Temă - Analiza Algoritmilor

Autor: Bloțiu Mihnea-Andrei - 323CA

Universitatea Politehnica București, Facultatea de Automatică și Calculatoare
mihnea.blotiu@stud.acs.upb.ro

Abstract. În această temă, urmează să analizez doi algoritmi exacți, doi algoritmi de aproximare și implicit să rezolv o problemă NP-completă, problema comis-voiajorului.

Keywords: Algoritm de aproximare · Problemă NP-completă · Problema comis-voiajorului.

1 Introducere

1.1 Descrierea problemei rezolvate

Problema Comis-Voiajorului este definită astfel: Fie G = (V, E) un graf neorientat complet în care oricare două vârfuri diferite ale grafului sunt unite printr-o latură căreia îi este asociat un cost strict pozitiv (costurile pentru sensuri contrare pot fi diferite). Cerinţa este de a determina un ciclu care începe de la un nod aleatoriu al grafului, care trece exact o dată prin toate celelalte noduri şi care se întoarce la nodul iniţial, cu condiţia ca acest ciclu sa aiba un cost minim. Costul unui ciclu este definit ca suma tuturor costurilor ataşate laturilor ciclului.

1.2 Exemple de aplicații practice ale acestei probleme

Prima aplicație practică vine chiar din denumirea problemei. Numele problemei provine din analogia cu un vanzator ambulant care pleacă dintr-un oraş, care trebuie să viziteze un număr de oraşe dat şi care apoi trebuie să se întoarcă la punctul de plecare, cu un efort minim (de exemplu timpul minim, caz în care costul fiecărei laturi este egal cu timpul necesar parcurgerii drumului). Astfel, putem să extindem această aplicație la ocupația curierilor din viața de zi cu zi ce pleacă dintr-un sediu și se întorc la finalul zilei în același loc după livrarea unui număr de colete în locații diferite. Alte exemple de aplicații practice:

1. Generarea traseelor urmate de dispozitivele de producere a circuitelor integrate;
2. Secvențierea unui genom (reconstruirea genomului pornind de la fragmente secvențiate între care există suprapuneri).

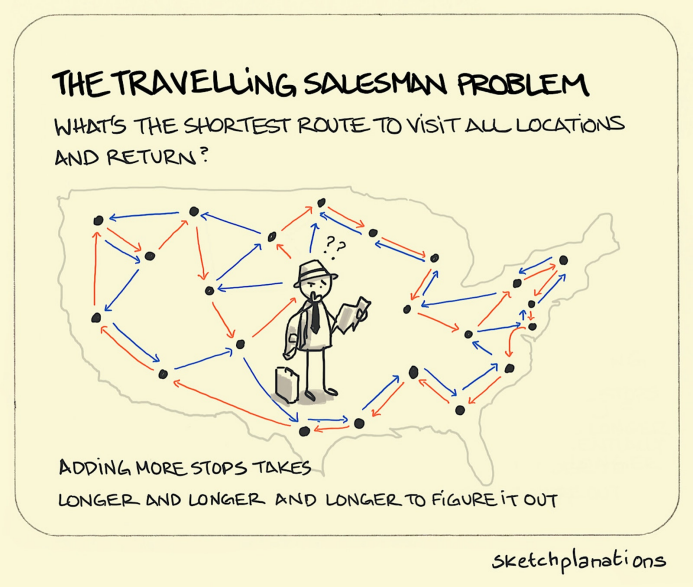

Fig. 1. Problema comis-voiajorului

1.3 Specificarea soluțiilor alese

Pentru rezolvarea acestei probleme am ales să analizez comparativ mai mulți algoritmi: doi dintre aceștia obțin întotdeauna rezultatul corect însă sunt foarte ineficienți (pentru mai mult de 15 noduri în graf, timpul de așteptare nu este unul agreabil), și încă doi care obțin un răspuns apropiat de cel corect și consumă o cantitate rezonabilă de resurse. Din punct de vedere formal, în varianta clasică problema este echivalentă cu cea a găsirii unui circuit Hamiltonian de cost minim într-un graf complet.

Acestea fiind spuse, problema va fi rezolvată după cum urmează:

1. exact, folosind varianta naivă (prin analiza tuturor circuitelor posibile, deci realizarea tuturor permutărilor posibile) pentru valori mici ale numărului de orașe, însă cum numărul de circuite este (n-1)!/2, complexitatea este O(n!);
2. exact, folosind metoda backtracking. Cum ruta este una ciclică, nu contează din ce nod pornim. Folosim primul nod ca punct de start și parcurgem graful în adâncime ținând de fiecare dată cont de costul minim posibi. Complexitatea este O(n!);
3. aproximativ, prin Greedy (metodă bună de complexitate $O(n^2 * \log_2 n)$, dar care nu pot fi considerată optimă deoarece nu întoarce rezultatul exact);
4. aproximativ, prin euristica Christofides, un algoritm care oferă în cel mai rău caz, un răspuns cu 3/2 mai mare față de cel optim. Din acest motiv, se mai numește și algoritmul 1,5-optim.

1.4 Euristicile aplicate pentru rezolvarea acestei probleme

Prin algoritm euristic vom înțelege un algoritm care furnizează soluții bune nu neapărat optime, care poate fi implementat rapid și furnizează rezultate in timp util.

Una dintre euristicile cel mai frecvent folosite este euristica Lin-Kernighan care se bazează pe ideea de a înlocui unele arce ale traseului cu altele în scopul reducerii costului. Cazul cel mai simplu este cel în care se înlocuiesc două arce ceea ce echivalează cu schimbarea ordinii de parcurgere a unei porțiuni din traseu (transformarea 2-opt).

Euristica Lin-Kernighan constă în a găsi o secvență de transformări de tip 2-opt și se bazează pe a căuta (utilizând o strategie de tip backtracking) secvența cea mai profitabilă de transformări. Spațiul de căutare este limitat la un număr relativ mic (de exemplu 5) de variante pentru fiecare etapă de transformare.

