

UNIVERSITATEA BABEȘ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA [Informatică Română]

LUCRARE DE LICENȚĂ

[Drumuri în graf folosind OpenMP]

Conducător științific
[Conf. dr. Niculescu Virginia]

Absolvent
[Cotoarbă Anamaria-Bianca]

2023

ABSTRACT

The area of algorithm analysis and assessment is the focus of this study. This research offers a substantial addition to computer science by investigating and developing testing as well as algorithmic analysis employing effective methods on graphs, such as finding the shortest path between any two nodes or breadth-first traversal in a graph. The goal of this project is to provide a tool that allows users to apply algorithms like Dijkstra, Bellman Ford, and BFS (Breadth First Search) on graphs. The project consists of 8 chapters, 5 of which cover the theoretical part and 3 for the practical part.

The project's idea, goal, and context are described in the theoretical section, along with the theories needed to understand the graphs, the purpose of the algorithms that must be executed within the application, the advantages of using the OpenMP library for parallelizing algorithms and an analysis of the metrics used to implement algorithms over various-sized graphs. A unique contribution is the creation of a set of data that highlights the need for parallelizing algorithms in comparison to sequential execution. The collected data demonstrates that as the size of the graph grows, parallel execution becomes more efficient.

The description of the application from the implementation perspective, examples of real-world situations where it can be used effectively, a clear separation of the architecture into client and server, implementations of these components, and the use of the user interface by picturing the website are all covered in the practical section. My own contribution to this section includes the scratch implementation of the sequential and parallel Dijkstra, Bellman Ford, and BFS algorithms, as well as my own design that was influenced by existing resources.

Due to the parallel execution type's superior efficiency versus sequential execution, the application was created to demonstrate the advantages of parallelizing algorithms. Dijkstra, Bellman Ford, and BFS were among the algorithms utilized, and the application was developed utilizing frameworks and libraries including OpenMP for parallel distribution using threads, Spring for connecting server and client sides, and Cytoscape for graph visualization.

Cuprins

I	Parte teoretică	1
1	Introducere	2
1.1	Contextul proiectului	2
1.2	Ideea și scopul proiectului	3
1.3	Structura proiectului	3
2	Teoria grafurilor și a drumurilor în graf	5
2.1	Introducere în teoria grafurilor	5
2.2	Introducere în drumuri în graf	7
3	Metodologie	9
3.1	Librăria OpenMP și framework-uri pentru probleme bazate pe grafuri	9
3.2	Algoritmi eficienți pentru modelarea rețelelor de drumuri	11
4	Analiza studiului și a performanței	13
4.1	Metrici ale performanței	13
4.2	Procesarea la scară largă a grafurilor folosind OpenMP	14
5	Concluzii	19
II	Parte practică	21
6	Utilizarea aplicației	22
6.1	Descrierea aplicației	22
6.2	Scenarii de utilizare	23
7	Arhitectura sistemului	24
7.1	Descrierea arhitecturii	24
7.2	Arhitectura clientului	24
7.3	Arhitectura serverului	25

8	Detalii de implementare	27
8.1	Implementarea protocolului de comunicație	27
8.2	Implementarea clientului	29
8.3	Implementarea serverului	33
8.4	Interfață de utilizare	41
	Bibliografie	45

Partea I

Parte teoretică

Capitolul 1

Introducere

1.1 Contextul proiectului

Acest proiect se încadrează în domeniul analizei și evaluării algoritmilor. Pentru explorarea și avansarea testării, precum și pentru analiza algoritmică, acest proiect contribuie semnificativ în domeniul informaticii, folosind metode eficiente asupra grafurilor, de pildă determinarea celor mai scurte drumuri între oricare două noduri sau parcurgerea în lățime într-un graf. Acest proiect își propune să ofere un instrument de aplicare a anumitor algoritmi precum Dijkstra, Bellman Ford și BFS (Breadth First Search) prin reprezentarea unei structuri de date de tip graf.

Componentele principale ale proiectului sunt partea de client, partea de server și partea de implementare a algoritmilor. Pe partea de client, s-a folosit o interfață de site web ușor de utilizat, care permite utilizatorilor să interacționeze și să testeze algoritmi Dijkstra, Bellman Ford și Breadth First Search. Această interfață oferă posibilitatea vizualizării grafurilor, a timpilor de execuție pentru fiecare algoritm, cât și detalii importante despre instrumentele și tehnicile folosite. Implementarea la nivelul clientului se bazează pe HTML, CSS și JavaScript, utilizând în același timp librăria Cytoscape.js pentru o interactivitate îmbunătățită.

Pe partea de server, proiectul depinde de framework-ul Spring, care stabilește o conexiune robustă între interfață și executabilele implementărilor algoritmilor. Această componentă are un rol crucial, deoarece asigură transferul eficient de date între backend și frontend.

Algoritmii sunt implementați în limbajul de programare C++. Pentru procesarea paralelă, este necesară folosirea unei librării precum OpenMP, care împarte munca între thread-uri și menține o bună comunicare între ele.

Această aplicație poate fi utilizată într-o varietate de domenii, cum ar fi domeniul transportului și al logisticii, ajutând la optimizarea rutelor, domeniul rețelelor sociale și al platformelor online, ajutând sistemele de recomandare pentru a sugera

utilizatorilor produse, servicii relevante, sau chiar și domeniul academic, servind ca un instrument de învățare pentru studenții care studiază algoritmi. Oferind o imagine de ansamblu asupra contextului proiectului, se pot aprofunda detaliile tehnice ale implementării, metodologia folosită pentru testare și evaluare, cât și analiza rezultatelor obținute în urma experimentării.

1.2 Ideea și scopul proiectului

Scopul acestui proiect este crearea unei aplicații pentru analiza tipurilor de execuție secvențial și paralel folosind algoritmi Dijkstra, BFS și Bellman Ford, cât și biblioteca OpenMP (Open-Multi Processing) care ajută la facilitarea paralelizării prin adăugarea de paralelism în bucle sau regiuni de cod. Ideea proiectului este de a dezvolta o aplicație flexibilă pentru analiza grafurilor de dimensiuni mici până la cele de dimensiuni mari, cât și evidențierea eficienței paralelizării algoritmilor, care poate fi utilizată în diverse situații.

Proiectul surprinde dezvoltarea unui site web axat pe testarea acestor algoritmi precizați cu ajutorul reprezentării sub formă de graf. Obiectivul principal al proiectului este de a compara și analiza performanța algoritmilor atunci când sunt implementați atât secvențial, cât și paralel.

Execuția algoritmilor va putea rula pe mai multe thread-uri folosind librăria OpenMP, permițând o analiză rapidă a grafurilor de diferite dimensiuni și complexități. Astfel, aplicația va permite utilizatorilor să exploreze rapid și precis o varietate de modele de grafuri și să genereze informații exacte și detaliate.

Scopul final al acestui proiect este de a dezvolta o aplicație eficientă și ușor de utilizat, care să ofere utilizatorilor informații clare și utile despre algoritmi și timpul de execuție al acestora, ajutându-i să perceapă nevoia de a utiliza distribuția paralelă față de cea secvențială.

1.3 Structura proiectului

Acest subcapitol descrie structura organizatorică a proiectului și oferă o perspectivă largă asupra elementelor cheie.

Structura proiectului constă din două componente principale: aplicația client și aplicația server. Interfața cu utilizatorul pentru interacțiunea cu sistemul este reprezentată de aplicația client, care va fi implementată sub forma unui site web, în timp ce aplicația server va fi responsabilă de procesarea și gestionarea datelor transmise de client și de executabilele algoritmilor.

Următoarele sunt principalele componente ale clientului:

- Interfața utilizator: Aceasta este partea vizuală a aplicației, care permite utilizatorilor să interacționeze cu sistemul prin intermediul unui site web, utilizând butoane, meniuri și permite vizualizarea statistică a diferenței între modul de lucru paralel și cel secvențial.

- Module pentru prelucrarea datelor: Aceste module sunt responsabile de procesarea datelor introduse de utilizator prin intermediul interfeței și de transmiterea acestora către server pentru prelucrare și generarea rezultatelor utilizând executabilele obținute în urma implementării algoritmilor Dijkstra, BFS și Bellman Ford. Acestea vor calcula cea mai scurtă rută între punctul de pornire 0 și restul nodurilor, precum și parcurgerea în lățime a grafului în cazul algoritmului BFS.

- Module de validare a datelor: Aceste module permit validarea datelor introduse înainte de a fi folosite în procesarea algoritmilor. De exemplu, se poate valida că s-a introdus un fișier înainte de a fi testat un algoritm.

Următoarele sunt principalele componente ale serverului:

- Module pentru stocarea datelor: Aceste module se ocupă cu gestionarea și stocarea datelor specifice aplicației, cum ar fi informațiile fișierelor.

- Module de comunicare: Aceste module conțin transferul de date între aplicația client și aplicația server. Pentru a realiza acest transfer, se va utiliza framework-ul Spring care oferă suport pentru diverse protocoale și tehnologii de comunicare, cum ar fi HTTP sau WebSocket.

O altă componentă esențială este implementarea algoritmilor. Aceasta a fost realizată în limbajul C++, folosind librăria OpenMP cu scopul paralelizării. S-a creat un proiect pentru fiecare metodă a fiecărui algoritm, ca ulterior să fie extrase executabilele lor pentru a putea fi procesate de server și pentru a returna rezultatul timpului de execuție.

Structura proiectului este concepută pentru a oferi o separare clară între aplicațiile client și server, permițând îmbunătățirea scalabilității și simplificarea implementării modificărilor într-un mod controlat, dar și o gestionare mai ușoară a componentelor individuale.

Capitolul 2

Teoria grafurilor și a drumurilor în graf

2.1 Introducere în teoria grafurilor

Teoria grafurilor este o ramură a matematicii care se ocupă cu studiul și analiza relațiilor dintre obiecte, oameni, locuri sau lucruri. Un graf este o structură de date formată dintr-o colecție de vârfuri, cunoscute și sub denumirea de noduri, și muchii care leagă aceste vârfuri. Muchiile reprezintă conexiunile dintre noduri și pot fi fie orientate, fie neorientate. Aceste muchii pot avea și valori numerice sau etichete specifice numite ponderi, ceea ce face ca graful să aparțină categoriei grafurilor ponderate.

O statistică folosită în mod obișnuit în teoria grafurilor este gradul unui nod. Gradul unui nod este numărul de muchii care se leagă de acesta. De exemplu, într-un graf al unei rețele sociale, gradul de nod al unei persoane reprezintă numărul de prieteni pe care îi are.

Teoria grafurilor poate fi utilizată în diverse domenii, cum ar fi informatică, biologie, fizică și economie. În informatică, teoria grafurilor este folosită pentru a modela rețele, cum ar fi internetul sau rețelele sociale.

Una dintre cele mai cunoscute probleme din teoria grafurilor este problema celor șapte poduri din Königsberg (2.1). În această problemă, obiectivul este de a găsi o plimbare prin orașul Königsberg care traversează fiecare dintre cele șapte poduri exact o dată. Aceste poduri formează muchiile unui graf. Problema a fost pusă pentru prima dată în secolul al XVIII-lea de către matematicianul Leonhard Euler, considerat drept “cel mai prolific matematician din istorie” conform [Tim23]. Soluția lui Euler la problemă este considerată pe scară largă ca una dintre lucrările fundamentale în domeniul teoriei grafurilor. Acesta a dovedit că este imposibil de găsit o astfel de plimbare.

