UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

IMPLEMENTAREA ȘI COMPARAREA ALGORITMILOR DISTRIBUIȚI ÎNTR-UN MEDIU SIMULAT

Absolvent
Bianca-Mihaela Stan

Coordonator științific
Conf. Dr. Ing. Paul Irofti

București, iunie 2023

Rezumat

Algoritmii distribuiți joacă un rol central în domeniul sistemelor distribuite. Asigurarea corectitudinii algoritmilor este esențială în studiul acestora. În general, descrierea algoritmilor distribuiți se concentrează pe prezentarea în pseudocod a unor notiuni teoretice. Acestă metodă este utilă în înțelegerea conceptelor introduse de algoritm, însă nu asigură corectitudinea în implementare.

Această lucrare explorează rolul utilizării unei simulări a algoritmilor în DistAlgo prin studiul a două probleme fundamentale: mulțimea independentă maximală (MIS) și arborele de acoperire minim (MST). Acest tip de simulare ajută la garantarea corectitudinii algoritmilor, poate ajuta la identificarea unei implementări ineficiente și este suficient de clară pentru a putea fi folosită în prezentarea inițială a unui algoritm.

Abstract

Distributed algorithms play a central role in distributed systems. Making sure algorithms are correct is essential in their study. Generally, the description of a distributed algorithm focuses on presenting a pseudocode of some theoretical ideas. This is useful in understanding the concepts, but doesn't ensure the correctness of the implementation.

This paper explores the role of using the simulation of an algorithm in DistAlgo for the study of two fundamental problems: maximal independent set (MIS) and minimum spanning tree (MST). This type of simulation helps ensure the correctness of the algorithms, can help identify inefficient implementations and is sufficiently clear to be used in the initial presentation of an algorithm.

Cuprins

Capitolul 1

Introducere

1.1 Algoritmi distribuiți

Sistemele distribuite sunt sisteme de mașini ale unei rețele care își coordonează acțiunile prin transmiterea de mesaje de-a lungul muchiilor rețelei. În astfel de sisteme, rezolvarea unei probleme, cum ar fi alegerea unui lider, impun o cu totul altă paradigmă de rezolvare datorită faptului că niciuna dintre mașini nu are toată informația despre rețea. Domeniul algoritmilor distribuiți studiază construirea algoritmilor care să rezolve probleme în context distribuit.

1.2 Scopul lucrării

Studiul algoritmilor distribuiți într-o rețea reală este în general dificil. Chiar și atunci când avem acces la o rețea suficient de mare pentru rularea unui algoritm distribuit, studiul poate beneficia de rularea pe rețele diferite sau de configurarea diferită a mașinilor.

În general studiul algoritmilor distribuiți se concentrează pe algoritmi teoretici exprimați prin pseudocod. Acesta este util în înțelegerea conceptelor introduse de un algoritm, dar nu este foarte precis prin faptul că nu asigură corectitudinea într-un context real. [17]

O alternativă o reprezintă simularea unui context distribuit pentru studiul algoritmilor. Această este o metodă accesibilă și flexibilă care facilitează studiul. Putem folosi paralelizarea disponibilă pe o mașină pentru a simula un algoritm distribuit prin transmiterea de mesaje între procesele individuale.

Simulările de acest tip au însă limitări. Această lucrare își propune să răspundă la întrebările:

Cât de util este studiul unui algoritm într-un mediu distribuit simulat?

Un alt considerent important este îmbunătățirea algoritmilor existenți. Pentru a utiliza o simulare a algoritmului pentru acest scop avem nevoie să știm dacă diferența de performanța dintre doi algoritmi ne poate spune ceva despre diferența de complexitate

acestora.

Putem valida o îmbunătățire de complexitate prin rularea într-un mediu simulat?

1.3 Structura lucrării

Lucrarea se împarte în următoarele capitole:

1. **Introducere**: descrie scopul și domeniul lucrării.
2. **Preliminarii**: descrie conceptele teoretice utilizate în lucrare, precum și alegerea mediului de simulare.
3. **Mulțimea independentă maximală**: prezintă implementarea a doi algoritmi distribuiți pentru rezolvarea problemei mulțimii independente maximale.
4. **Arborele de acoperire minim**: prezintă implementarea a doi algoritmi distribuiți pentru rezolvarea problemei arborelui de acoperire minim.
5. **Concluzii**: rezumă rezultatele obținute în urma studiului și direcțiile viitoare.

1.4 Motivația personală

Am ales această tema din nevoia de a înțelege în profunzime algoritmii distribuiți. Am ales să mă concentrez pe probleme simple și să explorez impedimentele ce apar în implementarea unor algoritmi deja cunoscuți. Cercetarea pe acest subiect m-a ajutat să înțeleg atât dificultatea cât și importanța abordării practice asupra algoritmilor distribuiți.

Capitolul 2

Preliminarii

2.1 Algoritmi distribuiți

Un algoritm distribuit este un algoritm care rulează pe procesele individuale ale unei rețele de procese interconectate. Rețelele folosesc algoritmi distribuiți pentru a asigura o cooperare între procese, cu scopul rezolvării unei probleme comune. [31]

De-a lungul lucrării voi considera că în sisteme nu intervin eșecuri, astfel încât mesajele sunt transmise corect, fără a fi corupte sau pierdute.

2.1.1 Algoritmi sincroni și asincroni

Algoritmii distribuiți sincroni rulează pe sisteme distribuite în care există o limită superioară cunoscută pentru fiecare pas de procesare, o limită cunoscuta pentru transmiterea mesajelor și în care procesele sunt caracterizate de un set de ceasuri fizice perfect sincronizate. Pentru astfel de sisteme, rularea algoritmilor poate fi organizată în runde sincrone. [19]

Algoritmii distribuiți asincroni permit fiecărui procesor să calculeze și să comunice în ritmul său. Acest tip de algoritm este util atunci când ne dorim ca procesele să nu aștepte răspunsul la anumite mesaje, să permitem o performanță crescută pentru unele procese și totodată o posibilă inactivitate temporară. [2]

2.1.2 Algoritmi randomizați

Un algoritm distribuit randomizat este un algoritm care conține o asignare a unei variabile aleatorii. Prin faptul că algoritmul nu are un comportament predefinit, bazându-se pe o variabilă aleatorie, acesta devine un algoritm randomizat. O consecință a folosirii unui algoritm randomizat este necesitatea folosirii unei analize probabilistice pentru a demonstra corectitudinea algoritmului. [22]

2.1.3 Calcularea complexității

Pentru calculul complexității de timp, avem două cazuri: sistemele sincrone și sistemele asincrone. Pentru sistemele sincrone, calculul se face în funcție de numărul de bătăi ale ceasului de sincronizare, numite "runde". În context asincron însă, complexitatea este reprezentată de numărul de unități de timp dintre început la final, cu asumpția că fiecare mesaj trimis durează o unitate de timp. [24]

Complexitatea de comunicare este numărul total de mesaje care sunt trimise între perechile de noduri în timpul rulării. În modelul CONGEST fiecare mesaj are dimensiune $O(logn)$. În cadrul modelului LOCAL nu există o limită pentru dimensiunea mesajelor trimise, astfel încât o măsurătoare mai bună poate fi complexitatea de biți. [24]

2.2 Alegerea mediului de implementare

De-a lungul timpului au apărut multe platforme/programe care să faciliteze simularea unui context distribuit. De-a lungul cercetării mele, am identificat următoarele medii de implementare:

- **Neko** [28]: un framework de Java care permite implementarea rapidă a algoritmilor distribuiți.
- **Pymote** [3]: o librărie de Python pentru simularea și evaluarea algoritmilor distribuiți bazați pe evenimente.
- **ViSiDia** [1]: un framework de Java pentru design-ul, simularea și vizualizarea algoritmilor distribuiți.
- **DAP** [5]: un toolkit de Java pentru simularea algoritmilor distribuiți.
- **DisASTer** [23]: librărie de Java, platformă pentru implementarea algoritmilor distribuiți.
- **DistAlgo** [17]: un limbaj de programare peste Python pentru implementarea algoritmilor distribuiți.
- **DAJ** [25]: un toolkit pentru implementarea, testarea, simularea și vizualizarea algoritmilor distribuiți.

În alegerea mediului de implementare am ales să folosesc limbajul Python. Cele două resurse găsite care folosesc limbajul Python sunt Pymote și DistAlgo, însă Pymote este depreciat. Astfel, voi alege să folosesc DistAlgo.

2.2.1 DistAlgo

Câteva avantaje în folosirea limbajului DistAlgo sunt:

1. Curba de învățare redusă. Limbajul DistAlgo are la bază limbajul Python, astfel că este foarte ușor de preluat și aplicat.

2. Claritatea limbajului Python. Simplitatea limbajului ajută la clarificarea algoritmilor distribuiți, care pot deveni destul de complecși.

3. Dimensiunea implementării. Algoritmii distribuiți își găsesc implementări succinte în limbajul DistAlgo, comparativ cu alte limbaje de programare. Un algoritm distribuit care ar fi fost scris în 3000 de linii de cod în C++ poate fi scris în 300 de linii de cod în DistAlgo. [17]

4. Eficiența DistAlgo. Prin optimizările compilatorului DistAlgo, dar și prin oportunitățile de optimizare mai evidente în cazul unei implementări clare, un algoritm implementat în DistAlgo poate reuși să fie mai eficient decât unul implementat în C++. [17]

5. Caracterul opensource. Întregul cod pentru limbajul DistAlgo este disponibil la adresa: https://github.com/DistAlgo/distalgo. Accesul la codul din spatele limbajului facilitează procesul de învățare.

Câteva contraargumente pentru folosirea DistAlgo sunt:

- Pentru folosirea DistAlgo este necesară una dintre versiunile de Python 3.5, 3.6 sau 3.7. Laptopurile cu procesor Apple M1 nu suportă versiuni de Python mai vechi de 3.8. Astfel, găsirea unei versiuni care să poată fi suportată de mașină a fost dificilă. În final am folosit versiunea 3.7.10.

- Codul sursă pentru limbajul DistAlgo include și un director de ”benchmarks” (evaluare a performanței algoritmilor) care conține anumiți *controllers* pentru a măsura performanța algoritmilor. Acest cod a fost scris în 2016, atunci când versiunea de Python utilizată era 3.6. Laptopurile cu procesor Apple M1 nu pot rula acel cod.

- Mesajele de eroare sunt vagi. Mesajul de eroare nu indică linia la care a apărut eroarea, ci indică ultima funcție *receive*, nu neapărat funcția care a generat eroarea.

  Spre exemplu, după primirea unui mesaj *Accept* se apelează procedura *report*. Pentru o eroare în procedura *report* vom primi doar o informație vagă despre mesajul *Accept*. Mai exact, eroarea nu afișează și numele mesajului *Accept*, ci doar parametrii săi. Pentru o eroare generată dintr-un mesaj *Accept* aceasta va fi marcată cu un anumit *handler 'P_handler_1569'*, iar mesajul de eroare va fi cel din Figura 2.1.

```
[2375] ghs.P<P:86c61>:ERROR: TypeError("'<' not supported between instances of 'str' and 'floa
t'") when calling handler '_P_handler_1569' with '{'w': 'fine', 'source_index': 18}': '<' not
supported between instances of 'str' and 'float'
```

Figura 2.1: Mesajul de eroare pentru o eroare generată din prelucrarea mesajului *Accept*

Capitolul 3

Mulțimea independentă maximală

În teoria grafurilor, o **mulțime independentă** de noduri este o mulțime de noduri dintr-un graf astfel încât oricare două noduri din mulțime nu sunt adiacente în graf. [32] O **mulțime independentă** de muchii este o mulțime de muchii dintr-un graf astfel încât oricare două muchii din mulțime nu sunt adiacente aceluiași nod din graf.

O **mulțime independentă maximală** (MIS) a unui graf este o mulțime independentă care nu este submulțime a niciunei alte mulțimi independente pentru acel graf. [33]

În context distribuit, problema mulțimii independente maximale are aplicații în mai multe contexte. Câteva exemple sunt task scheduling [10], alocarea resurselor [35] și comunicarea rețelelor de senzori [21].

