baza unor algoritmi discuți. Utilizatorul poate astfel verifica faptul că și-a înșușit conceptele și este capabil să urmărească pașii de rezolvare. Astfel de exemple se găsesc în secțiunile *Algoritmi aproximativi* subsecțiunea *Algoritmul lui Christofides*, *Algoritmi exacti* subsecțiunea *Branch and Bound* și *Algoritmi euristici* în care există 4 exemple de acest tip pe baza algoritmilor Farthest Insertion, Nearest Insertion, Cheapest Insertion și Nearest Neighbor. În imaginile următoare sunt prezentați pașii algoritmului Nearest Neighbor:

Exemplu pas cu pas pe baza algoritmului nearest neighbor



VERDE: nodurile și muchile ce fac parte din ciclul final

PORTOCALIU: muchile de lungime minimă ce leagă fiecare nod nevizitat de ciclu

GRI: noduri nevizitate și toate muchile ce nu se încadrează la verde și portocaliu

Se setează

Ciclu : 1

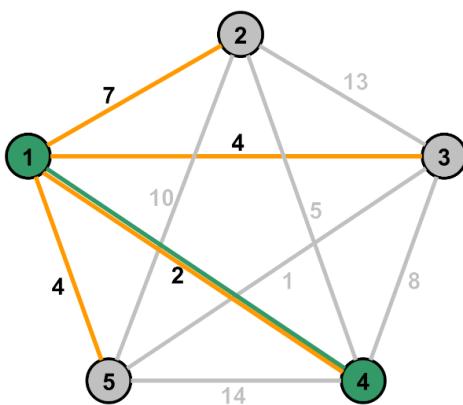
Soluție: 0

Precedent

Următorul

Resetează

Exemplu pas cu pas pe baza algoritmului nearest neighbor



Nod curent : 1

Nod nevizitat	Distanță
2	7
3	4
4	2
5	4

Nodul ales: 4

Cu distanță de la nodul curent 1 minimă (2)

Ciclu : 1,4

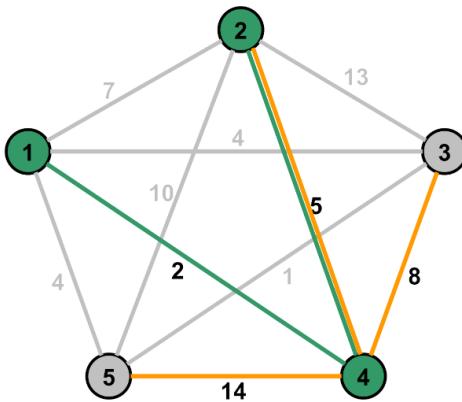
Soluție: 2

Precedent

Următorul

Resetează

Exemplu pas cu pas pe baza algoritmului nearest neighbor



Nod curent : 4

Nod nevizitat	Distanță
2	5
3	8
5	14

Nodul ales: 2

Cu distanța de la nodul curent 4 minimă (5)

Ciclu : 1,4,2

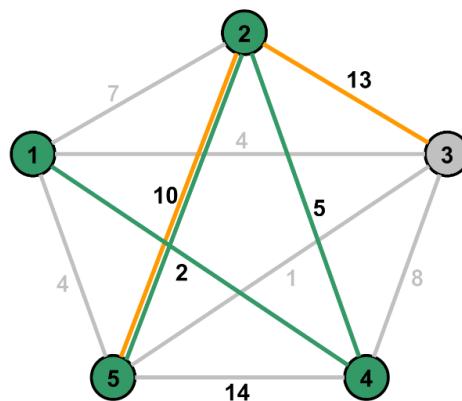
Soluție: 7

Precedent

Următorul

Resetează

Exemplu pas cu pas pe baza algoritmului nearest neighbor



Nod curent : 2

Nod nevizitat	Distanță
3	13
5	10

Nodul ales: 5

Cu distanța de la nodul curent 2 minimă (10)