O altă transformare utilizată de către euristicile pentru PCV (unde PCV = problema comis-voiajorului) este cea bazată pe 4 interschimbări numită tehnica "podului dublu" prin care un traseu de forma [i1..i2][i3..i4][i5..i6][i7..i8] este transformat într-un traseu de forma [i2..i1][i4..i3][i6..i5][i8..i7].

Algoritmul lui Christofides urmează un profil similar, dar combină arborele minim de acoperire cu o soluție a unei alte probleme, cuplajul perfect de pondere minimă. Aceasta oferă un drum PCV care este de cel mult 1,5 ori mai lung decât optimul. Algoritmul lui Christofides a fost unul dintre primii algoritmi de aproximare, și are în parte meritul de a atrage atenția asupra algoritmilor de aproximare ca abordare practică la probleme greu de rezolvat.

1.5 Criteriile de evaluare pentru soluția propusă

Având în vedere faptul că în cele ce urmează încercăm să comparăm doi algoritmi care obțin soluția exactă cu doi aproximativi, criteriile pe baza cărora voi compara algoritmii menționați anterior vor fi complexitatea implementării cât și eficiența fiecăruia din punct de vedere al timpului de execuție, dar și al spațiului folosit. Toate acestea vor fi raportate obligatoriu la probabilitatea obținerii unui răspuns corect sau cât mai aproape de cel exact.

In situațiile în care pentru anumite probleme complexe, pentru a căror rezolvare nu se cunosc algoritmi, sau aceștia sunt ineficienți(timp, memorie, cost), se preferă utilizarea unor algoritmi care rezolvă problema dată mult mai rapid, cu efort mic, însă nu ne va furniza întotdeauna cea mai bună soluție, ci doar soluții acceptabile, adică soluții corecte care pot fi eventual îmbunătățite.

Acestea fiind spuse, pentru a testa corectitudinea algoritmilor, voi propune un set de teste care să se aplice parțial pentru fiecare dintre algoritmi precizați anterior după cum urmează:

1. 20 teste relativ mici (adică număr relativ mic de noduri și implicit de muchii) pentru algoritmii ce furnizează o soluție exactă.
2. 16 teste mai mari (adică număr considerabil mai mare de noduri și implicit de muchii fată de testele anterioare) pe care nu am putea rula algoritmii

exacți, pentru cele două variante de algoritmi euristici menționați anterior (10 pentru Greedy și 6 pentru Christofides);
3. 2-3 teste cu unul/două cazuri particulare (includem aici teste deosebite cum ar fi un graf care nu are niciun nod, un graf cu un singur nod, sau un graf în care toate distantele sunt egale);

2 Prezentarea soluțiilor

Această problemă este una dintre problemele faimoase cunoscute ca fiind NP-Hard. Asta înseamnă că nu există nicio soluție cunoscută care să o rezolve în timp polinomial.

2.1 Descrierea modului în care funcționează algoritmii aleși

2.1.1 Soluția Naivă - Generarea tuturor permutărilor - Soluție exactă Vom reprezenta graful printr-o matrice de adiacență, adică o matrice graph în care pe graph[i][j] vom reprezenta lungimea străzii respective deoarece așa cum am precizat inițial, graful trebuie să fie unul complet deci trebuie să avem muchie între oricare două noduri. Se poate ca graful să fie de fapt un digraf complet adică să aibă lungimi diferite în funcție de direcție.

Are la bază generarea tuturor permutărilor posibile ale orașelor și căutarea celei mai bune variante posibile. Funcționalitatea este una destul de simplă și este prezentată în cele ce urmează:

- Considerăm primul oraș (0) ca fiind puncul de pornire și cel în care ar trebui să ne întoarcem;
- Generăm toate cele (n-1)! permutări ale celorlalte orașe;
- Calculăm costul tuturor permutărilor și ținem cont mereu de permutarea ce a avut costul minim;
- Întoarcem permutarea cu costul minim

2.1.2 Soluția prin backtracking - Soluție exactă Vom reprezenta graful printr-o matrice de adiacență, adică o matrice graph în care pe graph[i][j] vom reprezenta lungimea străzii respective deoarece așa cum am precizat inițial, graful trebuie să fie unul complet deci trebuie să avem muchie între oricare două noduri. Se poate ca graful să fie de fapt un digraf complet adică să aibă lungimi diferite în funcție de direcție. De asemenea, în variabila answer vom reține lungimea traseului curent, iar în vectorul visited orașele din care este compus acesta.

Acum urmează partea de backtracking. În primul rând, se observă ușor că nu contează din ce oraș pornim, pentru că oricum trebuie să ne întoarcem de unde am plecat; de exemplu, traseul [3, 2, 5, 1] este tot una cu [5, 1, 3, 2]. Prin urmare, vom fixa pe visited[0] valoarea true (adică vom porni mereu din orașul 0).

Vom defini funcția de backtracking cu antetul void tsp_back(listă parametrii), unde pe lângă cei menționați anterior, „curr_pos" reprezintă poziția curentă din traseul pe care îl generăm, „nodes" este numărul total de noduri, „count" este numărul de noduri pe care l-am parcurs până acum, iar „cost" este costul parțial de până la momentul curent. Dacă nu am ajuns pe poziția „nodes", înseamnă că nu am terminat generarea. În acest caz, traversăm toți vecinii nevizitați ai nodului de pe poziția curentă, creștem numărătoarea cu 1 și costul total cu cel corespunzător acestui pas.