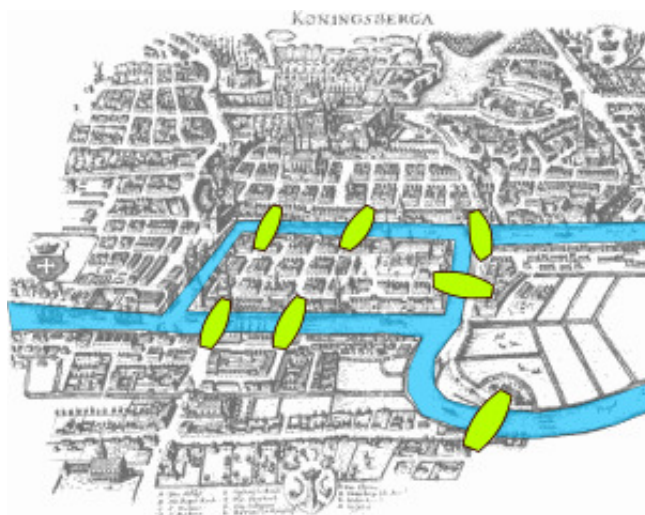


Figura 2.1: Hartă a orașului Königsberg care arată dispunerea actuală a celor șapte poduri, cu evidențierea râului Pregel și a podurilor [Ale22]

Teoria grafurilor joacă un rol esențial în analiza socială și în studiul rețelelor sociale. Prin reprezentarea conexiunilor dintre indivizi și entitățile sociale sub formă de grafuri, se pot analiza structurile comunităților, relațiile de influență, difuziunea informațiilor și alte fenomene sociale complexe. Un exemplu cunoscut în acest sens este analiza rețelelor sociale utilizată în studiul difuziunii informației pe platformele de socializare, cum ar fi Twitter sau Facebook. Prin aplicarea algoritmilor de propagare în grafuri, se poate observa cum se răspândesc știrile, ideile sau influența în cadrul acestor rețele.

Un caz concret a fost realizat de cercetători din departamentele de informatică și inginerie din diverse colțuri ale lumii, care prezintă modul în care știrile false răspândite despre COVID-19 pe mai multe platforme de socializare au afectat stabilitatea fizică și psihică a oamenilor, “The inflammable growth of misinformation on social media and other platforms during pandemic situations like COVID-19 can cause significant damage to the physical and mental stability of the people.”[SDD⁺22]. Astfel, ei au folosit instrumente și algoritmi de tip machine learning pentru a putea reprezenta diagrame legate de difuziunea informațiilor (2.2), care la rândul lor folosesc algoritmi asupra grafurilor.

Teoria grafurilor este utilizată și în domeniul bazelor de date, în special în bazele de date bazate pe grafuri. Acestea sunt sisteme de stocare și interogare care utilizează structura de graf pentru a reprezenta relațiile între entități. Prin utilizarea grafurilor orientate, se pot modela și accesa relațiile complexe dintre obiecte într-o manieră eficientă. Un exemplu cunoscut de bază de date bazată pe grafuri este Neo4j, care este folosită pentru stocarea și interogarea rețelelor sociale, a relațiilor produs-client în comerțul electronic și a altor structuri complexe de date.

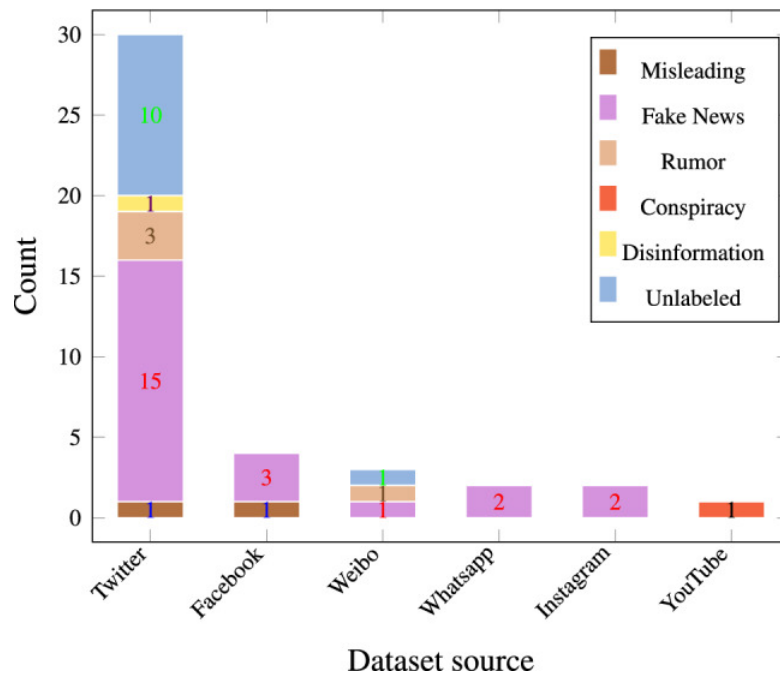


Figura 2.2: Variația datelor colectate de pe diverse platforme sociale [SDD⁺22]

2.2 Introducere în drumuri în graf

Drumurile în graf sunt un exemplu clasic de sistem care poate fi modelat folosind teoria grafurilor. În acest context, nodurile grafului reprezintă intersecții sau puncte de capăt, iar marginile reprezintă drumurile care le leagă. Modelând o rețea de drumuri sub formă de graf, se poate analiza și optimiza comportamentul acesteia într-o varietate de moduri. Spre exemplu, teoria grafurilor poate fi folosită pentru a identifica calea cea mai scurtă sau cea mai eficientă între două puncte dintr-o rețea de drumuri. Aceasta este cunoscută drept problema cu cea mai scurtă cale și este o problemă fundamentală în informatică. Prin identificarea celui mai scurt drum între două puncte, se minimizează timpul de călătorie și se reduce aglomerația drumurilor.

Drumurile în graf pot fi folosite și pentru a identifica noduri sau intersecții importante dintr-o rețea de drumuri. Acestea pot fi utilizate pentru a optimiza fluxul de trafic sau pentru a identifica potențiale blocaje sau vulnerabilități în rețea. De exemplu, un nod foarte central ar putea fi o locație ideală pentru un semn de circulație sau un sens giratoriu, în timp ce un nod cu centralitate scăzută ar putea fi bun pentru închiderea drumurilor.

Un fapt interesant despre drumurile în graf este că și rețelele de drumuri aparent simple pot avea proprietăți foarte complexe atunci când sunt privite ca grafuri. Un exemplu cunoscut este celebra „Teoremă a patru culori” care afirmă că orice hartă poate fi colorată cu doar patru culori astfel încât două regiuni adiacente să nu aibă aceeași culoare (2.3). Când este aplicată rețelelor de drumuri, această teoremă sugerează

rează că am putea folosi doar patru culori diferite pentru a distinge toate regiunile unei hărți rutiere, cum ar fi orașele sau țările.

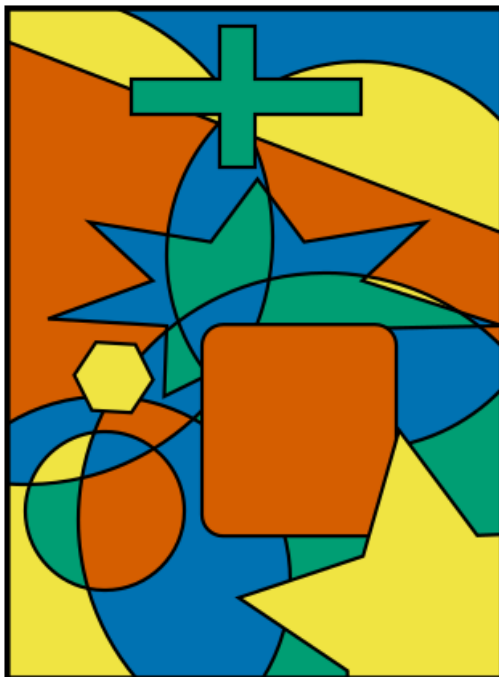


Figura 2.3: Exemplu de colorare a unei hărți conform teoremei [Sto]

În general, aceste exemple arată că și un lucru atât de simplu ca o rețea de drumuri poate avea proprietăți complexe atunci când este privit prin cazul teoriei grafurilor. Pe măsură ce sistemele de transport continuă să evolueze, teoria grafurilor și drumurile în graf rămân componente esențiale ale ingineriei și informaticii.

Capitolul 3

Metodologie

3.1 Bibliotecă OpenMP și framework-uri pentru probleme bazate pe grafuri

"The idea is you have some problem to solve (e.g finding the shortest path from your room to your first class), and you want to use your computer to solve it for you. Your primary concern is probably that your answer is correct (e.g you would likely be unhappy to find yourself at the wrong class). However, you also care that you can get the answer reasonably quickly (e.g. it would not be useful if your computer had to think about it until next semester)."

Potrivit [DRA16], accentul principal cade asupra obținerii soluției corecte la o anumită problemă. Însă, timpul necesar pentru a obține răspunsul este de asemenea important.

"The term "parallelism" or "parallel computing" refers the ability to run multiple computations (tasks) at the same time." [DRA16]

În contextul algoritmilor, utilizarea mai multor unități de procesare sau fire de execuție permite executarea simultană a sarcinilor. Această abordare de paralelizare permite distribuirea sarcinii de calcul, ceea ce poate duce la soluții mai rapide. Împărțind problema în subsarcini mai mici și procesându-le concomitent, algoritmi paraleli pot valorifica puterea procesoarelor moderne multicore sau a sistemelor de calcul distribuite. În consecință, algoritmi paraleli au potențialul de a reduce semnificativ timpul total de execuție pentru sarcinile intensive din punct de vedere computațional.

"Parallel computing has made a tremendous impact on a variety of areas ranging from computational simulations for scientific and engineering applications to commercial applications in data mining and transaction processing. The cost benefits of parallelism coupled with the performance requirements of applications present compelling arguments in favor of parallel computing." [GGKK03]

În lumea calculului paralel, librăriile și framework-urile reprezintă instrumente esențiale pentru dezvoltarea și implementarea eficientă a algoritmilor folosind paralelism. Când vine vorba de probleme bazate pe grafuri, cum ar fi analiza rețelelor de drumuri, planificarea rutieră sau procesarea de grafuri mari, utilizarea librăriei OpenMP și/sau a unui framework adecvat poate facilita paralelizarea și accelerarea operațiunilor asociate.

Librăria OpenMP este o implementare a standardului OpenMP care oferă suport pentru programarea paralelă în cadrul compilatorului GCC (GNU Compiler Collection) și a altor compilatoare compatibile. OpenMP permite programatorilor să specifice regiuni de cod care pot fi executate în paralel pe mai multe fire de execuție. Această abordare simplifică procesul de paralelizare a algoritmilor, inclusiv a celor care lucrează asupra grafurilor.

Utilizarea librăriei OpenMP în probleme bazate pe grafuri permite distribuirea sarcinilor și a datelor pe mai multe fire de execuție. În cadrul proiectului, algoritmi de parcurgere a grafurilor pot fi paralelizați folosind OpenMP pentru a explora diferite părți ale grafurilor în mod simultan. Astfel, se poate obține o accelerare semnificativă a timpului de execuție.

În conformitate cu [Boa21], beneficiile majore ale librăriei OpenMP sunt :

- Standardizarea: furnizează un standard pentru arhitecturi și platforme de tip “shared-memory” (sistem de calcul în care mai multe procesoare sau unități de procesare împart un spațiu de memorie fizică comun) .
- Productivitatea: stabilește un set limitat de directive simple, astfel se poate ajunge la un nivel semnificativ de paralelizare folosind doar 3 sau 4 directive (adnotare specială utilizată pentru a indica paralelismul și pentru a furniza instrucțiuni compilatorului sau sistemului de execuție).
- Portabilitatea: suportă limbajele de programare Fortran 77, 90 și 95 , C și C++.
- Ușor de utilizat: se poate realiza paralelizare incrementală și permite paralelism de nivel mic sau mare de granularitate.