Ciclu : 1,4,2,5

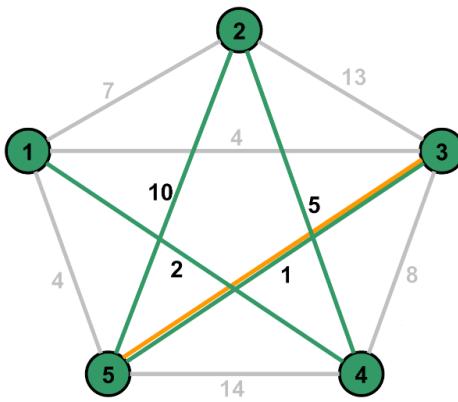
Soluție: 17

Precedent

Următorul

Resetează

Exemplu pas cu pas pe baza algoritmului nearest neighbor



Nod curent : 5

Nod nevizitat	Distanță
3	1

Nodul ales: 3

Cu distanța de la nodul curent 5 minimă (1)

Ciclu : 1,4,2,5,3

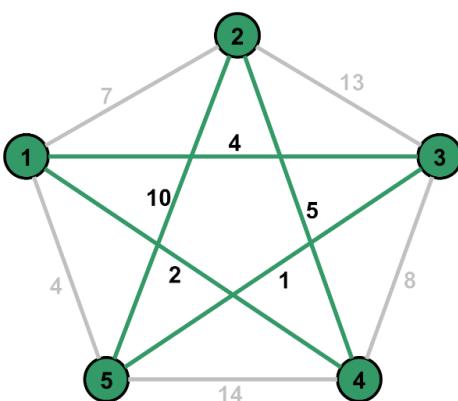
Soluție: 18

Precedent

Următorul

Resetează

Exemplu pas cu pas pe baza algoritmului nearest neighbor



Nod curent : 3

Întoarcere la nodul de start: 1

Distanța muchiei (3,1): 4

Ciclu : 1,4,2,5,3,1

Soluție: 22

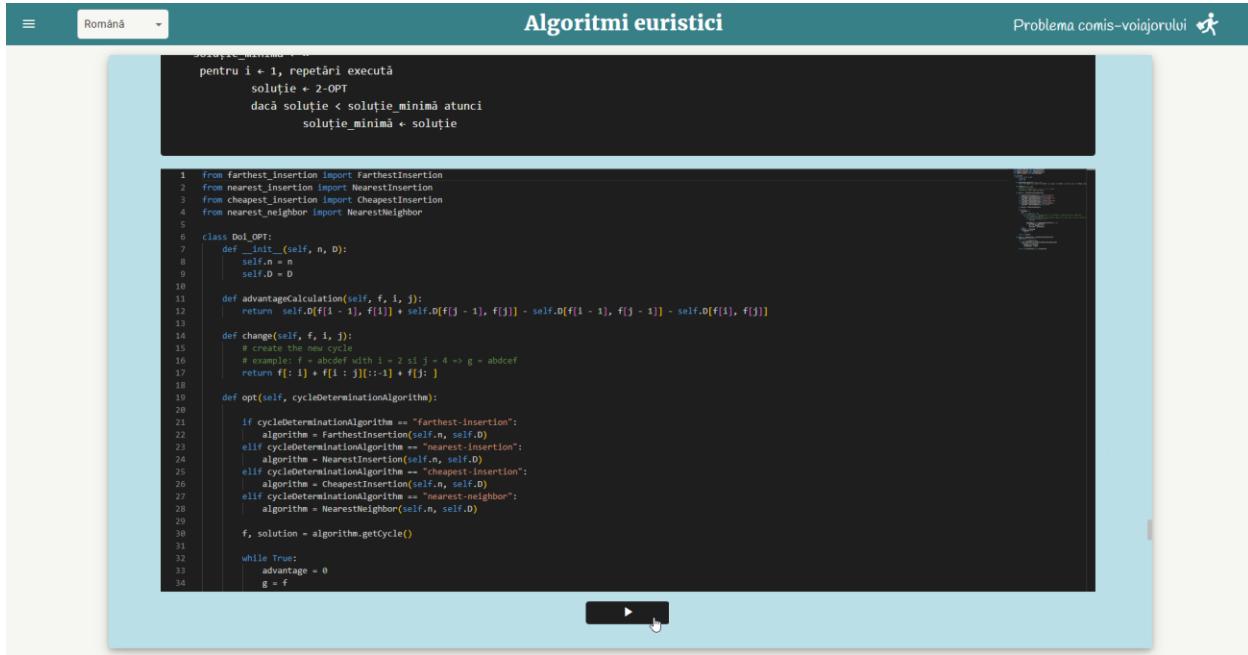
Precedent

Următorul

Resetează

Figura 6.10. Exemplu pas cu pas pentru algoritmul celui mai apropiat vecin.