Are la bază generarea tuturor permutărilor posibile ale orașelor și căutarea celei mai bune variante posibile doar că de data aceasta prin backtracking. Funcționalitatea este una destul de simplă și este prezentată în cele ce urmează:

- Considerăm primul oraș (0) ca fiind puncul de pornire și cel în care ar trebui să ne întoarcem; Așa cum am spus și anterior, deoarece avem un ciclu, putem considera orice nod ca fiind cel de plecare
- Începem să parcurgem din sursă către vecini în adâncime (DFS);
- Calculăm costul fiecărei parcurgeri și ținem minte costul minim pe care îl tot actualizăm;
- Întoarcem permutarea cu costul minim

2.1.3 Soluția folosind Greedy - Soluție Aproximativă Vom reprezenta graful printr-o matrice de adiacență, adică o matrice graph în care pe graph$[i][j]$ vom reprezenta lungimea străzii respective deoarece așa cum am precizat inițial, graful trebuie să fie unul complet deci trebuie să avem muchie între oricare două noduri. Se poate ca graful să fie de fapt un digraf complet adică să aibă lungimi diferite în funcție de direcție.

În cazul metodei (strategiei) greedy apare suplimentar ideea de a efectua în acel moment o alegere. Dintre toate nodurile următoare posibile de a fi vizitate sau dintre toţi paşii următori posibili, se alege acel nod sau pas care asigură un maximum de "câştig", de unde şi numele metodei: greedy = lacom. Funcționalitatea este una destul de simplă și este prezentată în cele ce urmează:

- Primul pas înseamnă crearea a două structuri de date adică: a unei liste care tine indicii orașelor din matricea inițială de distanțe dintre orașe și dacă ruta respectivă a fost vizitată sau nu și a unui vector rezultat care conține toate orașele în ordinea în care au fost parcurse și poate fi afișat;
- Parcurgem toată matricea inițială și încercăm să luăm o decizie în vederea următorului nod pe care îl parcurgem. În acest caz, următorul nod va fi întotdeauna cel mai apropiat de poziția curentă. Deci, dacă un anumit cost de a ajunge în următorul oraș din orașul curent este minim, atunci acesta va fi următorul și actualizăm costul curent;
- Întoarcem permutarea și costul minim pe care îl afișăm.

2.1.4 Soluția folosind euristica Christofides - Soluție Aproximativă - 1,5optimă Pentru a varia metoda de transmitere a datelor de intrare, acum, sunt oferite N orașe prin coordonatele acestora. Se construiește un vector de

structuri cu aceste orașe. Apoi, matricea de adiacență cu aceeași semnificație ca și în algoritmii anteriori prin calcularea matematică efectivă a distanței dintre două puncte din plan caracterizate de coordonate.

Acest algoritm vizează lucrurile dintr-un punct diferit de vedere și anume. Dacă avem un graf Eulerian, putem găsi un ciclu Eulerian în timp liniar. Deci, dacă am avea un graf Eulerian cu orașele din TSP ca vârfuri atunci am putea găsi un ciclu Eulerian care să fie soluția problemei noastre TSP. Din inegaliatea triunghiului, știm că un ciclu TSP nu poate fi mai lung decât un ciclu Eulerian.

Totuși, pentru a găsi ciclul Eulerian optim este o problemă cel puțin la fel de grea ca rezolvarea propriu-zisă a TSP-ului. O variantă de a face asta este folosirea algoritmilor de asociere minimă.

Funcționalitatea este una destul de complexă și este prezentată în cele ce urmează:

- Se construiește un MST folosind algoritmul lui Prim (unde MST = Minimal Spanning Tree);
- Se construiește un subgraf format doar din nodurile de ordin impar din MST;
- Se formează un nou graf prin combinația muchiilor din subgraf și cele din graful inițial;
- Se calculează un circuit eulerian pe noul graf;
- Se scot din acest circuit muchiile care vizitează noduri deja vizitate și se întoarce răspunsul final;
- Se calculează timpul necesar finalizării acestui algoritm și se scrie traseul parcurs în fișierele de output.

2.2 Analiza complexității soluțiilor

Așa cum am precizat mai sus, fiecare dintre acești algoritmi va fi analizat din punct de vedere al complexității care cuprinde următoarele capitole: complexitatea timpului de execuție, al spațiului folosit, dar și complexitatea efectivă a implementării și înțelegerii unei astfel de soluții.

2.2.1 Soluția Naivă - Generarea tuturor permutărilor - Soluție exactă Această soluție poate fi considerată drept cea mai directă soluție deoarece nu facem altceva decât să încercăm toate permutările posibile ale orașelor și să verificăm care dintre acestea este cea mai scurtă/cea mai puțin costisitoare.

Acestea fiind spuse timpul efectiv de rulare al algoritmului este unul deloc fericit, care se află în $O(n!)$ și provine din generarea efectivă a tuturor permutărilor posibile.

Totuși există și două aspecte bune în cadrul acestui algoritm. Complexitatea spațială se poate afla în $O(1)$ dacă ne interesează doar distanța totală minimă parcursă sau în $O(n)$ dacă dorim să afișăm și traseul parcurs. Astfel, am avea nevoie de un vector în care să memorăm ordinea parcurgerii orașelor. De asemenea, simplitatea implementării soluției face această variantă foarte ușor de scris și de înțeles în orice limbaj și de către oricine.

2.2.2 Soluția prin backtracking - Soluție exactă În cel mai nefavorabil caz, numărul total de cicluri hamiltoniene ale grafului dat este $(n-1)!$. Asta se întâmplă atunci când graful este complet, adică în cazul nostru. Există cazuri în care, chiar și cu optimizarea făcută prin branch and bound, vom fi nevoiți să generăm toate aceste cicluri. De exemplu, acest aspect se întâmplă atunci când graful este complet și toate muchiile au același cost.

Optimizarea „branch and bound" înseamnă că, dacă la un apel al funcției lungimea curentă este mai mare sau egală cu lungimea minimă obținută până atunci, sigur acel început de traseu nu va duce la o soluție mai bună decât cea pe care o avem deja. În acest caz, putem ieși din apel printr-un simplu return.