OpenMP folosește modelul “fork-join” de execuție paralelă (3.1). Pașii modelului sunt:

- Programul OpenMP începe cu un singur thread numit master thread.
- Master thread-ul se execută secvențial până când întâlnește o regiune paralelă numită parallel region, când creează un set de thread-uri numit team of parallel threads(fork).
- Când regiunea paralelă se termină, thread-urile se sincronizează și își încheie execuția(join).

Pe lângă librăria OpenMP, există și framework-uri specializate pentru astfel de probleme, care oferă funcționalități suplimentare pentru analiza și procesarea grafurilor. Două framework-uri cunoscute sunt GraphX (framework pentru procesarea

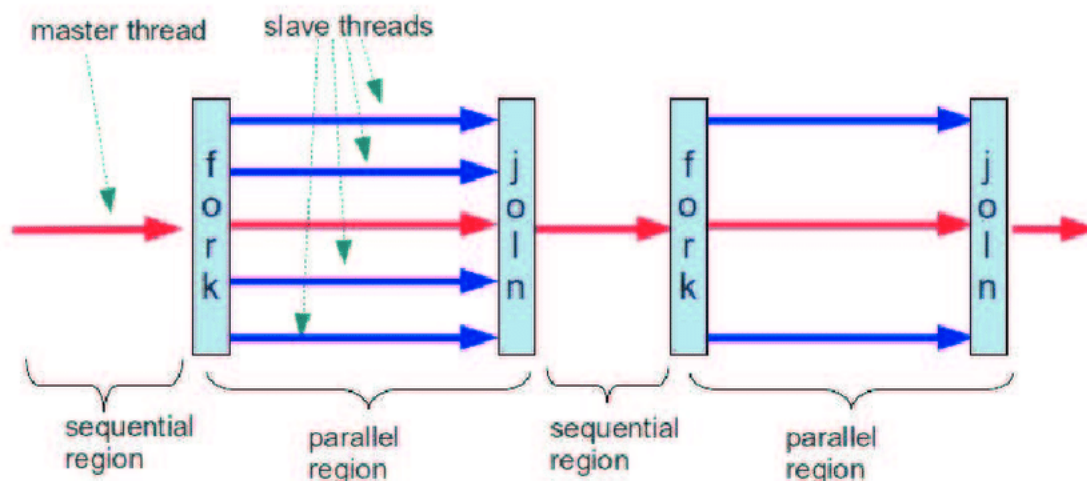


Figura 3.1: Modelul fork-join al OpenMP [Ha12]

grafurilor distribuite, dezvoltat de Apache Spark) și Giraph (framework de procesare a grafurilor distribuite, dezvoltat inițial de Facebook și ulterior donat către Apache Software Foundation). Aceste framework-uri integrează în mod transparent suportul OpenMP pentru a permite paralelizarea și procesarea eficientă a grafurilor pe mai multe noduri de calcul.

În acest sens, librăria OpenMP și framework-urile specializate au un rol important în dezvoltarea și implementarea eficientă a algoritmilor paraleli în probleme bazate pe grafuri. OpenMP facilitează programarea paralelă, în timp ce framework-urile specializate oferă abstracții și algoritmi optimizați pentru procesarea grafurilor.

3.2 Algoritmi eficienți pentru modelarea rețelelor de drumuri

Pentru a ilustra utilizarea librăriei OpenMP în reprezentarea grafurilor, se vor folosi diverși algoritmi pentru a identifica cele mai scurte drumuri între două puncte și parcurgeri ale drumurilor. Paralelizarea algoritmilor implică împărțirea grafului între thread-uri, astfel încât fiecare thread să facă operațiile specifice algoritmului pentru o parte a grafului. Apoi, prin comunicarea între ele, se actualizează și combină rezultatele parțiale pentru a obține rezultatul final.

Un algoritm clasic pentru căutarea celui mai scurt drum între două noduri este Dijkstra. Este un algoritm care funcționează eficient pe grafuri orientate și neorientate, fără muchii cu ponderi negative. Ideea din spatele algoritmului este explorarea iterativă a grafului, menținând un set de noduri vizitate și cele mai scurte distanțe ale acestora față de nodul sursă. În fiecare iterație selectează nodul nevizitat cu cea

mai mică distanță posibilă și actualizează distanțele nodurilor vecine dacă este găsit un drum mai scurt. Acest proces continuă până când toate nodurile sunt vizitate și sunt determinate cele mai scurte drumuri către toate nodurile accesibile pornind de la nodul sursă.

Datorită capacității și eficacității de găsim a celor mai scurte drumuri într-un graf, acesta a devenit un algoritm fundamental în teoria grafurilor și un instrument necesar pentru optimizarea problemelor cotidiene.

Pentru căutarea celor mai scurte drumuri între nodul sursă și toate celelalte noduri ale grafului, un algoritm cunoscut este Bellman Ford. Față de algoritmul Dijkstra, algoritmul Bellman Ford este capabil să găsească drumurile cele mai scurte și în cazul în care muchiile au ponderi negative. Acest algoritm funcționează prin actualizarea iterativă a muchiilor grafului. În fiecare iterație analizează toate muchiile și actualizează distanțele dacă este descoperită un drum mai scurt. Procesul se repetă de un maxim egal cu numărul nodurilor minus unu.

Bellman Ford este un algoritm versatil pentru găsirea celor mai scurte drumuri într-un graf care poate conține și ponderi negative. Deși are o complexitate de timp mai mare față de Dijkstra, acesta oferă avantajul de a prelucra anumite tipuri de graf pe care Dijkstra nu le poate.

Pentru a parcurge un graf într-o anumită ordine, un algoritm des folosit este BFS (Breadth-First Search). Acesta explorează toate nodurile în lățime, vizitând nodurile de la același nivel înainte de a trece la următorul nivel.

Algoritmul BFS menține o coadă de așteptare pentru nodurile care urmează a fi vizitate. Începe prin punerea nodului sursă în coadă și marcarea lui ca nod vizitat, apoi în fiecare iterație scoate câte un nod din coadă de așteptare, îl vizitează și pune în coadă toate nodurile vecine nevizitate. Procesul continuă până când coada devine goală de unde rezultă că toate nodurile au fost vizitate.

BFS este foarte utilizat pentru faptul că este un algoritm simplu și are o capacitate de a găsi cel mai scurt drum în grafurile neponderate.

Prin împărțirea grafurilor și comunicarea între thread-uri, se accelerează procesele de căutare și calcul, obținând rezultatele mai rapid decât o implementare secvențială. Utilizarea OpenMP în algoritmi precizați permite atingerea unei performanțe sporite.

Prin urmare, utilizarea OpenMP în reprezentarea drumurilor și aplicarea algoritmilor Bellman Ford, BFS și Dijkstra prin intermediul său aduc beneficii semnificative în ceea ce privește timpul de execuție, optimizarea resurselor de calcul și îmbunătățirea performanței aplicațiilor legate de gestionarea și analiza grafurilor.

Capitolul 4

Analiza studiului și a performanței

4.1 Metrici ale performanței

În cadrul proiectului, care implică implementarea algoritmilor Dijkstra, BFS și Bellman Ford într-o aplicație web care cuprinde testarea lor pe grafuri de diferite dimensiuni, este necesară o analiză a performanței pentru a evalua eficiența și scalabilitatea soluțiilor implementate. În acest capitol, se vor evidenția metricile de performanță utilizate și se va prezenta o analiză a rezultatelor obținute.

O parte dintre metricile de performanță analizate sunt:

- Timpul de execuție

Timpul de execuție este una dintre cele mai importante metrice ale performanței. Măsurarea timpului de execuție pentru fiecare algoritm implementat în funcție de dimensiunea grafurilor testate oferă o perspectivă asupra performanței soluțiilor. Compararea timpului de execuție între diferite dimensiuni ale grafurilor poate evidenția creșterea sau descreșterea eficienței algoritmilor pe măsură ce grafurile devin mai mari.

- Scalabilitatea

Scalabilitatea reprezintă capacitatea sistemului de a gestiona creșterea dimensiunii problemei sau a numărului de resurse utilizate. În contextul proiectului, evaluarea scalabilității se referă la capacitatea soluției de a procesa grafuri de dimensiuni foarte mari. Măsurarea timpului de execuție și compararea acestuia pentru diferite dimensiuni ale grafurilor poate oferi indicii despre scalabilitatea soluțiilor implementate.

- Eficiența

Eficiența este o măsură a utilizării eficiente a resurselor disponibile. În acest caz, eficiența se referă la raportul dintre timpul de execuție și numărul de resurse utilizate (procesoare, noduri de calcul etc.). Evaluarea eficienței poate implica măsurarea timpului de execuție pentru grafuri de aceeași dimensiune, dar cu un număr diferit

de resurse, și compararea rezultatelor obținute. O eficiență mai mare înseamnă că soluția utilizează resursele disponibile într-un mod mai controlat.

- Utilizarea resurselor de calcul

Măsurarea utilizării resurselor de calcul, cum ar fi procentul de utilizare a procesorului sau cantitatea de memorie utilizată va fi de folos în evaluarea eficienței soluției în utilizarea resurselor disponibile pe nodurile de calcul. O utilizare optimă a resurselor va contribui la obținerea unei performanțe mai bune și la evitarea congestiei resurselor.

Pentru a realiza analiza studiului și a performanței soluțiilor implementate în cadrul proiectului, se vor colecta date de performanță pentru fiecare algoritm implementat (Dijkstra, BFS și Bellman Ford) pentru diferite dimensiuni ale grafurilor testate.

Dacă algoritmii prezintă o performanță constantă sau chiar o îmbunătățire pe măsură ce grafurile devin mai mari, acest lucru poate indica o eficiență și scalabilitate mai bune.

De asemenea, se vor compara timpul de execuție și rezultatele obținute pentru fiecare algoritm (Dijkstra, BFS și Bellman Ford) pentru a identifica diferențele în performanță între aceștia. Se va analiza dacă un algoritm este mai rapid decât ceilalți în anumite situații sau dacă are o performanță mai bună pe grafuri de anumite dimensiuni. Această analiză va oferi informații utile despre alegerea algoritmului potrivit în funcție de caracteristicile problemei.

În concluzie, analiza studiului și a performanței permite evaluarea eficienței și scalabilității soluțiilor implementate în cadrul proiectului. Utilizând metrici de performanță precizate, se identifică punctele forte și posibilele îmbunătățiri ale aplicației. Aceste informații vor oferi o bază solidă pentru optimizarea și îmbunătățirea performanței soluției în viitor.

4.2 Procesarea la scară largă a grafurilor folosind OpenMP

Capitolul următor va explora conceptele și tehnicile asociate cu procesarea eficientă a grafurilor de dimensiuni mari utilizând librăria OpenMP.

Procesarea la scară largă a grafurilor reprezintă o provocare din cauza dimensiunii și complexității acestora. Librăria OpenMP oferă un model de programare paralelă puternic și scalabil, care poate fi folosit pentru a distribui sarcinile de calcul pe mai multe fire de execuție.

Un aspect important al procesării la scară largă a grafurilor este împărțirea acestora în subgrafuri și distribuirea lor către diferite thread-uri. Prin distribuirea echilibrată, se asigură o distribuție uniformă a sarcinilor de calcul și minimizarea comunicării între noduri.

Există diferite strategii de distribuție a grafurilor, precum distribuția blocului, distribuția ciclică sau distribuția pe bază de partiționare. Alegerea unei strategii potrivite depinde de caracteristicile grafului și de tipul de algoritmi care vor fi utilizați în procesare.

OpenMP oferă funcționalități potrivite pentru paralelismul de tip “shared-memory”, inclusiv regiuni paralele, sincronizarea thread-urilor și distribuția sarcinii de lucru.