Trecând peste secțiunea de exemple, utilizatorul are acces la cod implementat pe care îl poate rula în interiorul aplicației.



The screenshot shows a web-based application interface. At the top, there's a header bar with a menu icon, the language 'Română', and a logo for 'Problema comis-voiajorului'. Below the header is a dark panel containing the following Python code for the 2-OPT algorithm:

```

1  pentru i + 1, repetări execută
2      soluție ← 2-OPT
3      dacă soluție < soluție_minimă atunci
4          soluție_minimă ← soluție

1  from farthest_insertion import FarthestInsertion
2  from nearest_insertion import NearestInsertion
3  from cheapest_insertion import CheapestInsertion
4  from nearest_neighbor import NearestNeighbor

5
6  class D0_OPT:
7      def __init__(self, n, D):
8          self.n = n
9          self.D = D
10
11     def advantageCalculation(self, f, i, j):
12         return self.D[[i - 1], f[i]] + self.D[f[j - 1], f[j]] - self.D[f[i - 1], f[j - 1]] - self.D[f[i], f[j]]
13
14     def change(self, f, i, j):
15         # create the new cycle
16         # example: f = abcdef with i = 2 s.t. j = 4 => g = abcef
17         return f[:i] + f[i:j+1:-1] + f[j:]
18
19     def opt(self, cycleDeterminationAlgorithm):
20
21         if cycleDeterminationAlgorithm == "farthest-insertion":
22             algorithm = FarthestInsertion(self.n, self.D)
23         elif cycleDeterminationAlgorithm == "nearest-insertion":
24             algorithm = NearestInsertion(self.n, self.D)
25         elif cycleDeterminationAlgorithm == "cheapest-insertion":
26             algorithm = CheapestInsertion(self.n, self.D)
27         elif cycleDeterminationAlgorithm == "nearest-neighbor":
28             algorithm = NearestNeighbor(self.n, self.D)
29
30         f, solution = algorithm.getCycle()
31
32         while True:
33             advantage = 0
34             g = f

```

Below the code is a large button with a play icon and a mouse cursor pointing at it.

Figura 6.11. Secțiune de vizualizare cod pentru algoritmul 2-OPT

Apasă pe butonul *Testează codul* și se deschide secțiunea în care utilizatorul poate rula cod. Dacă apasă încă o dată pe același buton, secțiunea se închide.



Figura 6.12. Închiderea secțiunii de testare

Acesta poate alege tipul de input dorit și, dacă este cazul, alte opțiuni. Apoi apăsând pe buton pentru a rula codul pe inputul inserat și primește rezultatul corespunzător, respectiv mesajul întors de validare eşuată. În cazul rulării algoritmului 2-OPT, se afișează valoarea soluției, ciclul hamiltonian determinat și reprezentat într-un graf, ciclul obținut de euristică selectată și prima modificare a ciclului. Desigur, pentru alți algoritmi utilizatorul primește alte informații folositoare pentru cazul său.

Testează codul

Distanțe euclidiene

Ascunde testarea

farthest-insertion

nearest-insertion

cheapest-insertion

nearest-neighbor

Număr repetări: 1

Distanțe euclidiene înseamnă că se precizează pe prima linie coordonata Y. Se construiește un graf complet în care ponderați distanțele.

Introdu datele aici...

Soluție: 8878.8292
Ciclu hamiltonian: 1 -> 4 -> 2 -> 6 -> 8 -> 0 -> 5 -> 9 -> 3 -> 7 -> 1

Rulează

Ciclul hamiltonian obținut în urma rulării algoritmului heuristic greedy: 1 -> 4 -> 2 -> 6 -> 8 -> 0 -> 9 -> 5 -> 3 -> 7 cu distanță 9023.1391

Prima optimizare găsită transformă ciclul astfel. Din muchile (0, 9) și (5, 3) se aleg alte două muchii (0, 5) și (9, 3). Ciclul devine 1 -> 4 -> 2 -> 6 -> 8 -> 0 -> 5 -> 9 -> 3 -> 7 și soluția devine 8878.8292.