Prin urmare, complexitatea algoritmului este $O(n!)$. Teoretic, este o complexitate foarte proastă, însă în practică (pe teste obișnuite, cu N până în 20 - 30), algoritmul se comportă destul de bine. Există și o soluție ce folosește programare dinamică, de complexitate exponențială, însă nu este foarte practică din cauza cantității mare de spațiu necesară.

Așadar, complexitatea temporală este: $O(n!)$, deoarece pentru primul nod sunt N posibilități de a merge mai departe, iar pentru al doilea sunt n-1 și tot așa.

Totuși și în acest caz există două aspecte bune în cadrul algoritmului. Complexitatea spațială auxiliară se poate afla în $O(n)$ deoarece avem nevoie și de un vector de N poziții în care să ținem minte mereu ce noduri au fost deja vizitate. De asemenea, simplitatea implementării soluției face această variantă foarte ușor de scris și de înțeles în orice limbaj și de către oricine, poate chiar mai facilă decât varianta naivă.

2.2.3 Soluția folosind Greedy - Soluție Aproximativă Acest algoritm caută optimul local și optimizează cea mai bună soluție locală pentru a găsi un optim global. Începe prin sortarea tuturor marginilor și apoi selectează marginea cu costul minim. Selectează în mod continuu cele mai bune alegeri următoare, în condițiile în care nu se formează bucle. Complexitatea de calcul a algoritmului greedy este $O(n^2 log_2(n))$ și nu există nicio garanție că se găsește o soluție optimă globală.

Totuși și în acest caz există un aspect bun al algoritmului. Complexitatea spațială auxiliară afla în $O(n)$ deoarece avem nevoie și de un map de N asocieri în care să ținem minte mereu ce noduri au fost deja vizitate și de asemenea de un vector de N poziții în care eventual să întoarcem ordinea de parcurgere a nodurilor.

În altă ordine de idei, dificultatea implementării soluției și greutatea de a demonstra corectitudinea acesteia face această variantă foarte greu de scris și de înțeles în orice limbaj și de către oricine.

2.2.4 Soluția folosind euristica Christofides - Soluție Aproximativă - 1,5optimă Algoritmul lui Christofides este cel mai bun algoritm cunoscut pentru soluționarea problemei comis-voiajorului pe caz general. Având în vedere toate argumentele menționate în secțiunile anterioare, acest algoritm întoarce cu

siguranță o variantă de răspuns care este cu cel mult 3/2 mai costisitoare față de cea optimă. Totuși, marele avantaj al acestei soluții este faptul că răspunsul este oferit în timp polinomial, cu o complexitate declarată de $O(n^4)$

Partea mai puțin plăcută a acestui algoritm este complexitatea extrem de mare a dificultății scrierii și înțelegerii acestui algoritm adică implicit a demonstrației că acest algoritm este într-adevăr corect și că oferă întotdeauna un răspuns cu maxim 3/2 mai slab fată de varianta optimă. De asemenea și din punct de vedere al spațiului ocupat este cel mai costisitor algoritm din cei patru analizați.

2.3 Prezentarea principalelor avantaje și dezavantaje pentru soluțiile luate în considerare

2.3.1 Soluția Naivă - Generarea tuturor permutărilor - Soluție exactă

2.3.1.1 Avantaje

- Principalul avantaj al acestei abordări de a rezolva problema comis-voiajorului este faptul că vom obține mereu soluția exactă a problemei;
- Complexitatea spațială este una foarte bună cu variante constante adică $O(1)$ sau liniare adică $O(n)$ în care dorim să întoarcem și traseul parcurs nu doar distanța totală;
- Ușurința cu care se implementează o astfel de soluție face această variantă destul de ușor de înțeles pentru oricine, chiar și persoane care nu sunt specializate în domeniul programării.

2.3.1.2 Dezavantaje

- Principalul dezavantaj al acestei abordări de a rezolva problema comis-voiajorului este complexitatea temporală dezastruoasă, care se află în $O(n!)$, așadar, aplicabilitatea sa într-o situație reală este restrânsă destul de mult de numărul de orașe ce pot fi luate în considerare.

2.3.2 Soluția prin backtracking - Soluție exactă

2.3.2.1 Avantaje

- Codul este unul foarte intuitiv;
- La fel ca orice problemă rezolvată prin backtracking, codul are doar câteva linii, chiar mai scurt față de varianta naivă a problemei;

2.3.2.2 Dezavantaje

- Principalul dezavantaj al acestei abordări, la fel ca și la cea anterioară este complexitatea temporală pe care o avem la dispoziție, aceasta aflându-se tot în $O(n!)$, așadar aplicabilitaea sa se restrânge cel puțin la fel de mult ca în cazul anterior;

– O altă problemă de luat în considerare ar fi faptul că vorbim despre un algoritm recursiv, aspect ce uneori poate crea foarte mari probleme deoarece în general este de preferat evitarea recursivității. Principalul argument aici este faptul că deși noi nu avem nevoie decât de $O(n)$ spațiu auxiliar, totuși, funcția fiind recursivă, la fiecare apel al acesteia, informațiile sunt puse în mod repetat pe stivă. Stiva fiind o zonă limitată de memorie, aceasta poate fi consumată foarte ușor prin recursivitate;
– Un alt dezavantaj al acestei metode ar fi dificultatea de a urmări algoritmul si mai exact de a îl simula de către o persoană.
– Se poate preciza și faptul că un astfel de algoritm este foarte greu de optimizat uneori fiind chiar imposibil. În acest caz, așa cum a fost precizat anterior, se poate face o optimizare de tip „branch and bound" însă chiar și așa uneori suntem nevoiți să generăm toate permutările.