3.1 Problema localității

În context distribuit, fiecare nod are informații parțiale, locale, despre vecinii săi direcți. Problema localității încearcă să răspundă la întrebarea:

În ce măsură putem găsi o soluție globală lucrând cu date locale?[14]

Modelarea matematică - Modelul LOCAL Pentru a modela problema localității este folosit modelul LOCAL reprezentat printr-un graf neorientat, neponderat cu n noduri $G = (V, E)$ cu $V = \{1, 2, ..., n\}$. Există un proces pentru fiecare nod $v \in V$. Fiecare nod cunoaște valoarea lui n, propriul id, propriul index în mulțimea V și id-urile vecinilor. Algoritmii funcționează în runde sincrone. În fiecare rundă, fiecare nod calculează anumite valori bazate pe propriile informații locale și trimite un mesaj de dimensiune B biți către fiecare vecin al său. De obicei $B = O(logn)$. La finalul rundei fiecare nod primește și procesează mesajele primite de la proprii vecini. [14]

De ce este MIS-ul o problemă relevantă pentru problema localității? Vom nota cu Δ gradul maxim al unui nod din graf. Există 4 probleme clasice și centrale în

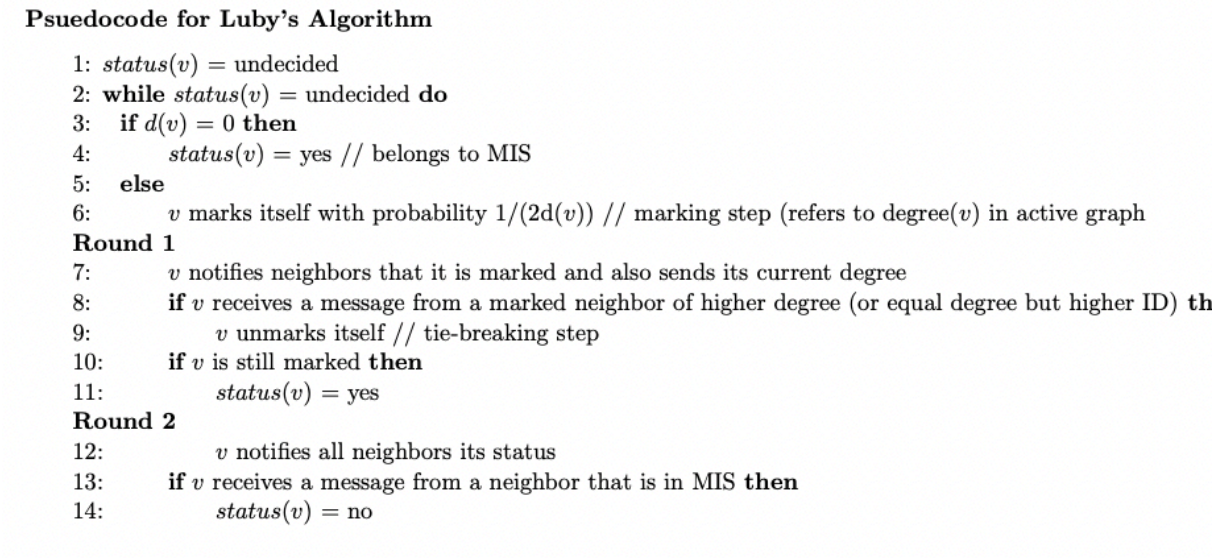

Figura 3.1: Pseudocodul algoritmului lui Luby[16]

problema localității, iar fiecare dintre acestea se reduce prima: [14]

1. Mulțimea independentă maximală de noduri.
2. Mulțimea independentă maximală de muchii.
3. Colorarea nodurilor cu $(\Delta + 1)$ culori, adică asignarea unei culori din mulțimea $1, 2, ... \Delta + 1$ fiecărui nod astfel încât să nu existe două noduri adiacente de aceeași culoare.
4. Colorarea muchiilor cu $(2\Delta - 1)$ culori, adică asignarea unei culori din mulțimea $1, 2, ..(2\Delta - 1)$ fiecărei muchii astfel încât să nu existe două muchii adiacente de aceeași culoare.

3.2 Algoritmul lui Luby

Algoritmul lui Luby [18], publicat în 1986 a fost primul algoritm distribuit randomizat care să rezolve problema găsirii unei mulțimi independente maximale. Acesta se bazează pe un agloritm de tip Monte Carlo, adică un algoritm randomizat al cărui rezultat poate fi, cu probabilitate de obicei mică, incorect. [7]

Algoritmul lui Luby folosește asignări randomizate ale nodurilor în MIS, însă evită posibilitatea unui rezultat incorect prin eliminarea din graf a nodurilor adăugate și a vecinilor lor și rerularea algoritmului.

3.2.1 Implementare

Pseudocodul prezentat în Figura 3.1 oferă multă libertate în privința detaliilor de implementare.

În general, algoritmul se împarte în câteva faze, care se repetă la fiecare rundă (rulare a while-ului din pseudocod):

- Faza 1 - liniile 3-6: Aderarea, în mod aleatoriu cu o probabilitate de $\frac{1}{2*d(v)}$, unde $d(v)$ este gradul nodului.
- Faza 2 - liniile 7-11: Fiecare nod primește informații despre statusul vecinilor săi. Cum nu putem avea două noduri vecine ca parte din MIS există o "competiție" între noduri: dintre două noduri vecine, cel cu probabilitatea mai mică este eliminat din MIS, adică nodul cu cei mai mulți vecini. Acesta revine la statusul NOT DECIDED.
- Faza 3 - liniile 12-14: Comunicarea noilor statusuri între vecini. Fiecare nod care are un vecin *IN MIS* ia statusul de *NOT IN MIS*.
- Faza 4: Comunicarea noilor statusuri între vecini. Fiecare nod elimină nodurile *NOT IN MIS* din lista sa de vecini.

Prima implementare: sincronă

Pentru a respecta modelul LOCAL și pentru a asigura integritatea rundelor, prima versiune a algoritmului implementată de mine este sincronă: nodurile sincronizează începutul fiecărei runde.

După fiecare rulare a celor 4 faze, nodurile se sincronizează cu un nod ***Coordonator***. Atunci când Coordonatorul primește confirmări de finalizare a rundei curente de la toate nodurile din graf, pornește următoarea rundă.

Prin acest mecanism ne asigurăm că toate nodurile se află la aceeași rundă și evităm situația comunicării unor informații decalate din cauza lipsei sincronizării. De asemenea, se reduce memoria utilizată de fiecare nod. La finalul celor 4 faze, toate informațiile primite de la vecini pot fi suprascrise.

A doua implementare: asincronă

Această implementare permite anumite diferențe minore și temporare de rundă între noduri. Pentru a suporta acest lucru, fiecare nod categorizează informațiile primite de la vecini în funcție de runda expeditorului. Astfel, crește complexitatea de memorie: fiecare nod ține în memorie toate informațiile primite în timpul rulării programului de la vecinii săi.

De ce este nevoie de reținerea acestor informații? Un exemplu concret este chiar pornirea rulării algoritmului în fiecare nod. Apelarea *start(ps)* nu poate porni toate nodurile în același moment. Astfel, poate avea loc următorul scenariu:

- Nodul A și nodul B sunt vecini.
- Nodul A pornește algoritmul și este adăugat aleatoriu la MIS. Trimite această informație către nodul B.
- Nodul B pornește algoritmul și procesează mesajul de la nodul A. Chiar dacă B nu a început încă runda, va avea nevoie de această informație, așa că o salvează în memorie.

3.2.2 Optimizări asupra algoritmului asincron

Amânarea mesajelor până când rundele sunt egale - optimizarea memoriei fiecărui nod

Pentru a scăpa de diferențele temporare de rundă putem amâna procesarea mesajelor până când runda nodului curent este egală cu runda mesajului. Acest lucru este însă foarte costisitor din punct de vedere al timpului de rulare. Pentru un graf cu 1000 de noduri, timpul de rulare a crescut de la 2s la 60s folosind această implementare.

Cum numărul de runde este relativ mic (11 runde pentru 1000 de noduri), iar numărul de vecini al fiecărui nod este relativ mic, omiterea utilizării acestei optimizări de memorie din cauza complexității adiționale de timp este justificată.

Limitarea memoriei la runda curentă și runda următoare

Dacă reducerea memoriei la runda curentă este prea costisitoare din punct de vedere al performanței, putem să limităm memoria suplimentară la runda curentă și runda următoare.

Astfel, fiecare nod reține două liste, una pentru runda curentă și una pentru runda viitoare. La finalul rundei listei curente i se asignează lista viitoare, iar lista rundei viitoare devine goală.

```
1 desires_current = desires_future
2 desires_future = []
```

Deciderea cât de repede posibil

După ce un nod decide aleatoriu să facă parte din MIS, trebuie să afle dacă vecinii săi au fost adăugați aleatoriu sau nu.

Pseudocodul indică o abordare eficientă, în care fiecare nod își schimbă marcajul în momentul în care află despre un vecin în MIS.

Implementarea curentă așteaptă informațiile de la toți vecinii înainte să le analizeze și să decidă dacă nodul rămâne în MIS sau nu. Este suficient un singur vecin în MIS cu *desire_level* mai mare pentru a scoate nodul curent din MIS. Așadar, vom optimiza condiția de așteptare astfel:

- Doar nodurile ale căror status este 'IN MIS' vor aștepta aceste mesaje de la vecini. Nodurile NOT DECIDED nu se pot folosi de această informație.
- Nodurile IN MIS vor aștepta până când, fie primesc mesaj de la un alt nod din MIS cu *desire_level* mai mare, fie până când primesc mesajele de la toți vecinii.

Astfel, condiția se schimbă din:

```
await(len(desires_current) == expect_confirmation_from)
```

în:

```
if MIS == "IN MIS":
    await(len(desires_current) == expect_confirmation_from) or
    len(list(filter(lambda x: x[1] == 'IN MIS' and x[0] >=
     ↪ desire_level, desires_current))) != 0
```

Analog pentru pasul în care un nod NOT DECIDED așteaptă un mesaj IN MIS ca să fie marcat ca NOT IN MIS. Acesta se modifică din:

```
await(len(propagate_mis_current) == expect_confirmation_from))
```

în:

```
await(len(propagate_mis_current) == expect_confirmation_from) or 'IN
 ↪ MIS' în propagate_mis_current)
```

Eliminarea confirmării

În prima implementare a algoritmului am adăugat, pentru siguranță, confirmări ale primirii și procesării mesajelor de către fiecare vecin. Astfel, când un nod trimitea mesaje vecinilor, pe lângă faptul că aștepta să primească răspuns de la toți vecinii, aștepta și ca fiecare vecin să îi confirme că a primit și procesat mesajul trimis.

```
await(each(p în list(neighbors.values()),
 ↪ has=received(('confirmation1', round_nr), from_= p)))
```

Nevoia pentru această confirmare poate fi eliminată însă. Algoritmul descris de Luby nu include mecanisme de fault-tolerance. Programul implementat de mine este corect chiar și în cazul unor întârzieri ale mesajelor, dar nu și pentru pierderea mesajelor. Chiar și asa, algoritmul este menit să exemplifice corectitudinea probabilistică, nu și fault tolerance-ul. Pentru a ne asigura că mesajele nu se pierd, putem folosi o configurare internă a DistAlgo. Configurarea "reliable" impune utilizarea protocolului TCP pentru comunicarea mesajelor.

```
config(channel = 'reliable')
```

3.2.3 Complexitate

Complexitatea algoritmului sincron este $O(\log n)$ (cu n fiind numărul de noduri al rețelei) deoarece algoritmul termină rularea în $O(\log n)$ runde, cu probabilitate mare. [24]

3.3 Algoritmul lui Ghaffari

Algoritmul lui Ghaffari [13] a fost descris în 2015 și folosește o versiune modificată a algoritmului lui Luby ca bază și integrează diferite alte tehnici și algoritmi rafinați pentru a obține o complexitate îmbunătățită de $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$. Acest algoritm se bazează be ideea de fragmentare a grafului, fenomen ce se întâmplă atunci când, eliminând progresiv nodurile din MIS și vecinii lor, rămânem cu grafuri mici, izolate. Algoritmul lui Ghaffari rulează o variație a algoritmului lui Luby în stadiul de pre-fragmentare și folosește mai apoi o combinație de alți algoritmi pentru stadiul post-fragmentare.