Figura 6.13. Secțiunea de inserare input și afișare răspuns

Exemplu de erori care pot fi întâlnite pentru inputul format din distanțe euclidiane:

10
0 1380 939
1 2848 96
2 3510 1671
3 457 334
4 3888 666
5 984 965
6 2721 1482
7 1286 525
8 2716 1432
9 738 1325

Already set the coordinates for node 7!

```

10
0 1380 939
1 2848 96
2 3510 1671
3 457 334
4 3888 666
5 984 965
6 2721 1482
7 1286 525
8 2716 1432
14 738 1325

```

Nodes are numbered from 0 to 9, but node 14 was found!

Figura 6.14. Exemplu de eroare

Dacă primește o eroare pe care nu o înțelege sau dorește să se informeze cu privire la modul în care este testat inputul pentru a-i verifica validitatea, utilizatorului i se oferă posibilitatea de a selecta secțiunea *Clasa Input* și a vedea în pagină codul sursă și explicații cu privire la cauzile de eroare.

Input Class

The problem of the traveling salesman

All the presented algorithms take as input data entered by the user. The input must be taken by an algorithm, processed and passed on to the classes that solve the TSP. The class that does this is called Input, it is the one that throws errors if the data entered does not correspond to what is necessary to solve the problem or the type of instance chosen by the user.

```

1 import math
2 import numpy as np
3
4 class Input:
5     def __init__(self, input, type):
6         self.type = type
7         self.n = len(inputList[0])
8         self.D = np.full((self.n, self.n), np.inf)
9
10    # enter data of the type (node 1, node 2, distance); matrix D can be symmetric (type 1) or asymmetric (type 2)
11    if self.type == 1 or self.type == 2:
12        if (len(inputList) - 2) % 3 != 0:
13            raise Exception("The entered input does not have the correct form!")
14
15        self.m = int(inputList[1])
16        self.data = []
17
18        for index in range(2, len(inputList), 3):
19            if int(inputList[index]) < 0 or int(inputList[index + 1]) < 0 or inputList[index + 2] < 0:
20                raise Exception("You entered at least one negative number!")
21
22            self.data.append([int(inputList[index]), int(inputList[index + 1]), inputList[index + 2]])
23
24    # enter data of the type (node, X coordinate, Y coordinate); the matrix D will always be symmetric
25    elif self.type == 3:
26        if (len(inputList) - 2) % 3 != 0:
27            raise Exception("The entered input does not have the correct form!")
28
29        if (len(inputList) - 1) / 3 != self.n:
30            raise Exception("The number of nodes is different than specified!")
31
32        self.m = None
33        self.data = [0] * self.n
34

```

List of possible errors and the reasons why they are raised

Errors that may occur regardless of the type of instance chosen

Figura 6.15. Clasa Input

7. Concluzii

Însumând informațiile prezentate în această lucrare, problema comis-voiajorului este o problemă computațională cunoscută și pentru care s-au creat de-a lungul timpului mulți algoritmi. Fiind NP-completă, nu există un algoritm care să ofere soluția optimă în timp polinomial pentru instanțe cu număr semnificativ de noduri.

Cu toate aceste, au fost dezvoltăți algoritmi care pot returna soluții bune pentru instanțe mari după cum s-a arătat în *Subcapitolul 5.2.* în care au fost rulate instanțe până la 7397 de noduri folosind algoritmi aproximativi (algoritmul arborelui dublu, algoritmul lui Christofides și algoritmul ce se bazează pe modul de rezolvare a algoritmului lui Christofides, dar în care se folosește o strategie greedy pentru a alege cuplajul perfect de cost cât mai mic). Acești algoritmi oferă soluții bune ce se află într-un interval de $1.50 * \text{OPT}$ de soluția optimă, acceptându-se instanțe mari. În *Subcapitolul 5.3.* sunt comparați algoritmii euristici 2-OPT și 3-OPT și aceștia sunt mai eficienți din punct de vedere a soluției rezultate, obținând valori mai mici de $1.15 * \text{OPT}$, dar se crește timpul de rulare, maximul întâlnit fiind de 3500 secunde pentru algoritmul 3-OPT știind că numărul nodurilor instanțelor variază între 51 (minim) și 400 (maxim). Iar în *Subcapitolul 5.4.* se ajunge la concluzia că algoritmul lui Christofides este cel mai potrivit pentru a rula TSP.