Este important să se utilizeze strategii eficiente de paralelizare pentru a minimiza supraîncărcarea și pentru a asigura un flux de lucru eficient între thread-uri. Directive precum “omp parallel for”, “omp parallel sections” pot fi folosite pentru a paraleliza calculele și a exploata paralelismul în procesarea grafurilor.

Optimizarea performanței reprezintă un aspect important al procesării la scară largă a grafurilor. Există mai multe tehnici care se pot utiliza pentru a îmbunătăți performanța soluției implementate, precum:

- Minimizarea comunicării: reducerea a cât mai mult posibil a comunicării între thread-uri pentru a evita supraîncărcarea asociată transferului de date,
- Paralelizarea operațiilor: identificarea operațiilor care pot fi paralelizate și distribuirea între thread-uri pentru a beneficia de puterea de calcul distribuită,
- Optimizarea distribuției grafurilor: alegerea strategiilor de distribuție adecvate pentru a asigura un echilibru al sarcinilor și o comunicare eficientă între thread-uri,
- Utilizarea memoriei cache: optimizarea accesului la date prin utilizarea memoriei cache pentru a reduce timpul de acces la datele distribuite.

Procesarea la scară largă a grafurilor cu ajutorul OpenMP reprezintă o abordare benefică și necesară pentru analiza și prelucrarea grafurilor de dimensiuni mari. Distribuția grafurilor, comunicarea și sincronizarea corectă între thread-uri, precum și optimizarea performanței sunt elemente importante în obținerea unor rezultate rapide și precise.

Pentru a evidenția nevoia paralelizării algoritmilor, s-au realizat execuții ale algoritmilor Dijkstra, Bellman Ford și BFS pe un set de date cu același număr de noduri. Pentru a obține aceste rezultate, s-au folosit grafuri de două dimensiuni, cu 100 de noduri și cu 1000 de noduri.

Este necesară aplicarea algoritmului BFS asupra grafurilor de dimensiuni foarte mari, fiindcă este un algoritm de o complexitate scăzută, pentru a evidenția diferența de timp între secvențial și paralel. Comparând algoritmii Dijkstra și Bellman Ford, se remarcă faptul că Dijkstra are o execuție mai rapidă față de Bellman Ford, datorită explorării grafului într-un mod mai eficient.

În cele trei tabele de mai jos se regăsesc rezultatele obținute în urma folosirii algoritmilor din cadrul aplicației.

Tabela 4.1: Dijkstra

Tip execuție	Dimensiune graf	Timp execuție (ms)	Media aritmetică a timpilor (ms)
Secvențial	100 noduri	6 6 5 6 5 6 6 5 7	5,8
	1000 noduri	7500 8426 7537 7417 7451 7066 7270 7354 7408 7481	7491,00
Paralel	100 noduri	13 11 16 19 12 15 14 9 10 11	13,00
	1000 noduri	1198 1269 1166 1155 1172 1173 1199 1196 1148 1161	1183,70

Tabela 4.2: Bellman Ford

Tip execuție	Dimensiune graf	Timp execuție (ms)	Media aritmetică a timpilor (ms)
Secvențial	100 noduri	9 8 9 9 10 9 8 9 8 8	8,7
	1000 noduri	10302 9902 9671 10313 9904 9837 9906 9634 9755 9758	9898,20
Paralel	100 noduri	25 26 19 24 24 28 27 24 24 25	24,6
	1000 noduri	9895 10628 6077 10106 10047 10735 9740 6234 10119 10280	9386,10

Tabela 4.3: BFS

Tip execuție	Dimensiune graf	Timp execuție (ms)	Media aritmetică a timpilor (ms)
Secvențial	100 noduri	0 0 1 1 0 0 0 0 0	0,20
	1000 noduri	171 134 140 129 161 128 131 209 135 135	147,30
Paralel	100 noduri	1 1 1 1 1 0 1 1 0 1	0,80
	1000 noduri	129 133 211 147 145 132 142 157 132 132	146,00

Capitolul 5

Concluzii

În acest proiect, s-a propus dezvoltarea unei aplicații pentru evidențierea beneficiului paralelizării algoritmilor și funcționarea lor folosind tipul de execuție paralel, având o eficiență mult mai bună față de cel secvențial. S-au folosit algoritmi precum Dijkstra, Bellman Ford și BFS și s-a implementat aplicația folosind framework-uri și librării precum OpenMP pentru distribuirea paralelă folosind thread-uri, Spring pentru conexiunea între partea de server și partea de client și Cytoscape pentru vizualizarea grafurilor.

În scopul de a analiza și găsi cele mai scurte rute între noduri, a fost necesară implementarea algoritmului Dijkstra. Acest algoritm a determinat celui mai scurt drum necesar pentru a ajunge de la un punct de plecare la un punct de destinație, luând în considerare ponderile asociate muchiilor grafului.

Pe lângă algoritmul Dijkstra, algoritmi BFS (Breadth-First Search) și Bellman Ford au fost necesari în proiectul dezvoltat. Acești algoritmi au oferit posibilitatea de a explora și analiza grafuri din perspectiva structurii lor, identificând conexiunile și descoperind informații despre distanța sau ierarhia dintre noduri.

Pentru a îmbunătăți performanța algoritmilor, s-a utilizat librăria OpenMP pentru a distribui datele folosind thread-uri în cadrul implementării algoritmilor. Această abordare a evidențiat împărțirea sarcinilor de calcul între mai multe noduri, optimizând astfel timpul de răspuns și asigurând o gestionare eficientă a resurselor disponibile.

Cu privire la evaluarea performanței algoritmilor, s-au folosit diverse metrice relevante, precum măsurarea timpului de execuție al algoritmilor Dijkstra, BFS și Bellman Ford în diferite scenarii de testare, înregistrând timpul necesar pentru a găsi cele mai scurte drumuri și pentru parcurgerea în lățime a grafurilor.

De asemenea, s-a monitorizat utilizarea resurselor, cum ar fi consumul de memorie și utilizarea procesorului, în timpul execuției algoritmilor paralelizați. S-au comparat aceste rezultate cu execuția secvențială a algoritmilor pentru a evalua impactul paralelizării asupra performanței generale.

Implementarea aplicației client-server, cu partea de interfață bazată pe un site web, oferă o experiență intuitivă și ușor de utilizat utilizatorilor. Această abordare a facilitat interacțiunea cu sistemul și a permis utilizatorilor să găsească rapid cele mai scurte drumuri și căutarea în lățime.

În concluzie, proiectul demonstrează că utilizarea algoritmilor Dijkstra, BFS și Bellman Ford în combinație cu librării precum OpenMP poate oferi o soluție mai eficientă și scalabilă pentru analiza drumurilor în graf. Acest lucru poate fi aplicat în diverse domenii, cum ar fi planificarea rutelor de transport, optimizarea rețelelor de comunicații sau navigația în timp real.

Partea II

Parte practică

Capitolul 6

Utilizarea aplicației

6.1 Descrierea aplicației

Proiectul presupune dezvoltarea unei aplicații de analiză și testare a algoritmilor Dijkstra, Bellman Ford și BFS pentru a determina cel mai scurt drum între oricare două puncte, respectiv parcurgerea în lățime a grafurilor. Această aplicație este folosită pentru a evidenția diferența timpului de execuție între programarea paralelă și cea secvențială, cât și pentru observarea eficacității modului paralel de lucru.

Obiectivele aplicației sunt:

- Implementarea algoritmilor Dijkstra, Bellman Ford și BFS.
- Testarea algoritmilor pe grafuri de diverse dimensiuni.
- Vizualizarea grafurilor și modul de funcționare al algoritmilor asupra lor.
- Evaluarea performanței implementărilor secvențiale și paralele.
- Oferirea informațiilor despre punctele forte și punctele slabe ale fiecărei implementări.
- Facilitarea aplicațiilor din lumea reală a algoritmilor prin arhitectura client-server.

Aplicația utilizează o arhitectură client-server, în care tehnologiile client și server joacă roluri importante.

Partea clientului este responsabilă pentru furnizarea unei interfețe intuitive și ușor de utilizat pentru ca utilizatorii să interacționeze cu aplicația. Sunt utilizate tehnologiile web moderne precum HTML, CSS și JavaScript, pentru a crea o interfață de utilizator dinamică, interactivă și receptivă.

Partea serverului este construită folosind framework-ul Spring, un framework popular bazat pe limbajul de programare Java pentru construirea de aplicații web scalabile. Acesta permite gestionarea eficientă a solicitărilor primite de la clienți, procesarea calculelor algoritmice și răspunsul cu rezultatele obținute.

Testarea algoritmilor Dijkstra, Bellman Ford și BFS pe grafuri este semnificativă

din mai multe motive:

- Performanța algoritmului: analizând performanța acestor algoritmi, se obțin informații despre eficiența și scalabilitatea lor. Aceste cunoștințe ajută la selectarea algoritmului adecvat pentru probleme specifice legate de graf, asigurând timpul optim de execuție.
- Aplicații din lumea reală: algoritmi testați au numeroase aplicații în diverse domenii, cum ar fi rețelele de transport, analiza rețelelor sociale și sistemele de recomandare. Înțelegerea punctelor forte și limitărilor lor permit aplicarea eficientă în scenarii din lumea reală.
- Implementări secvențiale și paralele: Prin implementarea atât secvențială, cât și paralelă, aplicația permite compararea performanțelor acestora și evaluarea beneficiilor paralelizării algoritmilor.

6.2 Scenarii de utilizare

Diversele scenarii de utilizare ale aplicației cuprind testarea algoritmilor Dijkstra, Bellman Ford și BFS pe grafuri. Aceste scenarii evidențiază modul în care aplicația poate fi utilizată în setări practice, prezentându-și capacitățile și beneficiile într-o arhitectură client-server. Prin înțelegerea acestor scenarii de utilizare, se obține o perspectivă mai profundă asupra valorii aplicației.

Un scenariu de utilizare este analiza comparativă a algoritmului. Astfel, cercetătorii și dezvoltatorii pot utiliza aplicația pentru a evalua eficiența și scalabilitatea acestor algoritmi pe diferite grafuri. Efectuând teste și analizând rezultatele, utilizatorii pot lua decizii cu privire la selecția algoritmului pe baza cerințelor și constrângerilor specifice.

Un alt scenariu de utilizare este validarea algoritmilor. Utilizatorii pot verifica corectitudinea și acuratețea implementărilor lor algoritmice comparând rezultatele generate de aplicație cu rezultatele așteptate. Acest proces de validare ajută la identificarea și rezolvarea oricăror potențiale probleme sau discrepante, asigurând fiabilitatea algoritmilor.

Un potențial scenariu de utilizare este ca aplicația să joace rolul unui instrument educațional. Profesorii și studenții pot folosi implementările secvențiale și paralele ale algoritmilor Dijkstra, Bellman Ford și BFS ca exemple pentru a înțelege principiile și funcționalitatea lor. Prin interacțiunea cu aplicația, utilizatorii pot câștiga experiență practică și pot dezvolta o înțelegere mai profundă a comportamentelor și caracteristicilor de performanță ale acestor algoritmi.

De la analiza și optimizarea algoritmilor până la scopuri educaționale și de cercetare, aplicația se adresează unei game largi de scenarii de utilizare în domeniul algoritmilor bazați pe grafuri.

Capitolul 7

Arhitectura sistemului

7.1 Descrierea arhitecturii

Arhitectura aplicației este concepută pentru a oferi o privire de ansamblu a componentelor și pentru a prezenta o modalitate scalabilă în testarea algoritmilor Dijkstra, Bellman Ford și BFS.

Obiectivele principale ale arhitecturii sunt:

- Asigurarea unei comunicări eficiente între componentele client și server.
- Facilitarea testării și evaluării algoritmilor fără întreruperi pe diferite tipuri de graf.
- Oferirea scalabilității și optimizarea performanței pentru gestionarea procesării grafurilor de dimensiuni mari.