Tehnicile și algoritmii utilizați post-fragmentare sunt foarte complecși și totodată incomplet descriși în lucrarea ce descrie algoritmul, astfel încât vom continua să explorăm variația algoritmului lui Luby folosită în stadiul pre-fragmentare.

3.3.1 Modificarile aduse algoritmului lui Luby

Variația algoritmului lui Luby propusă de Ghaffari [13] este o îmbunătățire a algoritmului lui Luby, care aduce două modificări:

- Probabilitatea cu care nodul este adăugat în MIS depinde acum de probabilitățile vecinilor din runda precedentă.
- Un nod rămâne în MIS doar dacă niciunul dintre vecinii săi nu a fost adăugat aleatoriu la MIS. Acest lucru crește numărul necesar de runde deoarece nu se mai aplică selecția celui mai bun nod adăugat local, ci se bazează pe rafinarea probabilităților cu scopul obținerii unei adăugări aleatorii corecte.

3.3.2 Implementare

Algoritmul lui Ghaffari modifică fazele 1 și 2 ale algoritmului lui Luby, păstrând însă identice fazele 3 și 4. Astfel, fazele 1 și 2 devin:

- Faza 1: Calcularea probabilității cu care fiecare nod va fi adăugat în MIS, bazat de probabilitățile vecinilor din runda precedentă. Aderarea, în mod aleatoriu, la MIS.
- Faza 2: Comunicarea statusului între vecini. Un nod rămâne în MIS doar dacă niciun alt vecin al său nu a fost adăugat în MIS.

La fel ca în cazul algoritmului lui Luby, prima implementare a algoritmului lui Ghaffari este parțial sincronă, folosindu-se de același proces ***Coordonator***. A doua implementare este asincronă și construiește pe varianta optimizată a algoritmului lui Luby care implementează Limitarea memoriei la runda curentă și runda următoare, Deciderea cât de repede posibil și Eliminarea confirmării, discutate mai sus.

Faza 1: Calcularea probabilității și aderarea la MIS Spre deosebire de algoritmul lui Luby, unde probabilitatea cu care un nod era adăugat în MIS era statică (în funcție de numărul de vecini) algoritmul lui Ghaffari propune o probabilitate care se modifică dinamic în fiecare rundă.

Vom nota valoarea probabilității cu care un nod v intră în MIS în runda x cu $p_x(v)$. Pentru runda inițială $p_0(v) = 1/2$. În fiecare rundă, fiecare nod trimite vecinilor valoarea probabilității sale. La finalul rundei, fiecare nod calculează suma acestor valori, pentru runda r, $S_r(v) = \sum_{u \in N(v)} p_r(u)$. Această valoare $S_r(v)$ va fi folosită în runda $r+1$ pentru calcularea probabilității cu care nodul v intra în MIS astfel:

$$p_{r+1}(v) = \begin{cases} p_r(v)/2 & S_r(v) \geq 2 \\ min(2 * p_r(v), 1/2) & S_r(v) < 2 \end{cases}$$

Sau, în implementare, unde $previous\_sum = S_r(v)$ și $desire\_level = p_{r+1}(v)$:

```
if previous_sum >= 2:
    desire_level = desire_level / 2
else:
    desire_level = min(2 * desire_level, 0.5)
```

3.3.3 Optimizări

Reținerea sumei probabilităților rundei precedente Spre deosebire de algoritmul lui Luby, pentru acest algoritm avem nevoie să cunoaștem și informații de la vecinii din runda precedentă pentru a putea decide probabilitatea curentă a nodului de a intra în MIS. Putem reține această valoare într-o variabilă, înainte de a reinițializa listele $desires\_current$ și $desires\_future$:

```
previous_sum = sum([x[0] for x în desires_current])
desires_current = desires_future
desires_future = []
```

3.3.4 Complexitate

Complexitatea de timp a algoritmului lui Ghaffari, folosind tehnicile de eficientizare din faza post-fragmentare, este $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$, unde Δ este gradul grafului, adică cel mai mare grad al unui nod din graful ce reprezintă rețeaua. Intuitiv, primul termen al complexității provine din rularea algoritmului prezentat mai sus, timp de $O(\log \Delta)$ runde. Al doilea termen al complexității reprezintă faza post-fragmentare a algoritmului, care folosește diferiți alți algoritmi. [13]

3.4 Comparații

3.4.1 Generarea grafurilor pentru testare

Pentru compararea celor doi algoritmi avem nevoie să generăm grafuri relevante pentru rularea algoritmilor. Pentru aceasta, vom folosi modelul de generare aleatorie a grafurilor numit Watts-Stogatz. Acesta generează grafuri care simulează relațiile din lumea reală, în special ideea că oamenii sunt foarte conectați. Teoria celor 6 grade de separare [8] spune că oricare doi oameni sunt la o distanță socială de maxim 6 grade. Acest efect se resimte în viata reală și este exprimat în general prin sintagma "Ce lume mica!". De aici vine și numele modelului de generare a grafurilor Watts-Stogatz pentru grafuri Small-World.

Pentru a genera grafurile vom folosi pachetul de Python NetworkX. Acesta conține instrumente pentru crearea, manipularea și studierea rețelelor complexe. [20] Una dintre metodele oferite de pachetul NetworkX este *connected_watts_strogatz_graph*, care generează un graf aleatoriu small world, conex, și primește următorii parametri:

- n - numărul de noduri care vor fi generate
- k - fiecare nod va fi conectat la k dintre vecini
- p - probabilitatea ca o muchie să fie schimbată
- *tries* - numărul de încercări pentru a genera un graf conex

Folosind acest pachet vom genera 10 grafuri distincte de 1000 de noduri.

3.4.2 Parametri de comparare

Parametrii de comparare folosiți sunt:

- timpul total al rulării algoritmului
- numărul de runde necesare pentru ca algoritmul să se termine
- numărul de runde necesare pentru ca majoritatea nodurilor să decidă dacă fac parte din MIS sau nu

Pentru a măsura toate aceste date pentru fiecare rulare a algoritmului, vom avea nevoie de două etape:

- Etapa 1: Rularea algoritmului și strângerea datelor despre nodurile individuale. În context distribuit, putem obține doar informații despre performanța locala a nodurilor.
- Etapa 2: Analizarea datelor și agregarea lor la nivel global. Prin sintetizarea datelor locale putem obține informații globale despre performanța.

Etapa 1: Obținerea datelor locale despre performanța Fiecare nod reține timpul înainte de prima sa rundă a algoritmului - $start\_time$.

După ce un nod află dacă se află sau nu în MIS, reține valoarea $finish\_time$. Înainte de finalizarea rulării sale a algoritmului scrie în fișierul de rezultate decizia finală (dacă este IN MIS sau NOT IN MIS), iar în fișierul de date valorile $start\_time$, $finish\_time$ și $finish\_time - start\_time, round\_nr$.

Etapa 2: Agregarea datelor locale despre performanța După rularea algoritmului, avem un script care analizează datele strânse și:

1. Validează corectitudinea MIS-ului generat de fiecare rulare.
2. Agregă informațiile locale și generează grafice de comparare a celor doi algoritmi.

Verificarea corectitudinii rezultatelor Pentru verificarea corectitudinii rezultatelor am implementat o funcție, ilustrată in Codul sursă 3.1, care testează dacă MIS-ul găsit este valid pentru graful dat. Funcția verifică, pentru fiecare nod:

- dacă nodul face parte din MIS, niciun alt vecin nu poate face parte din MIS (liniile 7-9)
- dacă nodul nu face parte din MIS, cel puțin un vecin trebuie să facă parte din MIS (liniile 10-12)

```
for key in initial_neighbors.keys():
    all = False
    for value in initial_neighbors[key]:
        a = key în mis
        b = value în mis
        all = all or b
        if a and b:
            # MIS-ul este incorect
            return False
    if key not in mis and all == False:
        # MIS-ul este incorect
        return False
    return True
```

Cod sursă 3.1: Verificarea corectitudinii MIS-ului

Rularea comparațiilor Pentru rularea comparațiilor vom folosi un script (Codul sursă 3.2) care să ruleze fiecare algoritm pe aceleași date de test. Vom testa de 3 ori (linia 6) pentru fiecare date de test (linia 4), iar rezultatul final va fii media celor 3 rulări pentru a elimina posibilitatea unei scăderi de performanță temporară a mașinii.

```
# i = indicele grafului
# j = indicele rularii
size=1000
for (( i=0; i <10; i++ ))
do
    for (( j=0; j <3; j++ ))
    do
        python -m da lubys_async.da "size" "i" "j"
        python -m da ghaffari_async.da "size" "i" "j"
    done
done
```

Cod sursă 3.2: Script care rulează cei doi algoritmi pentru fiecare dintre cele 10 grafuri

3.4.3 Rezultatele comparațiilor

Comparația performanței totale a algoritmilor asincroni Am comparat performanța totală a algoritmilor asincroni pe 10 grafuri aleatorii de 1000 de noduri. Pentru a

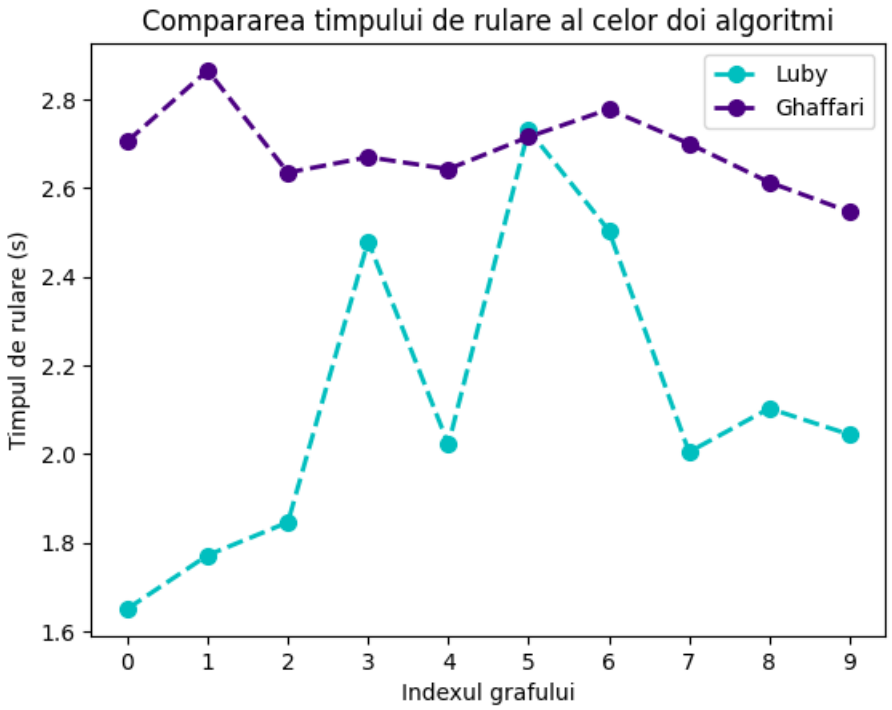

Figura 3.2: Compararea performanței algoritmilor lui Luby și lui Ghaffari pe 10 grafuri aleatorii de 1000 de noduri

normaliza variațiile de performanța, am rulat algoritmii de 3 ori pentru fiecare graf și am folosit media acestor performanțe în comparația finală.

Performanța algoritmului lui Luby este mai bună decât cea a algoritmului lui Ghaffari în toate cazurile (Figura 3.2) . Acest lucru se explică prin faptul că implementarea algoritmului lui Ghaffari reprezintă etapa pre-fragmentare a algoritmului de complexitate $O(\log \Delta) + 2^{O(\sqrt{\log \log n})}$.

În varianta implementată a algoritmului lui Ghaffari, pentru ca un nod să intre în MIS este nevoie ca niciunul dintre nodurile vecine să nu fi fost asignat aleatoriu în MIS. Algoritmul mizează pe rafinarea probabilității cu care un nod intră în MIS și nu pe negocierile ce au loc între noduri. Acest aspect duce la mărirea numărului de runde necesare pentru rezolvarea problemei. Evident, creșterea numărului de runde scade performanța. Astfel, rularea până la final a algoritmului pre-fragmentare induce o complexitate mai mare decât a algoritmului lui Luby.