Referitor la partea practică, consider că ceea ce am realizat prin crearea aplicației informative pe tema *Problema comis-voiajorului* este util. Aplicația este intuitivă și vine în ajutorul unui utilizator care dorește să capete o înțelegere mai riguroasă, sinteza făcută fiindu-i ușoară de înțeles deoarece este structurată astfel încât să fie simplu de citit. În plus, se bazează pe reprezentări vizuale, fiind oferite o multitudine de exemple prin imagini și reprezentări pas cu pas a unor algoritmi. Învățarea îi este întregită prin expunerea, pe lângă teorie, a codului schițat

în pseudocod pentru fiecare algoritm și prin implementarea acestora în Python. Varianta de care dispune aplicația poate fi rulată, acesta fiind un beneficiu adus aplicației.

Desigur, există posibilități de dezvoltare care pot fi aduse aplicației. În primul rând, o funcționalitate interesantă care poate exista în acest site este să se poată modifica codul existent în pagină și să se ruleze. Chiar și fără ea, aplicația oferă utilizatorului toate informațiile necesare precum algoritmii discutați și clasa *Input* în care se preia inputul pentru a putea să ruleze local. O altă îmbunătățire ar fi introducerea mai multor algoritmi pentru problema TSP pentru a-i oferi utilizatorului o experiență completă. Iar ultima, pentru a fi accesibilă pentru mai multe persoane, se pot introduce mai multe limbi în care să se afișeze textul din aplicație. Această modificare este ușor de realizat, necesitând doar traducerea textelor și adăugarea limbilor noi în dropdownul din bara de navigație și în lista corespunzătoare.

Bibliografie

- [1] Angular.io. „What is Angular?” [Interactiv] URL: <https://angular.io/guide/what-is-angular>
- [2] Biswas, Amartya S. R9.Approximation Algorithms: Traveling Salesman Problem MIT 6.046J Design and Analysis of Algorithms, Spring 2015. URL: https://www.youtube.com/watch?v=zM5MW5NKZJg&ab_channel=MITOpenCourseWare
- [3] Dasgupta, S., C. H. Papadimitriou, and U. V. Vazirani. "Algorithms", July 18, 2006.
- [4] GitHub codul și documentația @ngstack/code-editor. URL: <https://github.com/ngstack/code-editor> [Accesat 7 June 2023]
- [5] GitHub. URL:<https://github.com/pallets/flask> [Accesat 7 June 2023]
- [6] Golden, Bruce, Lawrence Bodin, T. Doyle, and W. Stewart Jr. "Approximate traveling salesman algorithms." *Operations research* 28, no. 3-part-ii (1980): 694-711. URL: <https://doi.org/10.1287/opre.28.3.694>
- [7] Graphology, [Interactiv]. <https://graphology.github.io/> [Accesat 11 June 2023]
- [8] Gutin, Gregory, and Abraham P. Punnen, eds. *The traveling salesman problem and its variations*. Vol. 12. Springer Science & Business Media, 2006.
- [9] Ilavarasi, K., and K. Suresh Joseph. "Variants of travelling salesman problem: A survey." In *International conference on information communication and embedded systems (ICICES2014)*, pp. 1-7. IEEE, 2014. doi: 10.1109/ICICES.2014.7033850.
- [10] Jonker, Roy, and Ton Volgenant. "Transforming asymmetric into symmetric traveling salesman problems." *Operations Research Letters* 2, no. 4 (1983): 161-163. URL: [https://doi.org/10.1016/0167-6377\(83\)90048-2](https://doi.org/10.1016/0167-6377(83)90048-2)
- [11] Jungnickel, Dieter, and D. Jungnickel. *Graphs, networks and algorithms*. Vol. 3. Berlin: Springer, 2005.