Prin proiectarea unei arhitecturi bine structurate, se propune crearea unei aplicații flexibile, care poate fi întreținută și capabilă să răspundă diverselor nevoi de testare și analiză a algoritmilor.

7.2 Arhitectura clientului

Arhitectura clientului este responsabilă cu furnizarea unei interfețe de utilizator interactive și ușor de folosit. Utilizează WebStorm ca mediu de dezvoltare integrat (IDE) pentru codificare, depanare și testare. Componentele clientului sunt dezvoltate folosind HTML, CSS și JavaScript (JS) ca limbaje de programare, oferind o experiență dinamică utilizatorilor. De asemenea, pentru o bună vizualizare a grafurilor, se regăsește în cadrul clientului și librăria Cytoscape.js care include o gamă largă de funcționalități asupra vizualizării și parcurgerii grafurilor.

HTML (Hypertext Markup Language) este limbajul standard utilizat pentru structurarea și prezentarea conținutului web. Acesta definește structura și aspectul paginilor web. Elementele HTML sunt utilizate pentru a crea o ierarhie logică, permițând

organizarea conținutului și a componentelor.

CSS (Cascading Style Sheets) este un limbaj folosit pentru stilizarea elementelor HTML. Permite definirea aspectului vizual al paginilor web, inclusiv culorile, fonturile, machetele și animațiile. Regulile și selectoarele CSS sunt folosite pentru a aplica stiluri în mod consecvent în diferite elemente, asigurând o interfață de utilizator coerentă și atractivă din punct de vedere vizual.

JavaScript este un limbaj de programare care adaugă interactivitate și comportament dinamic paginilor web. Permite gestionarea interacțiunilor utilizatorilor, efectuarea validărilor pe partea clientului și manipularea conținutului paginilor web. Presupune implementarea unei logici complexe, a gestionării evenimentelor și a operațiunilor asincrone, îmbunătățind experiența utilizatorului și capacitatea de răspuns a aplicației.

Arhitectura clientului utilizează aceste tehnologii cu scopul de a crea o interfață receptivă și intuitivă pentru a interacționa cu aplicația de testare a algoritmilor pe grafuri. HTML oferă structura, CSS adaugă stiluri vizuale, iar JavaScript aduce interactivitate și funcționalitate paginilor web. Cu WebStorm ca IDE, dezvoltatorii pot dezvolta, testa și depana eficient codul din partea clientului, asigurând o experiență de utilizator fluidă și captivantă.

7.3 Arhitectura serverului

Arhitectura serverului aplicației este concepută pentru a gestiona procesarea și execuția algoritmilor Dijkstra, Bellman Ford și BFS implementați în limbajul de programare C++. Utilizează framework-ul Spring împreună cu limbajul de programare Java. Serverul primește intrări de la client, invocă executabilul C++ corespunzător împreună cu fișierul primit și returnează rezultatele înapoi clientului.

Java servește ca limbaj de programare principal pentru dezvoltarea componentelor serverului aplicației. Oferă o platformă fiabilă pentru gestionarea cererilor web, gestionarea resurselor și rularea executabilelor algoritmilor C++.

Algoritmii Dijkstra, Bellman Ford și BFS sunt implementați în C++ în proiecte diferite pentru a obține o performanță optimă. Codul C++ este compilat în executabile care pot fi executate de server. Aceste executabile sunt responsabile pentru procesarea fișierului .txt încărcat care conține date despre graf și pentru generarea rezultatelor corecte pe baza algoritmului specificat.

Serverul folosește tipul de mapare POST folosind modelul de programare bazat pe adnotări al framework-ului Spring. Acestea sunt responsabile pentru primirea fișierului .txt încărcat de la client și pentru declanșarea execuției executabilului C++ corespunzător. Serverul primește fișierul de la client și îl transmite ca argument executabilului. La finalizare, serverul preia rezultatele obținute și le trimite înapoi

clientului.

Scopul arhitecturii serverului este să execute și să gestioneze algoritmi Dijkstra, Bellman Ford și BFS și să faciliteze integrarea și comunicarea între client și implementările algoritmilor. Serverul asigură securitatea, precum și gestionarea rapidă a potențialelor erori.

Capitolul 8

Detalii de implementare

8.1 Implementarea protocolului de comunicație

Pentru fiecare algoritm și fiecare tip de execuție (secvențial sau paralel) s-a creat o funcție (8.1) în limbajul Java folosind framework-ul Spring, care cuprinde:

- Anotarea cu `@PostMapping`, indicând faptul că gestionează o solicitare POST către punctul final dat ca parametru.
- Verificarea fișierului încărcat (`MultipartFile`), care în cazul în care este gol returnează un mesaj de eroare.
- Generarea unui număr aleator (`random_int`) folosind clasa `Math` într-un interval specificat și construirea unei căi de fișier pentru a salva fișierul încărcat de la pasul anterior. (se asigură că nu există deja fișierul generat în acea cale pentru a evita suprascrierea fișierelor existente; fișierul încărcat este apoi transferat și salvat în locația specificată pe partea de server).
- Construirea unei comenzi pentru a executa executabilul dorit prin adăugarea căii fișierului ca argument, apoi execută comanda folosind metoda `Runtime.getRuntime().exec()`, obținând instanța procesului.
- Crearea unui cititor pentru a reține rezultatul execuției algoritmului din fluxul de intrare al procesului. Citește fiecare linie a ieșirii și o adaugă la o ieșire numită `StringBuilder`.
- Așteptarea finalizării procesului folosind `process.waitFor()`. Dacă procesul se încheie cu un cod de 0 (indicând execuția cu succes), acesta șterge fișierul salvat, convertește ieșirea `StringBuilder` într-un șir de tip `String` și returnează rezultatul algoritmului.
- Captarea excepțiilor sau erorilor din timpul execuției algoritmului, returnând un mesaj de eroare.

La pornirea serverului, această funcție se ocupă de comunicarea între client, server și implementările algoritmilor Dijkstra, Bellman Ford și BFS.


```

21 @PostMapping("/dijkstra-s")
22 public String executeDijkstraSecv(@RequestParam("file") MultipartFile file) {
23     if (file.isEmpty()) {
24         return "Error: Empty file";
25     }
26     try {
27         int min = 1;
28         int max = 5000000;
29         int random_int = (int) Math.floor(Math.random() * (max - min + 1) + min);
30         String filePath = "C:/Users/user/Desktop/ServerLicenta/src/main/java/licenta/serverlicenta/Data"
31             + File.separator + random_int + file.getOriginalFilename();
32         while(new File(filePath).exists())
33         {
34             random_int = (int) Math.floor(Math.random() * (max - min + 1) + min);
35             filePath = "C:/Users/user/Desktop/ServerLicenta/src/main/java/licenta/serverlicenta/Data"
36                 + File.separator + random_int + file.getOriginalFilename();
37         }
38         file.transferTo(new File(filePath));
39         String command = path + "dijkstra_sequential.exe" + " " + filePath;
40         Process process = Runtime.getRuntime().exec(command);
41         BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
42         StringBuilder output = new StringBuilder();
43         String line;
44         while ((line = reader.readLine()) != null) {
45             output.append(line).append("\n");
46         }
47         int exitCode = process.waitFor();
48         if (exitCode == 0) {
49             new File(filePath).delete();
50             return output.toString();
51         } else {...}
52     } catch (IOException | InterruptedException e) {...}

```

Figura 8.1: Exemplu de funcție folosită pentru execuția secvențială a algoritmului Dijkstra

Pe partea de client, se regăsește funcția `executeNextAlgorithm` (8.2), fiind responsabilă cu conectarea clientului la partea de server și transmiterea și primirea datelor pentru algoritmul ales. Această funcție cuprinde:

- Folosirea API-ului de preluare pentru a trimite o solicitare POST către punctul final de pe partea serverului la `http://localhost:8080/` + adresa specificată în funcția din server. Include `formData`, care conține datele necesare pentru execuția algoritmului.
- Procesarea răspunsului când serverul răspunde, folosind `Promise`. Convertește răspunsul în format text folosind `response.text()`, iar rezultatul conține rezultatul execuției algoritmului returnat de server.
- Crearea unui nou element (`executionResult`) pentru a afișa numărul de execuție curent (`executionIndex + 1`) și rezultatul algoritmului. Acesta adaugă elementul într-un `div` care este încărcat cu un `div` existent (`timeContainer`), care reprezintă un container de afișare a rezultatelor.
- Extragerea timpului de execuție din rezultat apelând la `extractExecutionTime`. Rezultatul este un număr care reprezintă timpul de execuție în milisecunde. `ExecutionTime` este adăugat la `totalTime` care va calcula media aritmetică după ce toate

execuțiile funcției sunt terminate. În acest sens, dacă funcția trebuie executată de mai multe ori, programează următoarea execuție a funcției folosind `setTimeout` pe a adăuga o întârziere de timp între execuții. Apelează recursiv `executeNextAlgorithm` cu un `executionIndex` incrementat și transmite `totalTime` și `executionTimes`. Aceasta introduce o întârziere între execuțiile algoritmului definită de `delayBetweenExecutions`.

- Captarea erorilor în timpul execuției algoritmului (prinsă de blocul `.catch()`), înregistrarea erorii și trecerea la executarea următoare a funcției.

Aceste două funcții sunt cele care fac posibilă comunicarea între client și server.

```
function executeNextAlgorithm(executionIndex, totalTime, number = 0, executionTimes = []) {
  fetch({ input: 'http://localhost:8080/dijkstra-s', init: {
    method: 'POST',
    body: formData
  }}).then(response => response.text())
  .then(result => {
    let executionResult = document.createElement('p');
    executionResult.innerText = 'Exec ' + (executionIndex + 1) + ': ' + result;
    timeContainer.appendChild(executionResult);
    const executionTime = extractExecutionTime(result);
    executionTimes.push(executionTime);
    totalTime += executionTime;
    if (executionIndex < numberOfExecutions - 1) {
      setTimeout(handler, () => executeNextAlgorithm(executionIndex + 1, totalTime, executionTimes), delayBetweenExecutions)
    }
    else {
      const meanTime = totalTime / numberOfExecutions;
      const meanTimeFormatted = meanTime.toFixed(fractionDigits: 2);
      let meanResult = document.createElement('p');
      meanResult.innerText = 'Media aritmetica a timpilor executiilor este ' + meanTimeFormatted + 'ms.';
      timeContainer.appendChild(meanResult);
    }
  })
  .catch(error => {
    console.error('Error occurred during algorithm execution:', error);
    if (executionIndex < numberOfExecutions - 1) {
      setTimeout(handler, () => executeNextAlgorithm(executionIndex + 1, totalTime, executionTimes), delayBetweenExecutions)
    }
  })
}
```

Figura 8.2: Exemplu de funcție folosită pentru execuția secvențială a algoritmului Dijkstra

8.2 Implementarea clientului

În partea de client există 2 pagini pentru site-ul web, pagina de început (`index.html`), unde se află titlul proiectului și descrierile despre metodele și algoritmii folosiți în cadrul proiectului, precum și un buton de trecere la pagina principală.

În partea de head a html-ului se află referință către scripturile jquery și main.js, CSS-urile `normalize.css` și `home.css` și "meta charset="UTF-8"" care permite vizualizarea diacriticelor din limba română.

Pentru adăgarea animațiilor, există mai multe div-uri stilizate în CSS. Un exemplu este `div class="wrapper"` care conține mai multe elemente de tip "span" care reproduc fenomenul din viața reală a baloanelor de săpun care se sparg după un anumit timp. De asemenea, sunt folosite `div id="arrowAnim"` și `div id="arrowAnim2"`

pentru a anima săgețile ce indică apăsarea butonului 'Testare', reprezentat ca "button onclick="redirectToAlgorithms()"".