Implementarea poate fi eficientizată prin oprirea fazei de pre-fragmentare după $\log \Delta$ runde și adăugarea tehnicilor post-fragmentare.

Comparația numărului total de runde pentru algoritmii asincroni Faptul că algoritmul lui Ghaffari generează un număr mai mare de runde este evident și în graficul din Figura 3.3 care analizează media numărului de runde pentru rulările pe fiecare graf.

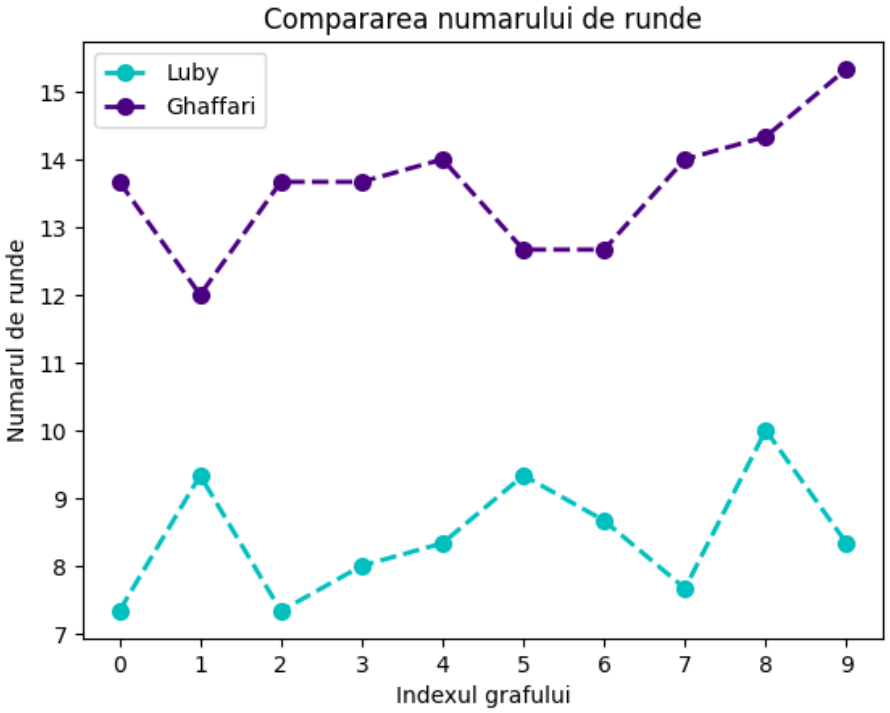

Figura 3.3: Compararea numărului total de runde necesare pentru rezolvarea problemei MIS folosind algoritmul lui Luby, respectiv algoritmul lui Ghaffari, pentru 10 grafuri aleatorii de 1000 de noduri

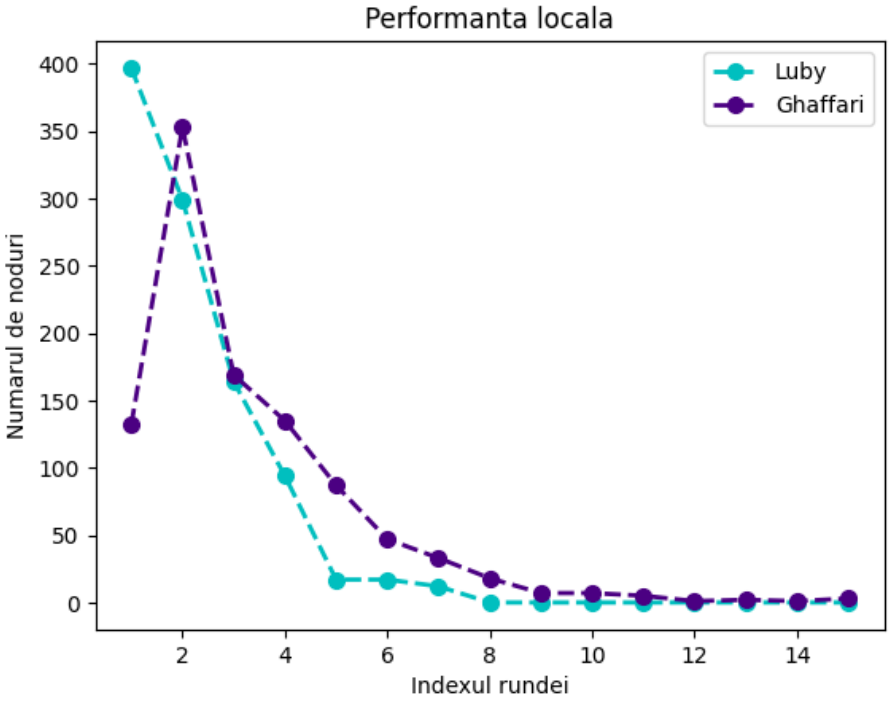

Figura 3.4: Numărul de noduri rezolvate în fiecare rundă de algoritmul lui Luby, respectiv algoritmul lui Ghaffari, pentru un graf de 1000 de noduri

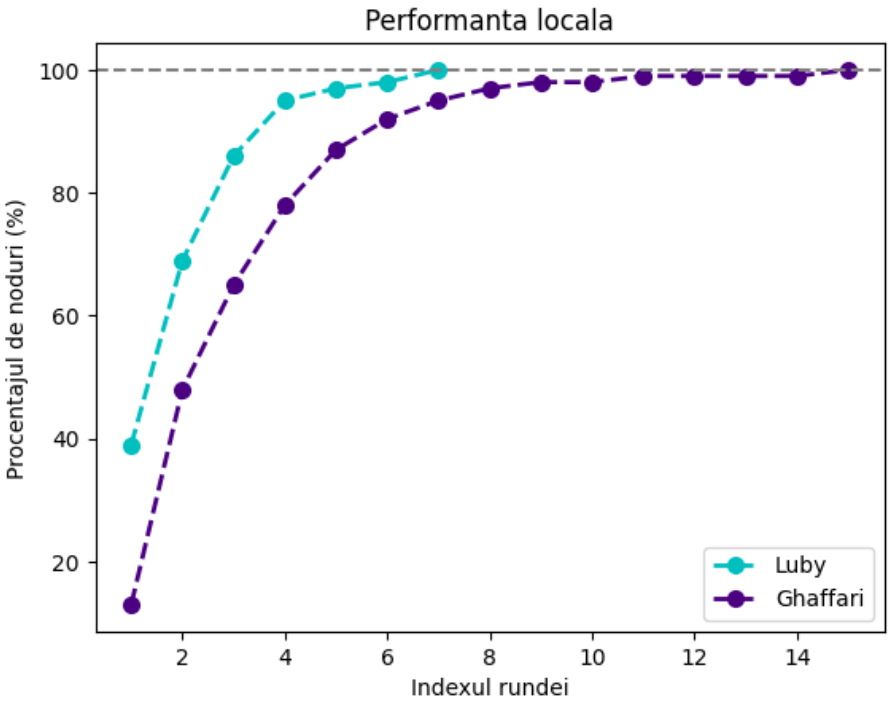

Figura 3.5: Procentajul de noduri rezolvate după fiecare rundă pentru rularea algoritmului lui Luby, respectiv algoritmului lui Ghaffari, pentru un graf de 1000 de noduri

Comparația performanței locale Atunci când un nod află dacă face parte din MIS sau nu spunem că acel nod este ***rezolvat***. O altă metrică ce merită analizată este:

După câte runde majoritatea nodurilor își cunosc apartenența la MIS (majoritatea nodurilor sunt rezolvate)?

Pentru această comparație vom analiza Figura 3.4. În prima rundă algoritmul lui Luby își bazează probabilitățile pe gradul unui nod, în timp ce algoritmului lui Ghaffari oferă o probabilitate uniformă de $\frac{1}{2}$ pentru fiecare nod. Acest fapt, împreună cu negocierea intrării în MIS, permite algoritmului lui Luby să dubleze valoarea nodurilor rezolvate de algoritmul lui Ghaffari în prima runda. Ajustarea probabilităților propusă de algoritmul lui Ghaffari reușește însă să accelereze rezolvarea după cea de-a doua rundă. Chiar dacă pornește mai încet și se termina mai încet, algoritmul lui Ghaffari rezolvă mai multe noduri per rundă decât algoritmul lui Luby între rundele 2 și 9 .

Comparația performanței locale în procentaj Pentru această comparație vom analiza Figura 3.5. Datorită faptului că în algoritmul lui Ghaffari este necesară ajustarea probabilităților, prima rundă a algoritmului nu rezolvă foarte multe noduri (13%, fata de 39% pentru algoritmul lui Luby). De asemenea, se remarcă dificultatea algoritmului Ghaffari de a trece de la un procentaj de 97% la un procentaj de 100% (8 runde) comparativ cu algoritmul lui Luby (3 runde).

Capitolul 4

Arborele de acoperire minim

În teoria grafurilor, un ***arbore de acoperire*** T al unui graf neorientat G este un subgraf al lui G care este arbore și care include toate nodurile din G.

Un ***arbore minim de acoperire*** (MST) T este un arbore de acoperire pentru graful neorientat, conex și ponderat G în care suma muchiilor este minimă. [30]

În context distribuit, problema arborelui minim de acoperire are aplicații în mai multe contexte: difuzarea eficientă a mesajelor într-o rețea, load balancing [26], clustering [9], algoritmi de consens [34], etc.

4.1 Problema congestiei

O rețea poate avea limitări din punct de vedere al capacității, prin mărginirea strictă a volumului de mesaje pe care le poate primi un nod sau o muchie. Comunicarea într-o astfel de rețea poate duce la congestie, adică la suprasolicitarea rețelei în punctele (nodurile sau muchiile) vulnerabile.

Evitarea acestei suprasolicitări a condus la enunțarea problemei congestiei și la crearea unui întreg subdomeniu de cercetare în cadrul algoritmilor distribuiți.

Într-un context ideal, descrierea unui algoritm distribuit ar răspunde la întrebarea: *"Cât de repede putem rezolva o problema în context distribuit?"* Prin prisma congestiei, însă, întrebarea la care trebuie să răspundem este: *"Cât de repede putem rezolva o problemă în context distribuit, cu limitări de capacitate?"*[14]

Modelarea matematică - modelul CONGEST Pentru a modela matematic problema congestiei, s-a definit modelul CONGEST, cu cele două variații ale sale: E-CONGEST și V-CONGEST. E-CONGEST limitează capacitatea muchiilor, în timp ce V-CONGEST limitează capacitatea nodurilor.

Astfel, modelul E-CONGEST este reprezentat printr-un graf neorientat cu n noduri $G = (V, E)$ cu $V = \{1, 2, ..., n\}$. Există un proces pentru fiecare nod $v \in V$. Fiecare nod cunoaște valoarea lui n, propriul id, propriul index în mulțimea V și id-urile vecinilor.

Algoritmii funcționează în runde sincrone. În fiecare rundă, fiecare nod calculează anumite valori bazate pe propriile informații locale și trimite un mesaj de dimensiune B biți către fiecare vecin al său. De obicei $B = O(logn)$. La finalul rundei fiecare nod primește și procesează mesajele primite de la proprii vecini.

Analog pentru modelul V-CONGEST.

Intuiție care leagă congestia de teoria grafurilor Intuitiv, putem lega noțiunea de congestie de gradul de conexitate al grafului.

Gradul de conexitate al unui graf este numărul minim de elemente (noduri sau muchii) care trebuie eliminate pentru a separa nodurile rămase în două sau mai multe subgrafuri izolate. O ***tăietură de noduri*** a unui graf conex este o mulțime de noduri a căror îndepărtare determină graful să nu mai fie conex. ***Conexitatea pe noduri*** este dimensiunea unei tăieturi minime. [29]

Aceste noțiuni caracterizează capacitatea unui graf de a transmite informație.

De ce este MST-ul o problemă relevantă pentru problema congestiei? MST-ul poate fi folosit pentru a reduce costul comunicării într-o rețea. Putem reprezenta o rețea reală sub forma unui graf ponderat, unde fiecare pondere reprezintă costul comunicării între cele două noduri pe care le conectează (nu capacitatea maximă de procesare a muchiei). Calculând MST-ul, ne asigurăm că minimizăm costul total de transmitere a datelor, ceea ce reduce traficul în rețea și îmbunătățește congestia.