2.3.3 Soluția folosind Greedy - Soluție Aproximativă

2.3.3.1 Avantaje

– Este foarte ușor de găsit o soluție Greedy de orice tip pentru această problemă;
– Complexitatea temporală $O(n^2 log_2 n)$ unde n = numărul de orașe a unui algoritm de tip Greedy este net superioară celorlalți algoritmi deoarece nu avem niciodată parte de recursivitate și nici de generarea tuturor permutărilor. Întotdeauna ne bazăm pe o decizie locală pe care dorim a o face universală;
– Având în vedere argumentul anterior, acest algoritm poate fi ușor folosit pe un număr semnificativ mai mare de orașe față de variantele exacte analizate anterior;
– Complexitate spațială foarte bună, cuprinsă în $O(n)$.

2.3.3.2 Dezavantaje

– Principalul dezavantaj al aceste soluții este faptul că nu oferă niciodată o soluție exactă, ba chiar mai mult de atât, soluția oferită de acesta este considerată sub-optimă, de foarte multe ori fiind departe de adevăr.
– Întâmpinăm o dificultate foarte mare în a demonstra dacă într-adevăr o soluție Greedy este într-adevăr corectă.

2.3.4 Soluția folosind euristica Christofides - Soluție Aproximativă - 1,5optimă

2.3.4.1 Avantaje

– Este considerat cel mai bun algoritm de rezolvare a PCV pe caz general de până acum;

– Deși acest algoritm este unul aproximativ, nu oferă soluția exactă, răspunsul acestuia este foarte bun, având un factor de aproximare de 1,5 motiv pentru care acest algoritm a fost numit 1,5-optim.
– Complexitatea temporală deși nu este deloc grozavă oferă un răspuns în timp polinomial $O(n^4)$ ceea ce îl face să poată fi folosi pe un număr foarte mare de orașe comparativ cu metodele exacte menționate mai devreme.

2.3.4.2 Dezavantaje

– Cel mai mare dezavantaj al acestui algoritm este dificultatea înțelegerii și a implementării, acestea fiind foarte ridicate comparativ cu tot ceea ce am analizat anterior.
– Argumentul anterior face demonstrarea corectitudinii acestui algoritm ca fiind una anevoioasă.

3 Evaluare

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

Încă de la început trebuie să aducem aminte faptul că modalitatea de citire a datelor a fost făcută variat, adică în două moduri pentru cei 4 algoritmi analizați.

– Prima variantă de citire a datelor se face după cum urmează:
  • se citește numărul de noduri ale grafului;
  • se citește o matrice de adiacență unde pe poziția i,j se află costul necesar parcurgerii muchiei de la nodul i la nodul j;

  și a fost aplicată pentru algoritmii: naiv, backtracking și greedy.
– A doua variantă de citire a datelor se face după cum urmează:
  • sunt oferite N orașe prin coordonatele acestora. Se construiește un vector de structuri cu aceste orașe.
  • se construiește matricea de adiacență cu aceeași semnificație ca și în algoritmul anterior prin calcularea matematică efectivă a distanței dintre două puncte din plan caracterizate de coordonate

  și a fost aplicată pentru algoritmul lui Christofides.

În ideea precizării anterioare se adaugă și faptul că algoritmii exacți nu pot fi testați pe un număr foarte mare de teste deoarece probabil că am aștepta zile/luni întregi și deci posibilitățile de testare a tuturor algoritmilor cu aceleași teste sunt inexistente.

Acestea fiind precizate, testele au fost gândite și generate după cum urmează:

3.1.1 Folderul in Conține 23 de teste variate care sunt aplicate pentru a testa algoritmii: naiv, backtracking și greedy deoarece aceștia au fost configurați pentru a avea modalitatea de citire a datelor identică. Testele sunt gândite după cum urmează:

- Testele 1-2: Teste de dimensiuni relativ mici (4-6 orașe) cu matrice simetrice.
- Testele 3-15, 18-22 : Cresc în dimensiuni (de la 2 până la 13 orașe) și nu reprezintă matrice simetrice.
- Testele 16, 17, 23: Reprezintă cazuri particulare de graf, naive însă care netratate pot cauza anumite probleme după cum urmează:
  - Testul 16: Graf fără niciun nod;
  - Testul 17: Graf cu un singur nod;
  - Testul 23: Graf cu toate costurile între toate laturile identice.

  simetrice.

3.1.2 Folderul tests_big_Greedy Conține două subfoldere:

- in - 10 teste cu număr mare de orașe folosite doar pentru algoritmul Greedy (20 - 50 orașe) deoarece pentru algoritmii exacți am aștepta foarte mult pentru a obține o soluție, deci nu este un timp rezonabil;
- out - cu răspunsurile corespunzătoare fiecărui test în parte.

Se vor rula testele în terminal după compilare folosind instrucțiunile din README.

Toate testele menționate anterior conțin drept distanță dintre orașe adică valorile din matricea de adiacență a grafului în intervalul $[0; 1000]$.

3.1.3 Folderul tests_Christofides Având în vedere faptul că modalitatea de introducere a datelor pentru algoritmul lui Christofides este diferită, aceste teste sunt special create pentru acest algoritm și nu pot fi rulate pe alți algoritmi. De asemenea, la acest algoritm avem teste foarte mari cu aplicabilitate mult mai mare în viața reală.

Conține 6 teste mari și foarte mari, de ordinul a 10.000 de orașe ce vor fi aplicate doar pe algoritmul Christofides după compilare folosind indicațiile din README.

De precizat este faptul că toate testele au fost generate random pentru precizările pe care le-am oferit până acum deoarece dorim să ne apropiem cât de mult de aplicația reală a acestui algoritm unde orașele sunt situate fără vreo regulă stabilită în prealabil.

Total teste generate: 39 teste

3.2 Menționarea specificațiilor sistemului de calcul pe care au fost rulate testele

- Procesor: AMD Ryzen 9 5900X 12-Core Processor @ 3.70 GHz;
- Memorie: G.Skill - 32GB RAM - 3200MHz - 2x Dual Channel;
- GPU: NVIDIA GForce GTX 1070;
- OS: Windows 11 - Pro.