Casetele de text pentru descrieri sunt elemente de tip "div", care au clasa diferită pentru a permite stilizarea diferită în CSS, la fel și pentru liniile negre verticale și orizontale unde sunt folosite elemente de tip "hr" și "div". În script-ul aferent acestei pagini (main.js 8.3) se află funcția pentru redirectionare spre pagina principală.

```
function redirectToAlgorithms() {
    window.location.href = "main.html";
}
```

Figura 8.3: Funcția care rulează când butonul de pe pagina de început este apăsător

Pagina principală (main.html) cuprinde mai multe elemente de tip "div" pentru animații:

- Pentru meniul din care poate fi selectat algoritmul dorit.
- Pentru casetele de text în care există descrierea aferentă algoritmului ales.
- Pentru casetele de text în care apar rezultatele execuției algoritmului dorit în urma apăsării butonului "Secvențial" sau "Paralel".
- Pentru butoanele de încărcare a fișierelor, testarea secvențială, respectiv paralelă a algoritmilor și vizualizarea instrucțiunilor de folosire a aplicației.
- Pentru caseta în care este vizualizat graful și aplicarea algoritmului asupra lui

Când un algoritm este selectat din meniu, vor apărea butoanele de testare, butonul de încărcarea fișierului și caseta de text aferentă. Pentru a face schimbarea de elemente, se va crea un "event listener" (8.4 care va fi declanșat când este selectat un algoritm și vor apărea elementele necesare.

```
for (let i = 0; i < dropdownItems.length; i++) {
    dropdownItems[i].addEventListener( type: "click", listener: function () {
        let selectedItem = this.textContent.trim();
        chosenAlg = selectedItem;
        fileUploadContainer.style.display = "none";
        fileInput.value = '';
        if (selectedItem === "Dijkstra" || selectedItem === "BFS" || selectedItem === "Bellman Ford") {
            fileUploadContainer.style.display = "block";
            testButtonSeq.style.display = "block";
            testButtonPar.style.display = "block";
            arrow1.style.display = "block";
            arrow2.style.display = "block";
            testButtonSeq.style.display = "block";
            testButtonPar.style.display = "block";
            arrowMain.style.display = "none";
            textChooseAlg.style.display = "none";
            btnChoosePerfType.style.display = "block";
            timeBFSDivSeq.style.display = "none";
            timeBFSDivSeq.innerHTML = '';
            timeBFSDivPar.style.display = "none";
            timeBFSDivPar.innerHTML = '';
            timeDijDivSeq.style.display = "none";
            timeDijDivSeq.innerHTML = '';
            timeDijDivPar.style.display = "none";
            timeDijDivPar.innerHTML = '';
            timeBFordDivSeq.style.display = "none";
            timeBFordDivSeq.innerHTML = '';
            timeBFordDivPar.style.display = "none";
            timeBFordDivPar.innerHTML = '';
        }
    });
}
```

Figura 8.4: Funcția care declanșează afișarea elementelor necesare

Pentru butoanele de testare secvențială și paralelă, se va crea tot un "event listener" (8.5) separat pentru fiecare, care va verifica inițial dacă a fost încărcat un fișier (în caz contrar afișează o casetă care informează utilizatorul că trebuie încărcat un fișier), apoi va verifica ce tip de algoritm a fost ales și va apela funcția descrisă la protocolul de comunicare pentru a primi rezultatele corespunzătoare.

```
testButtonSeq.addEventListener( type: "click", listener: function () {
  if (fileInput.files.length === 0) {
    showDialog( message: 'Încărcați un fișier', duration: 3500);
  } else {
    function extractExecutionTime(result) {
      const regex = /(\\d+)\\s*(?:milliseconds|ms)/;
      const match = result.match(regex);
      if (match && match[1]) {
        console.log("Obtained " + parseInt(match[1]));
        return parseInt(match[1]);
      }
      return 0;
    }
    if (chosenAlg === "Dijkstra") {
      if(timeDijDivPar.style.display === 'block' && timeDijDivSeq.style.display === 'block')
      {...}
      timeBFSDivSeq.style.display = 'none';
      timeBFordDivSeq.style.display = 'none';
      timeDijDivSeq.style.display = 'block';
      let file = fileInput.files[0];
      let formData = new FormData();
      formData.append( name: 'file', file);
      let timeContainer = document.getElementById( elementId: "dij-time-seq")
      timeContainer.innerHTML = '';
      let executionDescr = document.createElement( tagName: 'p');
      executionDescr.innerText = 'Se vor realiza 10 executii';
      timeContainer.appendChild(executionDescr);
      let numberOfExecutions = 10; // Number of times to execute the algorithm
      let delayBetweenExecutions = 1000; // Delay in milliseconds
      function executeNextAlgorithm(executionIndex, totalTime, number = 0, executionTimes = []) = []
```

Figura 8.5: Funcția care declanșează butonul de testare secvențială

Caseta pentru vizualizarea grafului folosește librăria Cytoscape.js pentru a crea un element de tip cytoscape conform datelor primite din fișierul încărcat și pentru a aplica algoritmul ales. Această librărie include funcții prestabilite pentru algoritmii Dijkstra, Bellman Ford și BFS, fiind mai ușor de prelucrat, precum și aspecte predefinite care îmbunătățesc modul de afișare a nodurilor și muchiilor (8.6). Când un fișier va fi încărcat cu un graf care este format din mai puțin de 90 de noduri, se va afișa în casetă graful și un mesaj cu nodurile parcurse de algoritm.

Panoul de instrucțiuni cuprinde 2 pagini, prima pagină cu instrucțiuni despre folosirea aplicației și a doua pagină cu instrucțiuni și detalii despre Cytoscape. Pentru a face trecerea de la prima pagină la a doua și invers, s-a folosit o funcție care de fiecare dată când butonul cu 'Pagina 1' sau 'Pagina 2' este apăsat, pagina respectivă butonului va fi afișată în locul celei curente (8.7). Totodată, butonul de închidere este vizibil doar când au fost parcurse ambele pagini cu instrucțiuni.

```

cy.layout({
  name: "breadthfirst",
  roots: '#0',
  avoidOverlap: true,
  nodeDimensionsIncludeLabels: true
}).run();

var bfs = cy.elements().bfs('#0', function () {
}, true);
var i = 0;
var highlightNextEle = function () {
  if (i < bfs.path.length) {
    bfs.path[i].addClass('highlighted');

    i++;
    setTimeout(highlightNextEle, 800);
  }
};
highlightNextEle();

var path = [];

bfs.path.each(function(element){
  if (element.isNode()) {
    path.push(element.id());
  }
});

```

Figura 8.6: Configurația aspectului elementului cytoscape și folosirea funcției bfs în a evidenția fiecare nod parcurs în drumul găsit

```

document.addEventListener( type: 'DOMContentLoaded', listener: function() {
  var instructionsBtn = document.getElementById( elementId: 'instructions-btn');
  var instructionsModal = document.getElementById( elementId: 'instructions-modal');
  var pages = document.getElementsByClassName( className: 'page');
  var currentPageIndex = 0;
  var closeBtn = document.getElementById( elementId: 'close-btn');
  var nextBtn = document.getElementById( elementId: 'next-btn');
  var prevBtn = document.getElementById( elementId: 'prev-btn');

  function showPage(pageIndex) {
    for (var i = 0; i < pages.length; i++) {
      pages[i].style.display = 'none';
    }
    pages[pageIndex].style.display = 'block';
    currentPageIndex = pageIndex;
  }

  function nextPage() {
    if (currentPageIndex < pages.length - 1) {
      showPage( pageIndex: currentPageIndex + 1);
    }
  }

  function prevPage() {...}

  instructionsBtn.addEventListener( type: 'click', listener: function() {
    showPage( pageIndex: 0);
    instructionsModal.style.display = 'block';
  });

  nextBtn.addEventListener( type: 'click', nextPage);
  prevBtn.addEventListener( type: 'click', prevPage);
  closeBtn.addEventListener( type: 'click', listener: function() {
    instructionsModal.style.display = 'none';
  });
});

```

Figura 8.7: Funcția care comută între pagini și iese din panoul de instrucțiuni prin apăsarea butonului "Închide"

8.3 Implementarea serverului

Serverul este implementat în limbajul de programare Java. Proiectul aferent este împărțit în următoarele foldere:

- AlgorithmsExes: conține cele 6 executabile ale algoritmilor implementați (2 pentru fiecare algoritm, secvențial și paralel).
- Controller: conține clasa AlgorithmController care este alcătuită din funcțiile precizate în capitolul protocolului de comunicare.
- Data: stochează temporar fișierele .txt trimise de către client.
- Configuration: conține configurarea CORS (8.8) care este utilizată pentru a controla și gestiona cererile HTTP în aplicația web. Definește un set de reguli care permit serverelor web să specifice ce origini au permisiunea de a accesa resursele sale.
- Rularea serverului se face folosind clasa ServerLicentaApplication (8.9), iar configurarea framework-ului Spring se află în fișierul pom.xml.

```
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class CorsConfiguration implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping(pathPattern: "**")
            .allowedOrigins("http://localhost:63342")
            .allowedMethods("*")
            .allowedHeaders("*");
    }
}
```

Figura 8.8: Configurare CORS

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ServerLicentaApplication {

    public static void main(String[] args) { SpringApplication.run(ServerLicentaApplication.class, args); }
```

Figura 8.9: clasa ServerLicentaApplication

Algoritmii Dijkstra, Bellman Ford și BFS sunt implementați în limbajul de programare C++. Pentru paralelizarea algoritmilor, s-a folosit librăria OpenMP pentru a paraleliza anumite bucăți de cod.

Pașii implementării fiecărui algoritm sunt:

Dijkstra secvențial (8.108.118.12)

- Se definesc 2 structuri Edge și Vertex pentru muchie, respectiv nod.

```
#define INT_MAX 100000
#define TRUE 1
#define FALSE 0

struct Edge {
    int u;
    int v;
};

struct Vertex {
    int title;
    bool visited;
};
```

Figura 8.10: algoritm Dijkstra - Secvențial - partea 1

- Funcția findEdge găsește ponderea unei muchii între două noduri. Iterează prin matricea de muchii și ponderi și returnează ponderea dacă se găsește o muchie corespunzătoare. Dacă nu este găsită nicio muchie, returnează INT_MAX (o valoare definită care ține locul lui infinit).

- Funcția minimum returnează valoarea minimă dintre 2 întregi A și B.

- Funcția minWeight calculează ponderea minimă dintre nodurile nevizitate. Inițializează variabila minimum cu INT_MAX și iterează prin noduri. Dacă se vizitează un nod, acesta este omis. În caz contrar, dacă lungimea (len) nodului este mai mică decât minimul curent, acesta actualizează minimumul.

- Funcția minPath găsește vârful cu ponderea minimă (min) printre nodurile nevizitate. Setează nodul vizitat de index i cu TRUE și returnează indexul acestuia.

- Funcția principală, DijkstraOMP, implementează algoritmul lui Dijkstra. Începe prin inițializarea nodului rădăcină așa cum este vizitat și inițializează matricea de lungime len cu 0 pentru nodul rădăcină și INT_MAX pentru toate celelalte noduri, apoi iterează prin noduri, setând lungimea fiecărui nod cu ponderea muchiei dintre rădăcină și acel nod. De asemenea, marchează nodul rădăcină ca fiind vizitat.