4.2 Algoritmul GHS

Algoritmul GHS (Gallager, Humblet și Spira)[12] a fost publicat în 1983 și stă la baza tuturor algoritmilor îmbunătățiți pentru problema găsirii unui MST. Articolul publicat de Gallager, Humblet și Spira propune un algoritm greedy, distribuit și asincron pentru determinarea MST-ului unui graf.

Numim ***fragmentul unui MST*** un subarbore al unui MST, adică o submulțime de muchii și noduri conectate ale acelui MST.

Numim ***muchia de pondere minimă care iese din fragment*** (MWOE) o muchie care are exact un nod în componentă.

Într-un graf ponderat G = (V, E), cu V mulțimea nodurilor și E mulțimea muchiilor, unde $E_1 \in E$, numim $w(E_1)$ ***ponderea muchiei*** E_1 în graful G.

Algoritmul se bazează pe câteva afirmații.

Afirmația 1 Fie graful ponderat, conex, neorientat G și fie T MST-ul asociat lui. Fie A și B două fragmente distincte și disjuncte ale lui T astfel încât $A \bigcup B \neq T$ (uniunea dintre A și B nu acoperă tot arborele T). Fie E_{min} MWOE-ul pentru fragmentul A, care

are un nod în A și un nod în B. Unirea fragmentelor A și B prin muchia E_{min} va fi tot un fragment al MST-ului.

Demonstrație: (metoda reducerii la absurd) Presupunem că E_{min} nu este în T.

$\iff$ E_{min} creează un ciclu cu alte muchii ale arborelui T.

$\iff$ Cel puțin una dintre aceste muchii din ciclu, E_x, este o muchie care iese din fragment. Cum E_{min} este MWOE, $w(E_x) \geq w(E_{min})$. Astfel, eliminând muchia E_x din MST și adăugând E_{min}, formam un nou MST T' cu ponderea totală mai mică decât a lui T.

$\iff$ Contrazice ipoteza, T era minim.

Afirmația 2 Dacă toate muchiile grafului conex au ponderi diferite, MST-ul este unic.

Demonstrație: (metoda reducerii la absurd) Presupunem că exista 2 MST-uri T_1 și T_2 într-un graf cu ponderi diferite. Fie e muchia minimă astfel încât $e \in T_1$ și $e \notin T_2$.

$\iff$ Muchia e formează un ciclu cu muchiile din T_2.

$\iff$ Există cel puțin o muchie $e' \in T_2$ astfel încât $e' \notin T_1$.

$\iff$ Toate ponderile grafului G sunt diferite, $\Rightarrow w(e) < w(e')$.

$\iff$ Putem scoate muchia e' din T_2 și adaugă muchia e, obținând un MST T_2' de pondere mai mică.

$\iff$ Contrazice ipoteza, T_2 era minim.

Afirmația 3 Orice graf ponderat poate fi transformat într-un graf ponderat cu ponderi distincte.

Pentru a transforma un graf cu muchii identice într-un graf cu muchii distincte, alipim indicele nodurilor adiacente unei muchii la finalul ponderii sale. MST-ul pentru graful modificat va fi valid pentru graful inițial.

Pentru fiecare muchie, adaugăm la final id-urile nodurilor sale. Spre exemplu, muchia 13 între nodurile 21 și 30 devine muchia 132130, iar muchia de pondere 1 între 3 și 2 devine 010302. Trebuie să ne asiguram că numărul de cifre adăugat la finalul oricărei ponderi este același, pentru a păstra compararea mai întâi după valoarea inițială a muchiei.

4.2.1 Descrierea algoritmului

Se presupune ca rețeaua este sigură, mesajele ajung la destinatar după un timp finit, fără erori de rețea. Acestea sunt procesate printr-o coadă. În cadrul limbajului DistAlgo, aceasta înseamnă configurarea:

```
config(channel = {'reliable', 'fifo'})
```

Această configurare forțează folosirea protocolului Transmission Control Protocol (TCP), care asigură o comunicare sigură între nodurile algoritmului.

Intuiție asupra algoritmului

Afirmația 3 ne asigură că putem rula algoritmul GHS pe orice graf ponderat, prin transformare. Folosind Afirmația 2, deducem că MST-ul găsit de algoritmul GHS pe un graf transformat este unic.

Algoritmul se bazează pe ideea de fragmente. La început, fiecare nod este propriul său fragment. Fragmentele se unesc prin MWOE până se ajunge la un singur fragment, folosind Afirmația 1.

MST-ul conține toate nodurile unui graf. Astfel, inițial, fiecare nod reprezintă un fragment unic. Unirea acestor fragmente va forma MST-ul final, cu toate nodurile grafului. Fragmentele se unesc de-a lungul MWOE-urilor, formând noi fragmente ale MST-ului.

Fiecare fragment are un id unic, definit ca valoarea unică a muchiei centrale fragmentului, aleasă după anumite criterii. De o parte și de alta a muchiei centrale se află cele două noduri care sunt lideri ai fragmentului.

Atunci când două fragmente se fuzionează, cel puțin unul dintre fragmente trebuie să își actualizeze id-ul, pentru a putea fi identificat ca identic cu celălalt fragment.

Un punct cheie al algoritmului este cazul unui fragment foarte mic A care fuzionează cu un fragment foarte mare B. Algoritmul impune că, întotdeauna, fragmentul mai mic va fi absorbit de fragmentul mai mare, preluându-i id-ul. Acest mecanism eficientizează fuzionarea.

Aici intervine problema măsurării dimensiunii fragmentelor, pentru a compara dimensiunile fragmentelor ce vor fuziona.

Măsurarea în timp real a dimensiunii unui fragment O primă idee este ca fiecare fragment să rețină numărul său de noduri, iar în momentul fuzionării fragmentele își vor comunica această valoare. Intervine însă impracticalitatea unei măsurători exacte a numărului de noduri în context distribuit, în care dimensiunea fragmentelor se schimbă dinamic. Schimbarea rapidă a numărului de noduri al unui fragment poate duce la inconsistențe în interiorul aceluiași fragment.

În schimb, vom folosi o noțiune care să aproximeze numărul de noduri ale unui fragment, aceea de nivel. Nivelul unui fragment funcționează astfel:

- Un fragment de dimensiune 1 (care conține un singur nod) are nivel 0.
- Atunci când un fragment de nivel mai mic este absorbit într-un fragment de nivel mai mare, acesta preia nivelul fragmentului mai mare.
- Atunci când două fragmente de nivel egal vor să fuzioneze, nivelul lor este incrementat cu 1. Astfel, din 2 fragmente de nivel 2 obținem un fragment de nivel

3, incrementarea nivelului fiind logaritmic proporțională la creșterea fragmentelor. Acest proces se numește unire.

4.2.2 Implementare

Fiecare nod se poate află în 3 stări:

- *Sleeping*: nodul nu participă la algoritm încă.
- *Find*: scopul principal al nodului este să găsească MWOE-ul și să îl trimită mai departe către rădăcină.
- *Found*: nodul a găsit MWOE-ul și l-a trimis către rădăcină.

În general, informațiile proprii vor avea un N la final, pentru deosebirea de informațiile vecinilor (SN - starea nodului curent, S - starea primită de la vecin).

De asemenea, fiecare muchie se poate afla în 3 stadii:

- *Basic*: Nu s-a decis încă dacă muchia face parte din MST sau nu.
- *Rejected*: Muchia nu face parte din MST, ea conectează 2 noduri care erau deja în MST, formând un ciclu.
- *Branch*: Muchia face parte din MST.

Algoritmul folosește câteva tipuri de mesaje predefinite pentru comunicare: *Connect*, *Initiate*, *Test*, *Accept*, *Reject*, *Report* și *ChangeRoot*.

4.2.3 Rolul lui *Connect*

```
msg = ("Connect", L, source_index)
```

Mesajul *Connect* conține următoarele informații:

- L : nivelul nodului emițător
- *source_index*: id-ul nodului emițător

Emiterea Atunci când un fragment F_1 identifică muchia AB ca fiind MWOE, cu $A \in F_1$, nodul A trimite nodului B mesajul *Connect*, pentru a-l înștiința de intenția de fuzionare de-a lungul acestei muchii.

Receptarea Atunci când un nod $A \in F_1$ primește un mesaj *Connect* din partea nodului $B \in F_2$, decide răspunsul astfel:

- Dacă L este strict mai mic decât nivelul nodului A, înseamnă că un fragment mai mic dorește să fie absorbit de F_1. Încep demersurile de absorbire a nodului B.
- Dacă L este egal cu LN:
  - Dacă muchia AB este MWOE pentru F_2, dar nu și pentru F_1, se decide amânarea procesării acestui mesaj și acesta revine în coada de mesaje primite de A, cu prioritate first-in-first-out. Este posibil ca într-o procesare viitoare nivelul lui F_1, LN, să fi crescut, și astfel F_2 să fie absorbit instant.
  - Dacă muchia AB este MWOE și pentru F_1 și pentru F_2, avem de-a face cu două fragmente de nivel egal ce vor să se unească de-a lungul muchiei AB. Încep demersurile de unire a celor două fragmente (*Initiate*).

Un caz particular este nodul de nivel 0. Atunci când un astfel de nod se trezește, cunoaște deja toate valorile muchiilor adiacente fragmentului său. Astfel, nu poate decât să își facă cunoscută intenția de a se conecta la cel mai apropiat fragment, trimițând un mesaj *Connect* către acesta.

4.2.4 Rolul lui *Initiate*

```
msg = ("Initiate", L, F, S, source_index)
```

Mesajul *Initiate* conține următoarele informații:

- L: nivelul nodului emițător
- F: id-ul fragmentului din care face parte nodul emițător
- S: starea fragmentului din care face parte nodul emițător
- *source_index*: id-ul nodului emițător

Emiterea Mesajul *Initiate* este trimis atunci când:

- Fragmentul F_1 vrea să absoarbă fragmentul F_2, și își impune propriul nivel, id-ul și starea asupra fragmentului F_2.
- Două fragmente de nivel egal vor să se unească, făcând schimb de mesaje *Initiate* care vor fi propagate în ambele fragmente. Valorile pentru L, F și S sunt agreate de ambele fragmente astfel:
  - $L' = L + 1$: două fragmente de nivel L formează un fragment de nivel $L + 1$

- $F = w(MOWE)$: id-ul fragmentului final este valoarea MWOE-ului dintre F_1 și F_2
- $S = Find$: după formarea unui nou fragment, reîncepe procedura de căutare a următorului MWOE

Receptarea Atunci când un nod $A \in F_1$ primește mesajul $Initiate$, aceasta are multiple roluri:

- Impunerea unor valori L, F și S pentru nivelul, id-ul și starea nodului receptor.
- Atribuirea valorii lui $in\_branch = source\_index$.
- Resetarea valorii MWOE-ului pentru nodul A în anticipare pentru următoarea rundă de căutare a MWOE-ului.
- Transmiterea acestor valori mai departe, de-a lungul subarborelui format de fragment. Fiecare nod reține numărul de noduri către care transmite mai departe această informație. Aceste noduri vor trebui să raporteze, prin mesajul *Report*, valoarea gasită pentru MWOE.
- Pornirea unei noi căutări a MWOE-ului atunci când starea impusă este $Find$.

Un caz particular este unirea a două fragmente de nivel egal: pentru ca aceasta să fie completă, este nevoie de interschimbarea a două mesaje de tip $Initiate$ de-a lungul MWOE-ului, ambele conținând același conținut.

4.2.5 Rolul lui $Test$

```
msg = ("Test", L, F, source_index)
```

Mesajul $Test$ conține următoarele informații:

- L: nivelul nodului emițător
- F: id-ul fragmentului din care face parte nodul emițător
- $source\_index$: id-ul nodului emițător

Emiterea Mesajul $Test$ este legat de procedura *test()* care, pentru orice nod A, analizează muchiile *Basic* ale lui A și o alege pe cea mai mică pentru a fi o potențială MWOE. De-a lungul acestei muchii trimite mesajul $Test$.