3.3 Ilustrarea folosind grafice și tabele a rezultatelor evaluării soluțiilor pe setul de teste

În cele ce urmează voi centraliza timpii obținuți de fiecare dintre algoritmi pe testele gândite special pentru acestea în câteva tabele și de asemenea voi reprezenta grafic timpul de rulare al fiecărui algoritm în funcție de diferite teste de intrare.

Pentru determinarea timpilor de rulare, a fost folosită funcția *clock*() din biblioteca $< bits/stdc++.h >$

3.3.1 Tabele pentru ficare algoritm Tabel pentru varianta naivă:

Nr.Test	Timp Naive (secunde)	Numărul de orașe testate
1	$3.4 * 10^{-5}$	4
2	10^{-5}	6
3	$1.2 * 10^{-5}$	4
4	$1.5 * 10^{-5}$	5
5	$1.3 * 10^{-5}$	7
6	$2.8 * 10^{-5}$	8
7	0.000192	9
8	0.001707	10
9	0.001722	10
10	0.210277	12
11	2.99414	13
12	2.953	13
13	2.98647	13
14	2.9748	13
15	2.96172	13
16	0	0
17	$7 * 10^{-6}$	1
18	$1.3 * 10^{-5}$	2
19	$9 * 10^{-6}$	3
20	0.017851	11
21	2.96253	13
22	2.94341	13
23	$1.1 * 10^{-5}$	6

Table 1. Timpii de rulare ai algoritmului naiv pentru fiecare test

Tabel pentru varianta prin backtracking:

Nr.Test	Timp Naive (secunde)	Numărul de orașe testate
1	10^{-6}	4
2	$4 * 10^{-6}$	6
3	$1 * 10^{-6}$	4
4	$2 * 10^{-6}$	5
5	$1.7 * 10^{-5}$	7
6	$9.7 * 10^{-5}$	8
7	0.000858	9
8	0.007841	10
9	0.007818	10
10	0.958762	12
11	12.2579	13
12	12.2445	13
13	12.2954	13
14	12.1581	13
15	12.1418	13
16	0	0
17	0	1
18	0	2
19	$1 * 10^{-6}$	3
20	0.081335	11
21	12.1232	13
22	12.1692	13
23	$3 * 10^{-6}$	6

Table 2. Timpii de rulare ai algoritmului prin backtracking pentru fiecare test

Tabel pentru varianta prin greedy:

Nr.Test	Timp Naive (secunde)	Numărul de orașe testate
1	$1.1 * 10^{-5}$	4
2	$1.9 * 10^{-5}$	6
3	$1.3 * 10^{-5}$	4
4	$1.1 * 10^{-5}$	5
5	$1.1 * 10^{-5}$	7
6	$9 * 10^{-6}$	8
7	$1.2 * 10^{-5}$	9
8	$1.6 * 10^{-5}$	10
9	$7 * 10^{-6}$	10
10	$1.2 * 10^{-5}$	12
11	$2 * 10^{-5}$	13
12	$1.5 * 10^{-5}$	13
13	$1.3 * 10^{-5}$	13
14	$1.9 * 10^{-5}$	13
15	$1.4 * 10^{-5}$	13
16	0	0
17	0	1
18	$1.1 * 10^{-5}$	2
19	$1 * 10^{-5}$	3
20	$1.5 * 10^{-5}$	11
21	$1.8 * 10^{-5}$	13
22	$1 * 10^{-5}$	13
23	$1 * 10^{-5}$	6

Table 3. Timpii de rulare ai algoritmului prin greedy pentru fiecare test

Tabel pentru varianta prin euristica Christofides:

Nr.Test	Timp Naive (secunde)	Numărul de orașe testate
1	0.00081	76
2	0.004804	280
3	41.389	15112
4	0.000428	5
5	0.062213	1000
6	14.3187	10000

Table 4. Timpii de rulare ai algoritmului prin euristica Christofides pentru fiecare test