```
int minimum(int A, int B) {
    if (A > B) {
        return B;
    } else {
        return A;
    }
}

int minWeight(int* len, Vertex* vertices, int V) {
    int minimum = INT_MAX;
    for (int i = 0; i < V; i++) {
        if (vertices[i].visited == TRUE) {
            continue;
        } else if (vertices[i].visited == FALSE && len[i] < minimum) {
            minimum = len[i];
        }
    }
    return minimum;
}

int minPath(Vertex* vertices, int* len, int V) {
    int i;
    int min = minWeight(len, vertices, V);

    for (i = 0; i < V; i++) {
        if (vertices[i].visited == FALSE && len[vertices[i].title] == min) {
            vertices[i].visited = TRUE;
            return i;
        }
    }
}
```

Figura 8.11: algoritm Dijkstra - Secvențial - partea 2

```

void DijkstraOMP(Vertex* vertices, Edge* edges, int* weights, Vertex* root, int V, int E) {
    double start, end;
    root->visited = TRUE;
    int len[V];
    len[(int)root->title] = 0;

    int i, j;

    for (i = 0; i < V; i++) {
        if (vertices[i].title != root->title) {
            len[(int)vertices[i].title] = findEdge(*root, vertices[i], edges, weights, E);
        } else {
            vertices[i].visited = TRUE;
        }
    }

    for (j = 0; j < V; j++) {
        Vertex u;
        int h = minPath(vertices, len, V);
        u = vertices[h];

        for (i = 0; i < V; i++) {
            if (vertices[i].visited == FALSE) {
                int c = findEdge(u, vertices[i], edges, weights, E);
                len[vertices[i].title] = minimum(len[vertices[i].title], len[u.title] + c);
            }
        }
    }
}

```

Figura 8.12: algoritm Dijkstra - Secvențial - partea 3

Dijkstra paralel (8.13)

Singura diferență se regăsește în funcția principală, DijkstraOMP, unde este adăugat `#pragma omp parallel for schedule(runtime) private(i)` în execuția ultimului for, paralelizându-l.

```

void DijkstraOMP(Vertex* vertices, Edge* edges, int* weights, Vertex* root, int V, int E) {
    double start, end;
    root->visited = TRUE;
    int len[V];
    len[(int)root->title] = 0;

    int i, j;

    for (i = 0; i < V; i++) {
        if (vertices[i].title != root->title) {
            len[(int)vertices[i].title] = findEdge(*root, vertices[i], edges, weights, E);
        } else {
            vertices[i].visited = TRUE;
        }
    }

    for (j = 0; j < V; j++) {
        Vertex u;
        int h = minPath(vertices, len, V);
        u = vertices[h];

#pragma omp parallel for schedule(runtime) private(i)
        for (i = 0; i < V; i++) {
            if (vertices[i].visited == FALSE) {
                int c = findEdge(u, vertices[i], edges, weights, E);
                len[vertices[i].title] = minimum(len[vertices[i].title], len[u.title] + c);
            }
        }
    }
}

```

Figura 8.13: algoritm Dijkstra - Paralel

Bellman Ford secvențial (8.148.158.16)

- Namespace-ul `utils` cuprinde funcții utile pentru această implementare, precum afișarea unui mesaj pentru erori, convertirea unei coordonate cu 2 dimensiuni în una singură, citire fișier și tipărire rezultat.
- Funcția principală, `bellman_ford`, Începe prin a inițializa matricea distanțelor

cu infinit pentru toate nodurile, cu excepția nodului sursă, care are setată valoarea 0. Funcția efectuează apoi $n - 1$ iterații, unde n este numărul de noduri din graf. În fiecare iterație, verifică fiecare muchie din graf și le relaxează dacă se găsește o cale mai scurtă. Dacă o muchie de la nodul u la nodul v are o pondere mai mică decât infinit, se verifică dacă muchia ($\text{dist}[u] + \text{pondere}$) duce la un drum mai scurt către v . Dacă este îndeplinită condiția, se actualizează $\text{dist}[v]$ cu noua distanță mai scurtă. Dacă nu apar alte modificări, indică faptul că cele mai scurte drumuri au fost găsite și funcția revine.

```
#define INF 1000000

/**...*/
namespace utils {
    int N; //number of vertices
    int *mat; // the adjacency matrix

    void abort_with_error_message(string msg) {
        std::cerr << msg << endl;
        abort();
    }

    //translate 2-dimension coordinate to 1-dimension
    int convert_dimension_2D_1D(int x, int y, int n) {
        return x * n + y;
    }

    int read_file(string filename) {
        std::ifstream inputf(filename, std::ifstream::in);
        if (!inputf.good()) {
            abort_with_error_message("ERROR OCCURRED WHILE READING INPUT FILE");
        }
        inputf >> N;
        //input matrix should be smaller than 20MB * 20MB
        assert(N < (1024 * 1024 * 20));
        mat = (int *) malloc(N * N * sizeof(int));
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++) {
                inputf >> mat[convert_dimension_2D_1D(i, j, N)];
            }
    }
}
```

Figura 8.14: algoritm Bellman Ford - Secvențial - partea 1

```
void bellman_ford(int n, int *mat, int *dist, bool *has_negative_cycle) {
    *has_negative_cycle = false;
    for (int i = 0; i < n; i++) {
        dist[i] = INF;
    }
    dist[0] = 0;
    bool has_change;

    for (int i = 0; i < n - 1; i++) { // n - 1 iteration
        has_change = false;
        for (int u = 0; u < n; u++) {
            for (int v = 0; v < n; v++) {
                int weight = mat[utils::convert_dimension_2D_1D(u, v, n)];
                if (weight < INF) { //test if u-v has an edge
                    if (dist[u] + weight < dist[v]) {
                        has_change = true;
                        dist[v] = dist[u] + weight;
                    }
                }
            }
        }
        if (!has_change) {
            return;
        }
    }

    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            int weight = mat[utils::convert_dimension_2D_1D(u, v, n)];
            if (weight < INF) {
```

Figura 8.15: algoritm Bellman Ford - Secvențial - partea 2

```

has_change = false;
for (int u = 0; u < n; u++) {
    for (int v = 0; v < n; v++) {
        int weight = mat[utils::convert_dimension_2D_1D(u, v, n)];
        if (weight < INF) { //test if u--v has an edge
            if (dist[u] + weight < dist[v]) {
                has_change = true;
                dist[v] = dist[u] + weight;
            }
        }
    }
}

if (!has_change) {
    return;
}

for (int u = 0; u < n; u++) {
    for (int v = 0; v < n; v++) {
        int weight = mat[utils::convert_dimension_2D_1D(u, v, n)];
        if (weight < INF) {
            if (dist[u] + weight < dist[v]) { // if we can relax one more
                *has_negative_cycle = true;
                return;
            }
        }
    }
}
}

```

Figura 8.16: algoritm Bellman Ford - Secvențial - partea 3

Bellman Ford paralel (8.178.188.19)

Față de implementarea secvențială, implementarea paralelă a algoritmului Bellman Ford conține și următoarele directive OpenMP:

- `omp_set_num_threads(p)`: include setarea numărului de fire.
- `#pragma omp parallel for`: crearea de bucle paralele.
- `#pragma omp parallel`: matricea dist este paralelizată pentru a distribui munca de setare a distanțelor inițiale la infinit.
- `#pragma omp barrier`: sincronizează toate thread-urile la sfârșitul fiecărei iterații a buclei exterioare. Acest lucru asigură că toate thread-urile și-au finalizat sarcinile respective înainte de a trece la următoarea iterație.

```

void bellman_ford(int p, int n, int *mat, int *dist, bool *has_negative_cycle) {
    int local_start[p], local_end[p];
    *has_negative_cycle = false;

    //step 1: set openmp thread number
    omp_set_num_threads(p);

    //step 2: find local task range
    int ave = n / p;
    #pragma omp parallel for
    for (int i = 0; i < p; i++) {
        local_start[i] = ave * i;
        local_end[i] = ave * (i + 1);
        if (i == p - 1) {
            local_end[i] = n;
        }
    }

    //step 3: bellman-ford algorithm
    //initialize distances
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        dist[i] = INF;
    }
    //root vertex always has distance 0
    dist[0] = 0;

    int iter_num = 0;
    bool has_change;
}

```

Figura 8.17: algoritm Bellman Ford - Paralel - partea 1

```

#pragma omp parallel for
for (int i = 0; i < n; i++) {
    dist[i] = INF;
}
//root vertex always has distance 0
dist[0] = 0;

int iter_num = 0;
bool has_change;
bool local_has_change[p];
#pragma omp parallel
{
    int my_rank = omp_get_thread_num();
    //bellman-ford algorithm
    for (int iter = 0; iter < n - 1; iter++) {
        local_has_change[my_rank] = false;
        for (int u = 0; u < n; u++) {
            for (int v = local_start[my_rank]; v < local_end[my_rank]; v++) {
                int weight = mat[utils::convert_dimension_2D_1D(u, v, n)];
                if (weight < INF) {
                    int new_dis = dist[u] + weight;
                    if (new_dis < dist[v]) {
                        local_has_change[my_rank] = true;
                        dist[v] = new_dis;
                    }
                }
            }
        }
    }
}
#pragma omp barrier
#pragma omp single

```

Figura 8.18: algoritm Bellman Ford - Paralel - partea 2

```

#pragma omp barrier
#pragma omp single
{
    iter_num++;
    has_change = false;
    for (int rank = 0; rank < p; rank++) {
        has_change |= local_has_change[rank];
    }

    if (!has_change) {
        break;
    }
}

if (iter_num == n - 1) {
    has_change = false;
    for (int u = 0; u < n; u++) {
#pragma omp parallel for reduction(|:has_change)
        for (int v = 0; v < n; v++) {
            int weight = mat[u * n + v];
            if (weight < INF) {
                if (dist[u] + weight < dist[v]) {
                    has_change = true;
                }
            }
        }
    }
    *has_negative_cycle = has_change;
}

```

Figura 8.19: algoritm Bellman Ford - Paralel - partea 3

BFS secvențial (8.208.21)

- Se definesc graph, un vector 2D care reprezintă lista de adiacență a grafului, qq o coadă de așteptare folosită pentru a stoca nodurile care urmează să fie procesate în timpul traversării BFS, marked o matrice folosită pentru a urmări starea fiecărui nod din grafic (0 = nemarcat, 1 = marcat, dar neprocesat, 2 = marcat și procesat).

- Funcția principală, s.bfs, ia un startNode ca parametru și începe prin a-l împinge în coada qq. Partea principală a funcției este o buclă while care continuă până când coada q devine goală. În interiorul buclei, se efectuează preluarea nodului din capul cozii folosind qq.front(), apoi îl elimină din coadă folosind qq.pop(). Se marchează nodul preluat ca procesat setând marked[nod] la 2. Se iterează prin nodurile adiacente ale nodului curent folosind o buclă for, iar dacă un nod adiacent este nemarcat

(`marked[graf[nod]][i] == 0`), acesta este împins în coada `qq` și marcat ca vizitat (`marked[graf[nod]][i] = 1`). Acest proces continuă până când toate nodurile accesibile de la `startNode` au fost vizitate și procesate. Algoritmul BFS explorează graful vizitând nodurile nivel cu nivel, asigurându-se că toate nodurile de la nivelul curent sunt vizitate înainte de a trece la nivelul următor.