Receptarea Atunci când un nod $A \in F_1$ primește un mesaj *Test* din partea unui nod $B \in F_2$ înseamnă că nodul B încearcă să afle dacă adăugarea muchiei AB în MST ar forma un ciclu, adică dacă $F_1 = F_2$. Răspunsul la un astfel de mesaj se decide astfel:

- Dacă $L < LN$ și $F \neq FN$ nodul marchează muchia ca *Branch* și răspunde imediat printr-un mesaj *Accept*.
- Dacă $L > LN$ și $F \neq FN$ nodul amână răspunsul la acest mesaj. Chiar dacă fragmentele sunt diferite, ne dorim ca intenția de fuzionare să vina din partea fragmentului mai mic, nu din partea fragmentului mai mare. Amânăm răspunsul în cazul în care nivelul lui A crește astfel încât $L > LN$ la un moment dat.
- Dacă $F = FN$ nodul marchează muchia ca *Rejected* răspunde imediat printr-un mesaj *Reject*, fiindcă muchia leagă două noduri ale aceluiași fragment.

4.2.6 Rolul lui *Accept*

```
msg = ("Accept", source_index)
```

Emiterea Nodul $A \in F_1$ transmite mesajul *Accept* nodului $B \in F_2$ atunci când muchia AB conectează două fragmente diferite, iar nivelul lui F_2 este mai mic decât nivelul lui F_1, astfel încât are sens ca F_2 să înceapă demersurile de absorbție/unire de-a lungul muchiei AB.

Receptarea Atunci când un nod A primește mesajul *Accept* de-a lungul muchiei AB compară ponderea muchiei AB cu ponderea MWOE-ului curent și îi actualizează valoarea dacă prima este mai mică. După care încearcă raportarea MWOE-ului.

4.2.7 Rolul lui *Reject*

```
msg = ("Reject", source_index)
```

Emiterea Nodul $A \in F_1$ transmite mesajul *Reject* nodului $B \in F_2$ atunci când muchia AB conectează două noduri ale aceluiași fragment.

Receptarea Atunci când un nod A primește mesajul *Reject* de-a lungul muchiei AB marchează muchia ca *Rejected* dacă avea starea *Basic*. Dacă muchia avea starea *Branch* este evident că aceasta conectează două noduri ale aceluiași fragment.

4.2.8 Rolul lui *Report*

```
msg = ("Report", w, source_index)
```

Mesajul *Report* conține următoarele informații:

- w: valoarea MWOE-ului găsit de nodul emițător
- $source\_index$: id-ul nodului emițător

Emiterea Atunci când un nod este sigur de valoarea MWOE-ului său, emite mesajul *Report* de-a lungul muchiei in_branch, care face parte din MST și este orientată către rădăcina fragmentului.

Receptarea Înainte ca un nod să transmită, de-a lungul in_branch-ului său, valoarea MWOE-ului, trebuie să se asigure că această valoare este corectă. În acest scop, așteaptă să primească MWOE-ul de la fiecare nod către care a transmis mesajul $Initiate$. Valoarea MWOE-ului va fi minimul dintre toate valorile primite și valoarea calculată independent de nod.

Există însă mai multe tipuri de raportare:

- Raportarea de-a lungul unei muchii oarecare. În acest caz se minimizează valoarea MWOE-ului considerând valoarea existentă și valoarea primită.
- Raportarea de-a lungul muchiei rădăcină, marcată prin faptul că $in\_branch = source\_index$. În acest caz, cele două rădăcini ale unui fragment comunică și trebuie să decidă care dintre rădăcini este mai aproape de MWOE-ul final.
  - Dacă rădăcina care primește mesajul nu a terminat de evaluat valoarea propriei MWOE, acesta amână procesarea mesajului *Report*.
  - Dacă nodul receptor este mai aproape de MWOE, acesta începe procedura $Chage - root$.
  - Dacă MWOE-ul nodului receptor și MWOE-ul emițătorului au valoarea ∞ atunci putem concluziona că există un singur fragment care reprezintă MST-ul. Algoritmul se termină.

Report **de-a lungul muchiei rădăcină** Un caz particular este comunicarea de-a lungul muchiei rădăcină. Prin faptul că avem o muchie rădăcină, fiecare fragment se împarte în două secțiuni: secțiunea care raportează către nodul din stânga rădăcinii și secțiunea care raportează către nodul din dreapta rădăcinii. Fie muchia AB muchia rădăcină a fragmentului F_1. Valoarea lui in_branch pentru nodul A va fi muchia AB, iar pentru nodul B va fi muchia BA. Fiecare rădăcină va raporta celeilalte rădăcini. Când nodul A

a primit *Report*-urile de la toate nodurile sale, acesta raportează mai departe, de-a lungul muchiei rădăcină, către nodul B.

Nodul B așteaptă să primească toate raportările sale înainte să proceseze mesajul primit de-a lungul muchiei rădăcină.

Când B primește toate *Report*-urile sale trimite mai întâi propriul *Report* către A și procesează *Report*-ul primit de la A. Dacă MWOE-ul său este mai mic, B inițiază procedura *ChangeRoot*, în secțiunea sa de fragment. A urmează aceeași procedură.

În cazul în care ambele noduri au $w(MWOE)$ infinit, înseamnă că fragmentul a acoperit toate nodurile și algoritmul se încheie.

4.2.9 Rolul lui *ChangeRoot*

```
msg = ("Change root", source_index)
```

Emiterea Atunci când un nod decide că MWOE-ul se află în secțiunea sa de fragment transmite mesajul *ChangeRoot* de-a lungul muchiei ce duce la MWOE.

Receptarea Prin mesajul *ChangeRoot* se parcurge tot drumul de la rădăcină la MWOE, folosind muchia MWOE-ului pentru fiecare nod întâlnit. Când mesajul ajunge la nodul de la marginea fragmentului, acel nod trimite un mesaj *Connect* către fragmentul ce se află de cealaltă parte a MWOE-ului, pentru a își face cunoscută intenția de fuzionare.

4.2.10 Optimizări

Eliminarea răspunsului *Reject* în unele cazuri

Algoritmul GHS vine cu o optimizare în procesul de testare, prezentată în Codul sursă 4.1. În general, ca răspuns la un mesaj $Test$, un nod va răspunde printr-un mesaj *Accept* sau *Reject*. Se face însă o excepție atunci când muchia leagă două noduri ale aceluiași fragment. Fie nodurile $A, B \in F_1$, unde A primește un mesaj $Test$ din partea nodului B. Dacă și A trimisese un mesaj $Test$ pe muchia AB, către B, omitem trimiterea unui mesaj *Reject* (liniile 6-7). Când B va primi mesajul de la A va constata că nivelele și id-urile fragmentelor sunt egale și va marca și el muchia ca *Rejected*. Fiindcă B nu va primi niciodată răspuns la propriul mesaj $Test$ trimis lui A, dar știe deja că răspunsul este negativ, reîncepe procedura *test()* după marcarea muchiei.

```
1 if F == FN:
2     if branches[souce_index]["state"] == "Basic":
3         # marcheaza muchia ca Rejected
4     if test_edge != source_index:
5         # trimite mesajul Reject
```

```
    else:
        # continua să cauti MWOE-ul fara să trimiti mesajul Reject
```

Cod sursă 4.1: Eliminarea răspunsului *Reject* în unele cazuri

4.2.11 Îmbunătățiri aduse de mine

Adăugarea redundanței pentru claritate Algoritmul GHS, desi corect, include multe condiții cărora li se poate adauga redundanță pentru a îmbunătăți claritatea algoritmului. Un astfel de exemplu este răspunsul la un mesaj *Test* care în algoritm are 3 ramuri posibile, evidențiate în Codul sursă 4.2.

```
if L > LN: # condiția A
    ...  # amana procesarea
elif F != FN: # condiția B
    ...  # accepta
else: #condiția C
    ...  # refuza
```

Cod sursă 4.2: Răspunsul la un mesaj *Test* în algoritmul original GHS

Faptul că nivele fragmentelor sunt diferite, implică faptul că fragmentele sunt de asemenea diferite. Astfel, pentru claritate, am transformat condițiile adăugând redundanța din Codul sursă 4.3.

```
if F != FN and L > LN: # condiția A
    ...  # amana răspunsul
elif F != FN and L <= LN: # condiția B
    ...  # accepta
elif F == FN: #condiția C
    ...  # refuza
```

Cod sursă 4.3: Răspunsul pentru un mesaj *Test* cu redundanță adăugată

Un alt exemplu este răspunsul la mesajul *Connect*, care de asemenea are 3 ramuri posibile, ilustrate în Codul sursă 4.4.

```
if L < LN: # condiția A
    ...  # trimite Initiate
elif branches[source_index]["state"] == "Basic": # condiția B
    ...  # amana procesarea
else: #condiția C
    ...  # unirea a două fragmente egale
```

Cod sursă 4.4: Răspunsul la un mesaj *Connect* în algoritmul original GHS

Un fragment poate absorbi, prin intermediul mesajului *Initiate*, doar de-a lungul unei muchii *Basic*, așa că putem modifica cele 3 condiții, adăugând redundanța din Codul sursă 4.5.

```
if L < LN and branches[source_index]["state"] == "Basic": # condiția A
    ...  # trimite Initiate
elif L >= LN and branches[source_index]["state"] == "Basic": #
 ↪ condiția B
    ...  # amana procesarea
elif L == LN and branches[source_index]["state"] == "Branch":
 ↪ #condiția C
    ...  # unirea a două fragmente egale
```

Cod sursă 4.5: Răspunsul pentru un mesaj *Connect* cu redundanță adăugată

Oprirea proceselor

După formarea MST-ului avem nevoie de o modalitate de terminare a algoritmului care să oprească procesul fiecărui nod din DistAlgo. Algoritmul include, la primirea mesajului *Report*, condiția $if(w == best\_wt == \infty)\ then\ halt$. Construind pe baza acestei condiții în implementare, fiecare nod are propria variabilă *kill* setată inițial $False$. Astfel, în procedura *run()* fiecare nod așteaptă ca valoarea variabilei *kill* să fie True, astfel:

```
def run():
    wakeup()
    await(kill == True)
```

Atunci când fiecare dintre cele două noduri rădăcină ale MST-ului identifică împlinirea condiției $if(w == best\_wt == \infty)\ then\ halt$, pe lângă setarea variabilei $kill = True$, aceasta trimite un mesaj *Kill* pe toate muchiile sale de tip *Branch* (în afară de muchia rădăcină). Astfel, fiecare rădăcină este responsabilă pentru oprirea proceselor în secțiunea sa de arbore.

Atunci când un nod primește mesajul *Kill* verifică încă o dată că situația sa este încheiată prin condiția:

```
if find_count == 0 and test_edge == None and SN == "Found"
```

Apoi transmite mesajul *Kill* mai departe către toate muchiile sale *Branch* și își setează propria valoare $kill = True$.

Eliminarea muchiei rădăcină Pentru simplificarea algoritmului vom elimina muchia rădăcină și cele două noduri rădăcină pentru a le înlocui cu un singur nod central (rădăcină) care ia deciziile. Această modificare simplifică, măcar conceptual, algoritmul.

Pentru a implementa această modificare este nevoie să schimbăm modul în care se unesc două fragmente de nivel egal pentru că acela este momentul în care se stabilește muchia rădăcină. În varianta clasică a algoritmului, ambele noduri rădăcină trimit un mesaj *Connect* de-a lungul muchiei rădăcină. Ca răspuns, ambele vor trimite un mesaj *Initiate* de-a lungul muchiei rădăcină. În algoritmul modificat cel mai mare nod dintre cele două noduri rădăcină va deveni ***nodul central***. Doar acest nod central va trimite mesaje *Initiate* pe toate muchiile sale de tip *Branch*. Vom numi această variantă a algoritmului Single Node Root GHS (SNR-GHS).