3.3.2 Grafice pentru fiecare algoritm

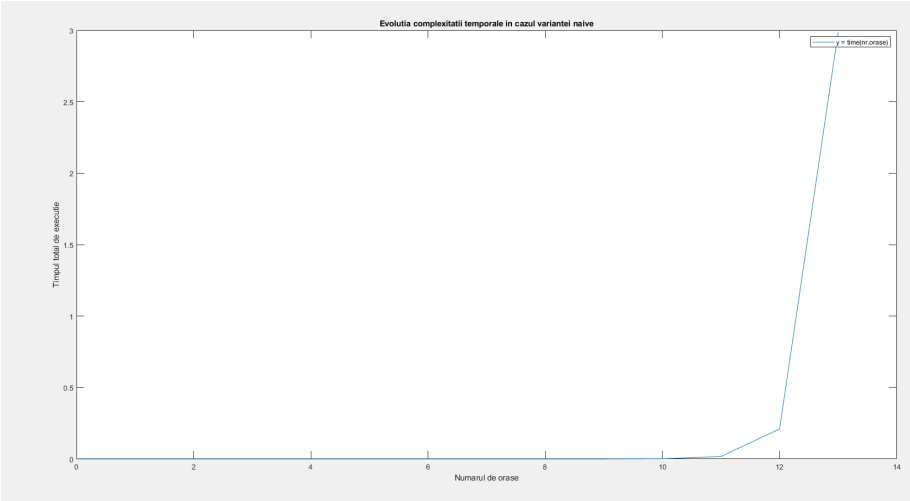

Fig. 2. Evolutia complexitatii temporale in cazul variantei naive

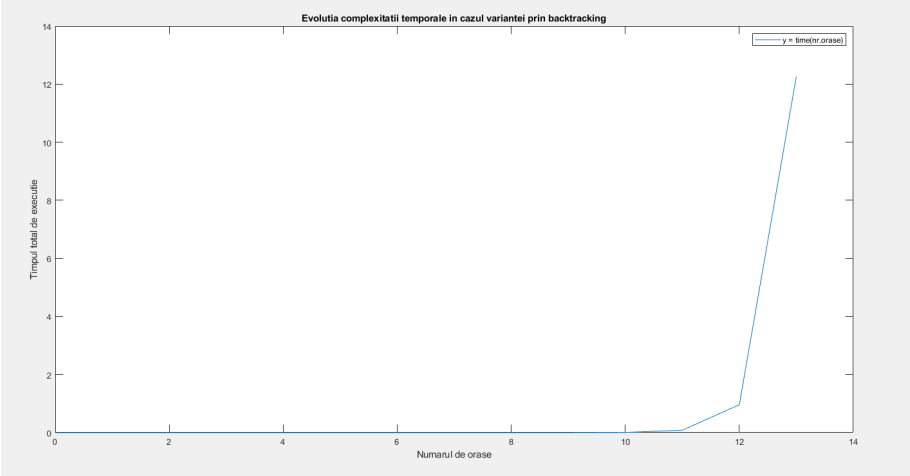

Fig. 3. Evolutia complexitatii temporale in cazul variantei backtracking

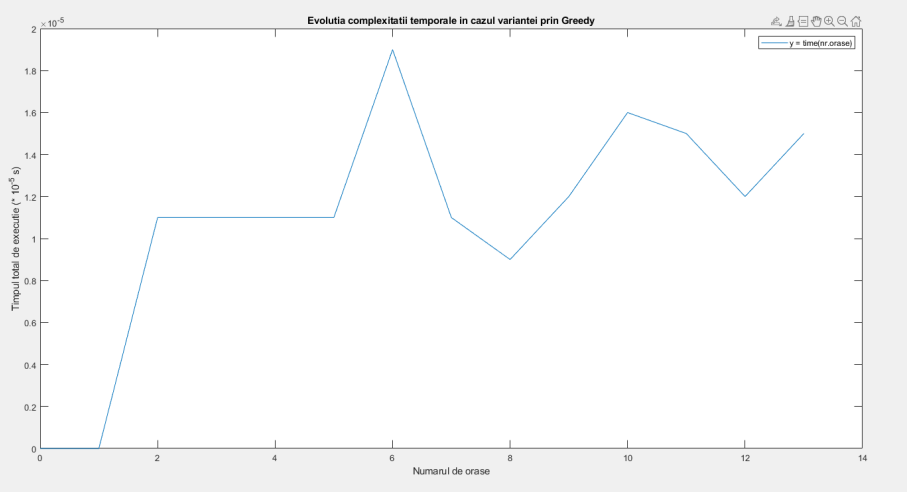

Fig. 4. Evolutia complexitatii temporale in cazul variantei Greedy

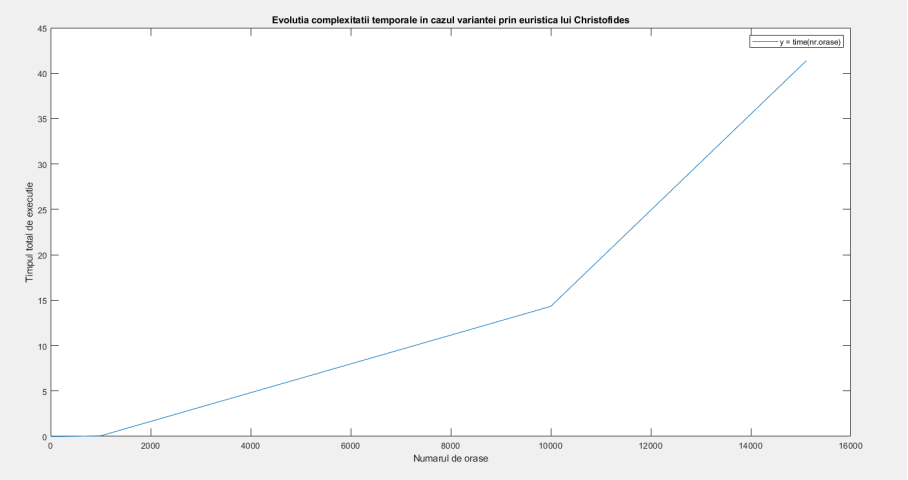

Fig. 5. Evolutia complexitatii temporale in cazul variantei prin euristica lui Christofides

3.4 Prezentarea succintă a valorilor obținute pe teste

Având în vedere toate specificațiile algoritmilor pe care le-am studiat anterior, testarea tuturor celor patru algoritmi analizați a funcționat conform așteptărilor.

Vom analiza pentru început cei 2 algoritmi exacți ce au avut același set de teste după cum urmează:

– Ambii algoritmi au evoluat „mai mult decât exponențial” odată cu mărirea dimensiunii testelor și deci implicit a numărului de orașe;