```
vector<vector<int>>>graph;
queue <int> qq;
int *marked = NULL;
```

Figura 8.20: algoritm BFS - Secvențial - partea 1

```
void s_bfs (int startNode) {
    qq.push (startNode);

    while (! qq.empty()) {
        int node = qq.front();
        qq.pop();

        marked[node] = 2;

        for (unsigned int i = 0; i < graph[node].size(); ++i) {
            if ( marked[graf[node]][i] == 0 ) {
                qq.push (graph[node][i]);
                marked[graf[node][i]] = 1;
            }
        }
    }
}
```

Figura 8.21: algoritm BFS - Secvențial - partea 2

BFS paralel (8.228.23)

Față de implementarea secvențială, implementarea paralelă a algoritmului BFS conține și următoarele directive OpenMP:

- `omp_set_num_threads(threads_num)`: setează numărul de thread-uri.
- `omp_init_lock(&lck)`: inițializează o blocare care este utilizată pentru a sincroniza accesul la structurile de date partajate.

În interiorul buclei principale, o directivă unică `omp` este utilizată pentru a se asigura că un singur fir realizează anumite operații. Mai exact, nodul frontal este preluat din coadă (`qq.front()`), marcat ca procesat (`marked[node] = 2`) și apoi eliminat din coadă (`qq.pop()`). Aceste operațiuni sunt executate de un singur thread pentru a evita conflictele.

- `#pragma omp barrier`: sincronizează toate thread-urile înainte de a trece la următoarea parte a codului. Acest lucru asigură că toate thread-urile de execuție au terminat de procesat nodul curent înainte de a trece mai departe.

- `#pragma omp parallel for`: paralelizează bucla de iterare a nodurilor adiacente. Acest lucru permite mai multor thread-uri să execute iterațiile buclei în paralel.
- `omp_destroy_lock(&lck)`: distruge blocarea făcută la începutul codului (deblochează `omp_init_lock(&lck)`).

```
void p_bfs()
{
    // Setup number of threads
    omp_set_num_threads(threads_num);

    // Setup omp lock
    omp_lock_t lck;
    omp_init_lock(&lck);

    root = 0;
    qq.push(root);
    while (!qq.empty())
    {
#pragma omp parallel
    {
#pragma omp single
        {
            node = qq.front();
            qq.pop();

            marked[node] = 2;
        }
    }

#pragma omp barrier
    }
}
```

Figura 8.22: algoritm BFS - Paralel - partea 1

```
#pragma omp parallel for shared(lck) schedule(dynamic)
for (i = 0; i < graph[node].size(); ++i)
{
    omp_set_lock(&lck);
    if (marked[graph[node][i]] == 0)
    {
        qq.push(graph[node][i]);
        marked[graph[node][i]] = 1;
    }
    omp_unset_lock(&lck);
}

// Cleanup
omp_destroy_lock(&lck);
free(marked);
}
```

Figura 8.23: algoritm BFS - Paralel - partea 2

8.4 Interfață de utilizare

Interfața de utilizare cuprinde următoarele acțiuni:

- Vizionarea paginii de început unde se află descrierile aferente proiectului.



- Apăsarea butonului 'Testare' și redirectionarea spre pagina principală.



- Vizualizarea instrucțiunilor aplicației și a librăriei Cytoscape.js.

Instrucțiuni Algoritmi

1. Alege tipul de algoritm dorit selectând-ul din meniul 'Algoritmi'.
2. Încarcă un fișier de tip .txt apăsând butonul 'Choose File'.
3. Apasă butoanele 'Secvențial' și 'Paralel' pentru a porni rularea.

! Așteaptă până la terminarea tuturor celor 10 execuții până a apăsa din nou pe același buton.

Tipuri de conținut pentru fișierul .txt acceptate:

Dijkstra > pe prima linie numărul de noduri și de muchii, iar pe următoarele linii nodurile fiecărei muchii și ponderea muchiei. Pe ultima linie va fi nodul sursă.

BFS > pe prima linie numărul de noduri, iar pe următoarele linii nodurile muchiilor.

Bellman Ford > pe prima linie numărul de noduri, iar pe următoarele linii matricea costurilor grafului ponderat și orientat. Pe ultima linie va fi nodul sursă.

Pagina 2

Detalii Cytoscape

Cytoscape este o librărie JavaScript utilizată pentru construirea și vizualizarea grafurilor interactiv. Ea oferă un set amplu de funcționalități pentru crearea și personalizarea grafurilor, precum și pentru aplicarea de algoritmi de analiză și manipulare a grafurilor.

Grafurile care pot fi vizualizate trebuie să aibă maxim 90 de noduri, altfel nu se va afișa nimic în chenarul din dreapta.

Fiecare nod parcurs de algoritm va fi colorat cu roșu.

Detalii algoritmi:

Dijkstra > se va găsi cel mai scurt drum de la nodul sursă (0) spre un nod aleatoriu.

BFS > se va aplica căutarea începând cu nodul sursă (0).

Bellman Ford > se va găsi cel mai scurt drum de la nodul sursă (0) spre un nod aleatoriu.

Pagina 1 Închide

- Vizualizarea unui graf neorientat în caseta din dreapta (graful trebuie să aibă maxim 90 de noduri) când se folosește algoritmul BFS.

Algoritmi
Încarcă fișier
Choose File 16nodes.txt
Secvențial << Paralel <<
Alege un tip de testare
Instrucțiuni
Înapoi

Breadth-First-Search

BFS este un algoritm de căutare sau traversare a unui graf care vizitează nodurile în funcție de nivelurile lor de adâncime în raport cu un nod de start. BFS explorează întâi toți vecinii nodului de start, apoi vecinii vecinilor și așa mai departe, astfel încât să exploreze mai întâi nodurile de la același nivel înainte de a trece la nivelul următor.

Execuție algoritm

În partea dreaptă vor fi proiectate cele două tipuri de execuție folosite: secvențial și paralel.

Fiecare tip de execuție va fi executat de 10 ori (execuțiile sunt independente) pentru a măsura timpul procesării algoritmului.

Se va putea observa că pentru BFS, fiind un algoritm care nu are o complexitate mărită, este nevoie de grafiuri de dimensiuni foarte mari pentru ca execuția paralelă să fie mai eficientă decât cea secvențială.

Path: 0 > 1 > 2 > 3 > 4 > 5 > 7 > 8 > 12 > 6 > 11 > 10 > 9 > 13 > 15 > 14

Activate Windows
Go to Settings to activate Windows.

- Testarea butoanelor 'Secvențial' și 'Paralel' pe un fișier .txt încărcat folosind butonul 'Choose File'.

Algoritmi

Încarcă fișier

Choose File

100nodes.txt

Secvențial <<<

Paralel <<<

Alege un tip de testare

Instrucțiuni

Înapoi

Dijkstra

Dijkstra este un algoritm clasic utilizat pentru a găsi cel mai scurt drum între două noduri într-un graf ponderat. Acesta funcționează eficient pe grafuri fără muchii cu ponderi negative.

Graf ponderat

Un graf ponderat constă într-o colecție de noduri interconectate prin muchii, fiecare dintre ele având o valoare numerică asociată numită "ponderare". Aceste ponderi reprezintă costul sau distanța relației dintre nodurile conectate.

Execuție algoritm

În partea dreaptă vor fi proiectate cele două tipuri de execuție folosite: secvențial și paralel.

Fiecare tip de execuție va fi executat de 10 ori (execuțiile sunt independente) pentru a măsura timpul procesării algoritmului.

Se va putea observa că pentru grafurile de dimensiune mare (peste 1000 de noduri) execuția paralelă este mai rapidă.

Se vor realiza 10 execuții

Observație : Modul de lucru paralel este mai eficient cu cat dimensiunea grafului crește.

Exec 1: Serial Dijkstra: 6 milliseconds

Exec 2: Serial Dijkstra: 7 milliseconds

Exec 3: Serial Dijkstra: 7 milliseconds

Exec 4: Serial Dijkstra: 6 milliseconds

Exec 5: Serial Dijkstra: 6 milliseconds

Exec 6: Serial Dijkstra: 7 milliseconds

Exec 7: Serial Dijkstra: 7 milliseconds

Exec 8: Serial Dijkstra: 6 milliseconds

Exec 9: Serial Dijkstra: 5 milliseconds

Exec 10: Serial Dijkstra: 7 milliseconds

Media aritmetica a timpilor executiilor este 6.40ms.

Se vor realiza 10 execuții

Observație : Modul de lucru paralel este mai eficient cu cat dimensiunea grafului crește.

Exec 1: Dijkstra paralel: 16 milliseconds

Exec 2: Dijkstra paralel: 13 milliseconds

Exec 3: Dijkstra paralel: 14 milliseconds

Exec 4: Dijkstra paralel: 13 milliseconds

Exec 5: Dijkstra paralel: 13 milliseconds

Exec 6: Dijkstra paralel: 15 milliseconds

Exec 7: Dijkstra paralel: 13 milliseconds

Exec 8: Dijkstra paralel: 11 milliseconds

Exec 9: Dijkstra paralel: 14 milliseconds

Exec 10: Dijkstra paralel: 14 milliseconds

Media aritmetica a timpilor executiilor este 13.60ms.

Activate Windows
Go to Settings to activate Windows.

- Vizualizarea unui graf orientat ponderat când se folosește algoritmul Bellman Ford.

Algoritmi

Încarcă fișier

Choose File

10nodes.txt

Secvențial <<<

Paralel <<<

Alege un tip de testare

Instrucțiuni

Înapoi

Bellman Ford

Bellman Ford este un algoritm clasic utilizat pentru a găsi cel mai scurt drum între un nod de început și toate celelalte noduri într-un graf ponderat (definiția se află în cadrul descrierii algoritmului Dijkstra) și/sau orientat, inclusiv grafurile care conțin muchii cu ponderi negative.

Graf orientat

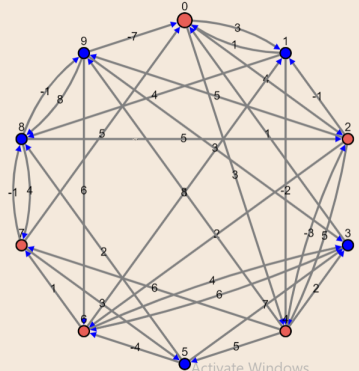
Un graf orientat este o structură de date care constă dintr-o colecție de noduri și muchii, în care fiecare muchie are o direcție asociată. Această direcție indică sensul în care se poate face traversarea de la un nod la altul.

Execuție algoritm

În partea dreaptă vor fi proiectate cele două tipuri de execuție folosite: secvențial și paralel.

Fiecare tip de execuție va fi executat de 10 ori (execuțiile sunt independente) pentru a măsura timpul procesării algoritmului.

Shortest path from Node 0 to Node 7 is : 0 > 4 > 2 > 6 > 7



Activate Windows
Go to Settings to activate Windows.

Bibliografie

Bibliografie

- [Ale22] Leandro Alegsa. Problema podurilor din Königsberg. <https://ro.alegsaonline.com/art/89192>, 2022.
- [Boa21] OpenMP Architecture Review Board. *OpenMP 5.2 Reference Guide*. OpenMP Architecture Review Board, 2021.
- [DRA16] PPAP DRAFT. *Algorithm Design: Parallel and Sequential*. 2016.
- [GGKK03] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- [Ha12] Viet Hai Ha. Optimization of memory management on distributed machine. https://www.researchgate.net/publication/281015120_Optimization_of_memory_management_on_distributed_machine, 2012.
- [SDD⁺22] A.R. Sanaullah, Anupam Das, Anik Das, Muhammad Ashad Kabir, and Kai Shu. Applications of machine learning for covid-19 misinformation: a systematic review. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9336132/>, 2022.
- [Sto] Cristiana Stoica. Teorema celor 4 culori - de la istorie la matematică -. https://www.academia.edu/13920260/TEOREMA_CELOR_4_CULORI_TEOREMA_CELOR_4_CULORI_DE_LA_ISTORIE_LA_DE_LA_ISTORIE_LA_MATEMATIC_MATEMATIC_%C4%82_%C4%82.
- [Tim23] Todd Timmons. The birth of graph theory: Leonhard Euler and the Königsberg bridge problem. <https://www.encyclopedia.com/science/encyclopedias-almanacs-transcripts-and-maps/birth-graph-theory-leonhard-euler-and-koenigsberg-bridge-problem>, 2023.