Vom adăuga funcția *Initiate*(), ilustrată în Codul sursă 4.6, care:

- Setează valorile $best\_edge, best\_wt$ și $find\_count$ pentru începerea unei noi căutări a MWOE-ului (liniile 2-4).
- Transmite mesajul *Initiate* mai departe, pe toate muchiile de tip *Branch*, actualizând valoarea lui $find\_count$ (liniile 6-10). Funcția are grijă ca un nod să nu trimită mesajul *Initiate* și pe muchia pe care l-a primit (in_branch).
- Apelează procedura *test()* pentru căutarea MWOE-ului (liniile 11-12).

```
1  def Initiate():
2      best_edge = None
3      best_wt = math.inf
4      find_count = 0
5
6      for (iterator, branch) în list(branches.items()):
7          if branch["state"] == "Branch" and (in_branch == None or
           ↪  (in_branch != None and iterator != in_branch)):
8              send(("Initiate", LN, FN, SN, index), to =
               ↪  branch["process_id"])
9              if SN == "Find":
10                 find_count += 1
11     if SN == "Find":
12         test()
```

Cod sursă 4.6: Funcția *Initiate()*

În cazul unirii dintre două fragmente de același nivel, la primirea mesajului *Connect*, nodul central trebuie doar să ruleze Codul sursă 4.7 care incrementează nivelul (linia 1),

setează starea (linia 2) și îl marchează ca nod central al fragmentului (linia 3), care nu are nevoie de o muchie către nodul central (linia 4). Restul procesului de inițializare se realizează prin apelarea funcției *Initiate()* (linia 5).

```
1 LN += 1
2 SN = "Find"
3 i_am_root = True
4 in_branch = None
5 Initiate()
```

Cod sursă 4.7: Simplificarea codului pentru nodul central

De asemenea, răspunsul la mesajul *Initiate* va fi cel din Codul sursă 4.8, care preia valorile părintelui (liniile 1-3), setează calea către nodul central (linia 4) și înștiințează nodul că nu este nod central (linia 5). Restul procesului de inițializare se realizează prin apelarea funcției *Initiate()*.

```
1 LN = L
2 FN = F
3 SN = S
4 in_branch = source_index
5 i_am_root = False
6 Initiate()
```

Cod sursă 4.8: Simplificarea codului pentru răspunsul la mesajul *Initiate*

De asemenea, trebuie alterată modalitatea de raportare. Nu se vor mai trimite mesaje reciproce de-a lungul muchiei rădăcină, ci nodul central va aștepta raportările de la toate muchiile sale *Branch* și va decide pașii următori (terminarea algoritmului sau *ChangeRoot*).

În Codul sursă 4.9 se ilustrează varianta inițială a raportării, așa cum este prezentată în [12]. Se face distincție între mesajele *Report* de-a lungul muchiei rădăcină și celelalte mesaje (linia 7 si linia 13). De asemenea, mesajul primit de-a lungul muchiei rădăcină nu este procesat (liniile 14-15) până când se găsește MWOE-ul (liniile 17-20). Decizia între apelarea procedurii *change_root* și oprirea proceselor se face tot aici. Funcția *report()* este folosită doar pentru a testa condiția de raportare (linia 2) și a transmite mesajul *Report*.

În schimb, în Codul sursă 4.10 este prezentată raportarea pentru un singur nod central, aceasta fiind o versiune mai clară și mai scurtă. Răspunsul la mesajul *Report* este folosit doar pentru a actualiza valoarea lui *best_wt* și *best_edge* (liniile 14-16). Logica specifică nodului central se află în funcția *report()* (linia 1), fiind de altfel similară cu logica originală.

```
def report():
if find_count == 0 and test_edge == None:
    SN = "Found"
    send(("Report", best_wt, index), to = in_process)

def receive(msg = ("Report", w, source_index)):
if source_index != in_branch:
    find_count -= 1
    if w < best_wt:
        best_wt = w
        best_edge = source_index
    report()
else:
    if SN == "Find":
        send(("Report", w, source_index), to = self)
    else:
        if w > best_wt:
            change_root()
        elif math.isinf(best_wt) and math.isinf(w):
            # opreste toate procesele de pe muchiile de tip Branch
```

Cod sursă 4.9: Procedura de raportare pentru două muchii radacină

```
def report():
if find_count == 0 and test_edge == None:
    SN = "Found"
    if i_am_root == "False":
        send(("Report", best_wt, index), to = in_process)
    else:
        if not math.isinf(best_wt):
            change_root()
        else:
            # opreste toate procesele de pe muchiile de tip Branch

def receive(msg = ("Report", w, source_index)):
find_count -= 1
if w < best_wt:
    best_wt = w
    best_edge = source_index
report()
```

Cod sursă 4.10: Procedura de raportare pentru un singur nod central

Această versiune a algoritmului este mult mai ușor de înțeles și de modificat.

4.2.12 Complexitate

Utilizarea nivelelor ce se incrementează logaritmic cu dimensiunea fragmentului ajută și în calcularea complexității. Astfel, este intuitiv că algoritmul va avea cel mult $\log n$ runde, unde o rundă este etapa de căutare a MWOE-ului.

Acest raționament ne conduce la o complexitate $O(D \log n)$, unde D este diametrul MST-ului. Complexitatea algoritmului depinde de diametrul MST-ului, nu de diametrul grafului. Aceasta este o distincție semnificativă, pentru că diametrul MST-ul poate fi egal cu numărul de noduri al grafului.

Astfel, complexitatea reală, worst-case, a algoritmului GHS este $O(n \log n)$.

4.3 Algoritmul de comparație

4.3.1 Algoritmul CT

Algoritmul GHS [12], apărut în 1983, a fost primul algoritm distribuit pentru rezolvarea problemei MST. Ulterior, au apărut diverși algoritmi care îmbunătățesc algoritmul de bază. În 1985 apare lucrarea [6] care descrie algoritmul CT, un algoritm care aduce îmbunătățiri versiunii originale, cu muchia rădăcină a algoritmului. Aceste îmbunătățiri, deși ingenioase, nu ridică complexitatea worst-case peste $O(n \log n)$ și pare să aibă o complexitate a mesajelor mai mare decât cel inițial.

Algoritmul introduce însă câteva concepte utile, iar înțelegerea lui poate ajuta în înțelegerea evoluției ideilor pentru rezolvarea problemei.

Implementare

Algoritmul CT aduce două schimbări algoritmului GHS:

1. O actualizare mai rapidă a nivelului unui fragment.
2. O modificare în fuzionarea fragmentelor.

Vom explora doar prima modificare, fiind utilă în studiul algoritmilor ulteriori.

Actualizarea rapidă a nivelului unui fragment Această modificare impune reținerea unei valori exacte a numărului de noduri al fragmentului. Fiecare nod x va reține într-o variabilă *size_count* dimensiunea secțiunii de fragment care raportează către x. Intuitiv, valoarea *size_count* pentru frunzele fragmentului va fi 1. Prin intermediul procedurii *Report*, nodurile vor raporta și valoarea *size_count* și vor însuma valorile primite.

```
1 if size_count >= 2**(LN + 1):
2     LN = int(math.log2(size_count))
3     self.SN = "Find"
```

Cod sursă 4.11: Actualizarea mai rapidă a nivelului

4.3.2 Algoritmul Awerbuch

Ulterior, în 1987, Baruch Awerbuch a publicat lucrarea [4] în care descrie un algoritm de complexitate de timp $O(n)$ și de comunicare $O(m + n \log n)$.

Deși nivelele sunt o metodă eficientă de aproximare a dimensiunii unui fragment, nu reprezintă însă o metodă foarte precisă. Spre exemplu, este posibil ca un fragment F_1 să aștepte ca un fragment alăturat F_2 să își incrementeze nivelul chiar dacă dimensiunea reală a fragmentului F_2 era mai mare decât dimensiunea fragmentului F_1. Această așteptare afectează performanța.

Pentru a rezolva această problemă, Awerbuch propune două noi mecanisme de actualizare a nivelului: **Test-Distance** și **Root-Update**. Dezavantajul adus de aceste noi mecanisme este creșterea numărului de mesaje necesare. Pentru a combate acest dezavantaj, algoritmul propune un compromis: vom folosi metodele eficiente de estimare a dimensiunii doar după ce dimensiunea fragmentului depășește $\frac{N}{\log_2 N}$, unde N este numărul de noduri al grafului.

Așadar, pentru implementare avem nevoie să cunoaștem numărul exact de noduri al unui fragment și ne putem folosi de modalitatea de numărare descrisă în algoritmul CT prin variabila *size_count*. Când $size\_count \geq \frac{N}{\log_2 N}$ algoritmului GHS i se adaugă noi modalități de actualizare a nivelului. Algoritmul Awebuch mentionează existenta unui singur nod rădăcină, așa că vom folosi SNR-GHS, fiind și mai ușor de manipulat. În continuare, când spunem GHS ne vom referi la SNR-GHS.

Root-Update

Această metodă crește nivelul rădăcinii dacă fragmentul nu a găsit MWOE-ul într-un anumit interval de timp.

Această procedură este activată atunci când:

1. *Initiate*: mesajul *Initiate* a înaintat într-un fragment F de dimensiune L pe o distanța mai mare de 2^{L+1}.

2. Muchii interne: un nod a detectat mai mult de 2^{L+1} muchii interne atunci când căuta MWOE-ul.

Procedura Root-Update implică oprirea căutării MWOE-ului, incrementarea nivelului, transmiterea valorii noului nivel în arbore și începerea unui nou proces de căutare a MWOE-ului.

Initiate Pentru implementarea primului caz al activării procedurii Root-Update vom adaugă doi parametri mesajului *Initiate*:

- *passed*: transmis de la rădăcina fragmentului către toate nodurile, este *True* dacă fragmentul are o dimensiune care depășește $\frac{N}{\log_2 N}$.
- H: inițializat de către rădăcină cu valoarea 2^{L+1}, este decrementat de către fiecare nod.

Atunci când un nod primește un mesaj *Initiate* cu H având valoarea 0 acesta trimite către rădăcină un mesaj *expInitiate* care să înștiințeze rădăcina că *Initiate*-ul a expirat (Codul sursă 4.12).

Codul sursă 4.13 prezintă procesarea mesajului *expInitiate*. Prin natura mesajului *Initiate*, toți fiii nodului central de pe nivelul 2^{L+1} vor trimite un mesaj *expInitiate*. Pentru a nu inunda nodul central cu mesaje *expInitiate*, un nod va trimite mai departe mesajul *expInitiate* doar dacă se află în starea *Find* (linia 2). După ce trimite mai departe mesajul, nodul trece în starea *Found*. Nodul central reacționează prin incrementarea nivelului și repornirea procedurii de căutare a MWOE-ului (liniile 4-8), în timp ce restul nodurilor doar trimit mai departe mesajul (linia 10).

```
1  if H == 0:
2      send(("expInitiate", L, index), to =
       ↪ branches[source_index]["process_id"]
3          return
```

Cod sursă 4.12: Condiția de expirare a mesajului *Initiate*

Iar răspunsul la mesajul *expInitiate* este:

```
1  def receive(msg = ("expInitiate", L, source_index)):
2      if L == LN and SN == "Find":
3          SN = "Found"
4          if i_am_root == True:
5              LN += 1
6              HN = 2**(LN +1)
7              SN = "Find"
8              Initiate()
9          else:
10             send(("expInitiate", L, index), to =
                ↪ branches[in_branch]["process_id"])
```

Cod sursă 4.13: Procesarea mesajului *expInitiate*

În realitate, chiar și pe un graf cu 100 de noduri, situația în care $H == 0$ este destul de rară. Însă, pentru a testa viabilitatea metodei $Root - Update$, vom seta valoarea $passed\_threshold = True$ de la începutul algoritmului și vom modifica această condiție în $H \leq 5$. Astfel, atunci când un mesaj *Initiate* este transmis cu $H \leq 5$, mesajul *expInitiate* va fi transmis înapoi către rădăcină, care va incrementa nivelul. Acest proces se repetă până când $2^{L+1} > 5$.

După aceste două modificări, observăm că programul intră într-o buclă infinită în majoritatea rulărilor. Aici intervine utilitatea redundanței conservative în design-ul algoritmilor distribuiți. Aceste retrimiteri succesive ale mesajului *Initiate* lasă anumite "bucle" deschise. Vom explora cauzele acestor bucle infinite și posibile rezolvări.

Mesaje *Report* expirate Există mesaje *Report* trimise atunci când un fragment avea nivelul L_1 care ajung să fie procesate din coadă abia atunci când fragmentul are nivelul L_2. Evident, aceste raportări nu mai sunt de actualitate și trebuie ignorate. Vom procesa doar *Report*-urile al căror nivel este același cu cel curent (Codul sursă 4.14).