- [12] Kindervater, Gerard AP, Jan Karel Lenstra, and David B. Shmoys. "The parallel complexity of TSP heuristics." *Journal of Algorithms* 10.2 (1989): 249-270. URL: [https://doi.org/10.1016/0196-6774\(89\)90015-1](https://doi.org/10.1016/0196-6774(89)90015-1)
- [13] Laporte, Gilbert. "The traveling salesman problem: An overview of exact and approximate algorithms." *European Journal of Operational Research* 59.2 (1992): 231-247. URL: [https://doi.org/10.1016/0377-2217\(92\)90138-Y](https://doi.org/10.1016/0377-2217(92)90138-Y)
- [14] Marinescu Ghemeaci, Ruxandra. Cursul "Metodat Branch and Bound". Algoritmi avansați. *Universitatea din București*, 2016
- [15] Nipuni Arunodi, „Top 10 Angular Component Libraries for 2022”, 12 October 2022 [Interactiv]. URL: <https://www.syncfusion.com/blogs/post/top-10-angular-component-libraries-for-2022.aspx> [Accesat 7 June 2023]
- [16] Pandey, Shourya. “Approximating Metric TSP” *Indian Institute of Technology Bombay*. 13 April 2019. URL: https://www.cse.iitb.ac.in/~rgurjar/CS759/Metric_TSP.pdf
- [17] Python.org. "PEP 3333 – Python Web Server Gateway Interface v1.0.1". URL: <https://peps.python.org/pep-3333/> [Accesat 7 June 2023]
- [18] Rafanavicius, Vytautas, Piotras Cimperman, Vladas Taluntis, Ka Lok Man, Gintaras Volkvicius, Martynas Jurkynas, and Justas Bezaras. "Efficient path planning methods for UAVs inspecting power lines." In *2017 International Conference on Platform Technology and Service (PlatCon)*, pp. 1-6. IEEE, 2017. doi: 10.1109/PlatCon.2017.7883704.
- [19] Roughgarden, Tim. *Algorithms Illuminated: Algorithms for NP-hard Problems*. Soundlikeyourself Publishing, LLC, 2020.
- [20] Seshadhri, C. “Edmond’s blossom algorithm” CSE202, Lec 2, Spring 2021. URL: https://www.youtube.com/watch?v=znLprRdvxII&list=PLOQjlWvnI0fbn9zAJfvJoQF1nc50KQR9g&index=3&ab_channel=C.Seshadhri [Accesat 7 June 2023]
- [21] SigmaJs, [Interasctiv]. URL: <https://www.sigmajjs.org/> [Accesat 7 June 2023]
- [22] Verma, Sanjay „Angular Image Slider for Lighbox”, 24 December 2022. [Interactiv]. URL: <https://www.npmjs.com/package/ng-image-slider?activeTab=readme> [Accesat 7 June 2023]
- [23] W3Schools, „TypeScript Introduction” [Interactiv] URL: https://www.w3schools.com/typescript/typescript_intro.php [Accesat 7 June 2023]
- [24] Korte, Bernhard H., Jens Vygen, B. Korte, and J. Vygen. *Combinatorial optimization*. Vol. 1. Berlin: Springer, 2011. DOI 10.1007/978-3-540-71844-4

[25] Wikipedia. [Interactiv] URL: [https://en.wikipedia.org/wiki/Flask_\(web_framework\)](https://en.wikipedia.org/wiki/Flask_(web_framework))

[Accesat 7 June 2023]

[26] Williamson, David P., and David B. Shmoys. *The design of approximation algorithms*.

Cambridge university press, 2011. URL: <http://designofapproxalgs.com/book.pdf>

[27] Predescu, Denisa. *Implementarea algoritmilor prezentăți în lucrare*. URL GitHub:

<https://github.com/denisapredescu/TSP-algoritmi>

[28] Predescu, Denisa. *Implementarea aplicației ce vizează Problema comis-voiajorului*. URL

Github: <https://github.com/denisapredescu/TSP-App>