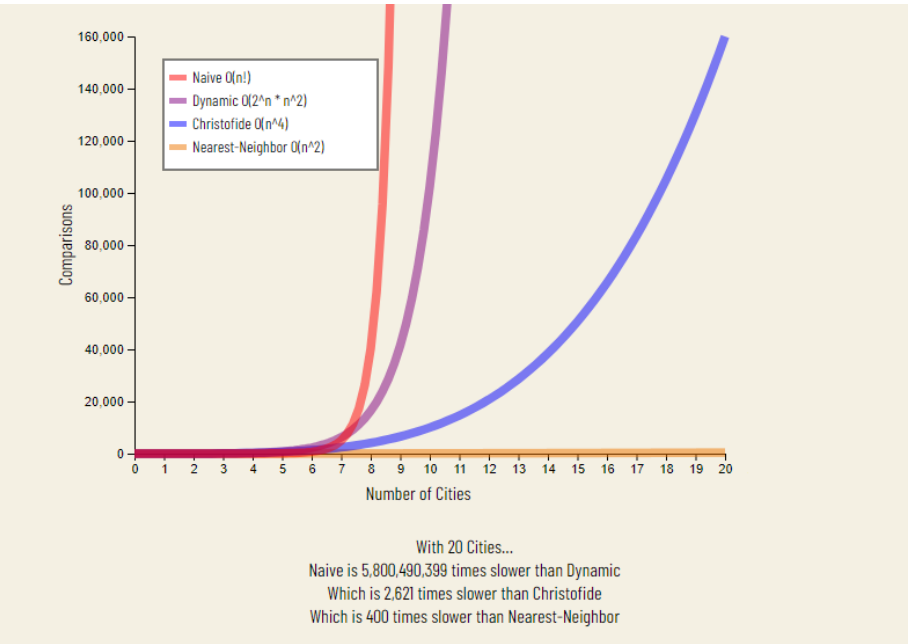

Fig. 6. Evolutia complexitatii temporale in cazul tuturor variantelor prin comparatie + Programare dinamica

- Putem observa faptul că pe măsură ce am avansat în testare și implicit au apărut teste mai mari, totuși, varianta naivă s-a descurcat mult mai bine față de cea prin backtracking (aproximativ 2 secunde față de 12 secunde pentru 13 orașe) aspect la care ne-am fi așteptat având în vedere că noi comparăm un algoritm iterativ cu unul recursiv;
- În abele cazuri a fost oferit un rezultat exact pentru un număr mic de orașe într-un timp rezonabil.
- Graficele 2 și 3 arată evoluția similară a celor doi algoritmi de-a lungul celor 23 de teste efectuate pe aceștia.

În legătură cu algoritmul Greedy, dacă analizăm figura 4, acesta pare că a evoluat destul de haotic însă având în vedere faptul că ordinul timpului de execuție a fost pentru toate teste de aproximativ 10^{-5}, o explicație posibilă ar fi erorile de calcul apărute efectiv în timpul rulării testelor.

De asemenea, acest algoritm a fost testat folosind și teste mult mai mari de până la 50 de orașe și deși acest lucru nu se poate observa în figura 4, timpul de execuție a avut același ordin și a evoluat conform graficului galben din figura 6 comparativ cu ceilalți algoritmi.

În ceea ce privește euristica nebanală analizată, cea a lui Christofides, acesta a răspuns conform așteptărilor fără nicio problemă. Așa cum se poate observa în tabelul 4, pentru teste de sub 1000 de orașe, s-a comportat mai mult decât

bine și am avut timpi de rulare de sub o secundă. De asemenea, dintre algoritmii analizați, acesta este și singurul ce a putut fi rulat pe un test cu peste 15.000 de orașe cu garanția de a avea un rezultat cu cel mult 3/2 mai mare fată de cel minim.

De asemenea, dacă urmărim graficul 5 și graficul albastru din figura 6 putem observa o evoluție similară în ambele cazuri, mult mai bună fată de cele exacte (ceea ce era de așteptat) însă și mai slabă fată de Greedy aspect ce nu este deloc surprinzător dacă aruncăm o scurtă privire asupra complexității algoritmilor.

4 Concluzii

4.1 Abordarea problemei în practică. Alegerea unui algoritm în funcție de situația oferită

Având în vedere tot studiul realizat anterior putem afirma cu siguranță faptul că nu există o soluție complet generală pentru rezolvarea problemei comisvoiajorului pe care să o putem alege indiferent de situație. De asemenea trebuie să ținem cont și de faptul că au fost analizați doar 4 algoritmi dintre care doi exacți și doi euristici. Există la momentul actual mulți alți algoritmi care încearcă să găsească soluții cât mai bune pentru a rezolva această problemă pe caz general, însă ea rămâne una dintre problemele nerezolvate ale lumii.

Acestea fiind spuse, pentru teste suficient de mici, care pot fi considerate de exemplu locațiile de livrare dintr-un oraș, este de ales întotdeauna o soluție exactă deoarece dorim să eficientizăm cât de mult se poate deplasarea. Totuși, soluțiile exacte au un timp de execuție suficient de mare și pot fi foarte greu îmbunătățite pe cazuri particulare mai puțin favorabile (ca de exemplu un graf complet care are toate costurile dintre noduri egale).

În acest moment suntem puși în situația de a alege un algoritm aproximativ. Deși ne-ar plăcea să avem un răspuns exact, nu avem această posibilitate. Dorim să facem un compromis astfel încât să putem afla un răspuns pentru setul de date propus însă să nu ne îndepărtăm foarte tare de soluția exactă.

Așadar, propunerea mea pentru soluționarea acestei probleme este alegerea întotdeauna a variantei naive și nu backtracking pentru teste relativ mici unde putem afla un răspuns exact, iar apoi euristica lui Christofides în defavoarea Greedy-ului pentru că deși avem un timp de răspuns mult mai mare, avem garanția de a obține întotdeauna un răspuns de tipul 1,5-optim. Din păcate această granție nu există pentru soluția Greedy motiv pentru care consider că nu este potrivită în niciun caz.

5 Referințe

1. https://www.youtube.com/watch?v=cY4HiiFHO1o
2. http://software.ucv.ro/ cmihaescu/ro/teaching/ACA/docs/PCV.pdf;
3. https://infogenius.ro/problema-comis-voiajorului-backtracking/
4. https://staff.fmi.uvt.ro/ daniela.zaharie/am2016/lab/lab2/algmet16_lab2.pdf;
5. https://blog.routific.com/travelling-salesman-problem
6. https://www.geeksforgeeks.org/travelling-salesman-problem-implementation-using-backtracking/
7. https://www.geeksforgeeks.org/travelling-salesman-problem-greedy-approach/
8. https://stemlounge.com/animated-algorithms-for-the-traveling-salesman-problem/
9. https://github.com/sth144/christofides-algorithm-cpp