Astfel, în răspunsul la un mesaj Report, adăugăm condiția:

```
1 def receive(msg = ("Report", w, count, L, source_index)):
2     if L != LN:
3         return
```

Cod sursă 4.14: Condiția adaugată la raspunsul pentru mesajul *Report*

Mesaje *Accept* de la un fragment mai mic Știm că un nod dintr-un fragment de nivel L_1 va răspunde cu *Accept* unui mesaj $Test$ din partea unui nod de nivel L_2 doar dacă $L_2 \leq L_1$. Așadar, dacă nodul de nivel L_1 primește un *Accept* din partea unui nod de nivel mai mic, acest *Accept* trebuie ignorat (Codul sursă 4.15). Acest lucru se poate întâmpla din cauza creșterii nivelului lui L_1 ulterior trimiterii unui mesaj $Test$.

În răspunsul la un mesaj *Accept* adăugăm condiția:

```
1 def receive(msg = ("Accept", L, source_index)):
2     if L < LN:
3         return
```

Cod sursă 4.15: Condiția adaugată la raspunsul pentru mesajul *Accept* pentru nivel

Mesaje repetate *Accept* Se poate întâmpla ca, din cauza trimiterii unor mesaje *Initiate* succesive, un nod de nivel L_1 să primească mesajul *Accept* din partea unui nod de nivel $L_2 \geq L_1$ de mai multe ori. Așadar, dacă muchia pe care a fost trimis mesajul *Accept* este deja *Branch*, mesajul trebuie ignorat (Codul sursa 4.16).

```
def receive(msg = ("Accept", L, source_index)):
    if branches[source_index]["state"] == "Branch":
        return
```

Cod sursă 4.16: Condiția adăugată la răspunsul pentru mesajul *Accept* pentru starea muchiei

Adăugarea acestei redundanțe asigură stabilitatea algoritmului chiar și în cazul extrem în care $passed\_threshold = True$ de la început și condiția de expirare a mesajului *Initiate* este $H \leq 5$. Acum că am validat acest caz extrem, readucem $passed\_threshold$ și H la valorile lor inițiale.

Muchii interne Cel de-al doilea caz al activării Root-Update (ilustrat în Codul sursă 4.17) este foarte similar cu mecanismul descris în algoritmul CT, cu diferența că orice nod din graf, nu doar rădăcina, poate activa procedura (liniile 5-7) atunci când observă că dimensiunea sa este mai mare ca 2^{L+1}.

```
def receive(msg = ("Report", w, count, source_index)):
    if L == LN:
        find_count +=1
        size_count += count
        if size_count >= 2**(LN + 1):
            # send expInitiate on the in_branch
            return
        if w < best_wt:
            best_wt = w
            best_edge = source_index
        report()
```

Cod sursă 4.17: Răspunsul la mesajul *Report* cu adăugarea activării Root-Update

Test-Distance

Procedura Test-Distance tratează cazul în care un fragment este absorbit de un fragment mai mare însă trebuie să aștepte pentru a deveni o parte activă a fragmentului mare. Pentru a eficientiza acest timp de așteptare, fragmentul iterează distanța până la noua rădăcină și își incrementează nivelul dacă distanța este prea mare.

Fie fragmentul F_1 de nivel L_1, cu $v \in F_1$ și fragment F_2 de nivel L_2, cu $w \in F_2$. Știm că $L_1 < L_2$ și că muchia (v, w) conectează cele două fragmente. Fragmentul F_1 decide că muchia (v, w) este MWOE și începe procedura *ChangeRoot* pe această muchie. Din punctul de vedere al lui F_1, care nu cunoaște valoarea lui L_2 în timp real, există 3 cazuri posibile:

1. $L_1 < L_2$, caz în care F_1 trimite un mesaj *Connect* către w, iar w este cel care decide că F_1 trebuie absorbit.

2. $L_1 = L_2$, caz în care și v și w vor trimite mesaje *Connect* de-a lungul muchiei (v, w), iar cel mai mare dintre ele va trimite și un mesaj *Initiate*.

3. $L_1 > L_2$, caz în care v va trimite un mesaj *Connect* și va aștepta procesarea mesajului de către w.

Algoritmul Awerbuch descrie folosirea procedurii Test-Distance doar în cazul $L_1 < L_2$. Procedura Test-Distance funcționează astfel:

- Nodul v trimite un mesaj *Exploration* către părintele său w. Acest mesaj conține un contor inițializat cu valoarea 2^{L_1+1}.

- Atunci când un nod primește mesajul *Exploration* scade din valoarea lui numărul său de muchii de tip *Branch*.

  - Dacă contorul rămâne pozitiv, nodul trimite mai departe mesajul, cu valoarea alterată a contorului, către părintele său.

  - Dacă contorul devine negativ înainte de a ajunge la rădăcina fragmentului, se trimite un mesaj *Ack* înapoi către nodul v. Atunci când *Ack* ajunge la v, nodul v transmite un mesaj în F_1 care determină fiecare nod să își crească nivelul cu 1. După trimiterea acestui mesaj, spunem că procedura Test-Distance a fost un succes iar nodul v o restartează. Procesul continuă până când procedura eșuează.

  - Dacă contorul ajunge la rădăcină și este pozitiv spunem că procedura a eșuat.

Nu este clar însă în ce moment nodul v începe procedura Test-Distance, deoarece doar nodul w decide că F_1 va fi absorbit. v este notificat doar atunci când primește mesajul *Initiate*.

4.3.3 Algoritmul Faloutsos-Molle

În 2004 a apărut lucrarea [11] care explorează problemele și inconsistențele din algoritmul lui Awerbuch și încearcă să le corecteze pentru a menține complexitatea inițială promovată de algoritmul lui Awerbuch.

Algoritmul Faloutsos-Molle introduce o serie de modificări:

Limitarea numărului de muchii *Rejected*

Algoritmul Faloutsos-Molle adaugă o nouă metodă de incrementare a nivelului unui fragment, prin analizarea numărului de muchii *Rejected*. Astfel, în timpul testării, fiecare

nod reține numărul de muchii către care a trimis un mesaj *Reject*. Dacă acest număr ajunge la valoarea 2^{L+1} se activează procedura Root-Update, așa cum ilustrează Codul sursă 4.18.

Pentru implementare, fiecare nod va reține o valoare *nr_rejected* inițializată cu 0 la *Initiate*. De fiecare dată când un nod marchează o muchie ca *Rejcted nr_rejected* este incrementat.

```
1 if nr_rejected >= 2**(LN+1):
2     #trimite expInitiate către rădăcină
```

Cod sursă 4.18: Activarea procedurii Root-Update prin limitarea numărului de muchii *Rejected*

Find, Found-Undecided și Found-Notified

Algoritmul face distincția între două tipuri de stări $Found$, similar cu algoritmul CT. Un nod se află în starea $FoundUndecided$ după ce a trimis mesajul *Report* și trece în starea $FoundDecided$ atunci când primește notificarea de la rădăcină că MWOE-ul a fost găsit ($MWOEfound$ sau $ChangeRoot$).

Un nod aflat în starea $FoundUndecided$ amână răspunsul la orice mesaj $Connect$ primit dacă nivelul mesajului este $\geq LN$.

Un nod aflat în starea $FoundNotified$ răspunde imediat la orice mesaj $Connect$ de același nivel.

Mesajul $MWOEfound$

Mesajul $MWOEfound$ are două roluri:

- Este transmis de rădăcină pentru a înștiința toate nodurile fragmentului că MWOE-ul a fost găsit.
- Este folosit în procedura Test-Distance pentru a anunța incrementarea nivelului în subfragmentul F_1.

Atunci când rădăcină începe procedura Change-Root trimite și mesajul $MWOEfound$ către toate nodurile din fragmentul său.

Procedura de fuzionare a fragmentelor se modifică astfel încât, atunci când un nod v trimite mesajul $Connect$ în procedura Change-Root către un nod w, avem două cazuri posibile:

1. $L_w > L_v$, caz în care w răspunde prin mesajul *acceptSub*. Dacă nodul w se află în starea $FoundUndecided$ își va aminti muchia wv și va trimite încă un *acceptSub* atunci când intră în starea $FoundNotified$.

2. $L_w = L_v$ nodul w așteaptă până intra în starea $FoundNotified$ pentru a răspunde nodului v.

Un prim pas în implementarea algoritmului este adăugarea mesajului $MWOEfound$.

```
msg = ("MWOEfound", souce_index)
```

Atunci când orice nod trimite mesajul $ChangeRoot$, propagă și mesajul $MWOEfound$ de-a lungul a toate muchiile sale de tip $Branch$. Atunci când un nod primește mesajul $MWOEfound$ modifică starea nodului în $FoundNotified$ și propagă mai departe mesajul.

Mesajul $AcceptSubmission$ Următorul pas în implementare este adăugarea mesajului $AcceptSubmission$. Este trimis de către un fragment mai mare în starea $FoundNotified$ sau $FoundUndecided$ către un fragment mai mic atunci când acesta îi acceptă absorbția. Acesta va fi folosit pentru a porni procedura Test-Distance.

Atunci când un nod v din fragmentul F_1 trimite mesajul $Connect$ către un nod w din fragmentul F_2, cu $L_2 > L_1$, fragmentul F_2 dorește să absoarbă fragmentul F_1. În algoritmul GHS, răspunsul era un mesaj $Initiate$. Dacă însă starea S_2 a fragmentului F_2 este $FoundUndecided$ sau $FoundNotified$, fragmentul F_1 nu va putea participa la căutarea MWOE-ului. În noul algoritm, răspunsul în acest caz este $AcceptSubmission$. Când nodul v primește $AcceptSubmission$ din partea unui fragment aflat în starea $FoundNotified$, acesta pornește procedura Test-Distance.

Dacă fragmentul F_2 se află în starea $FoundUndecided$, va marca acea muchie pentru retrimiterea mesajului atunci când starea se schimbă.

Astfel, în cele două cazuri în care un nod își modifică starea în $FoundNotified$ ($ChangeRoot$ și primirea mesajului MWOEfound), fiecare nod iterează asupra muchiilor sale și retrimite $AcceptSubmission$ muchiilor marcate (Codul sursă 4.19).

```
1 def change_root():
2     SN = "Found-Notified"
3     for (iterator, branch) în branches.items():
4         if branch["resendAcceptSubmission"] == True:
5             # trimite mesajul AcceptSubmission pe branch
```

Cod sursă 4.19: Retrimiterea mesajului $AcceptSubmission$

Adăugarea mesajului $AcceptSubmission$ duce la rularea infinită a programului prin faptul că modifică modalitatea de fuzionare a fragmentelor. Chiar și în starea $FoundNotified$ sau $FoundUndecided$, fragmentele mai mari absorbeau fragmentele mai mici. Inserarea unui interval de timp în care fragmentul mai mic este absorbit de fragmentul mai mare dar păstrează o identitate separată duce la o rulare infinită pe grafuri mari, când procesul de degugging este greu.

Pseudocodul algoritmului Faloutsos-Molle

Pe lângă descrierea amănunțită a algoritmului, lucrarea [11] include și un pseudocod. Acesta este însă incomplet, cu parametrii lipsă în funcții, proceduri declarate dar nu și apelate și pași lipsă. Am rescris acel pseudocod, folosind experiența trecută cu algoritmul și clarificând detaliile. Însă și această implementare duce la o rulare infinită.

Devine astfel evidentă importanța implementării algoritmilor distribuiți teoretici într-un mediu de simulare cum este DistAlgo. Lucrarea originală a lui Awerbuch [4] a reprezentat un rezultat important în studiul problemei MST. Neavând însă și o implementare practică, măcar simulată, a algoritmului, a fost mai apoi tratată de [11] pentru a clarifica unele puncte cheie omise. Chiar dacă a reprezentat un studiu remarcabil, nici lucrarea aceasta nu a putut însă aduce contribuții clare în privința practicalității: pseudocodul prezentat nu poate fi folosit dincolo de inspirație.

4.4 Compararea algoritmilor

4.4.1 Generarea grafurilor pentru testare

Pentru a testa cei doi algoritmi avem nevoie să generam grafuri conexe, neorientate, ponderate. Pentru simplitate, vom genera grafuri ponderate cu ponderi unice, astfel încât să nu fie nevoie să trecem grafurile prin transformarea descrisă în Afirmația 3.

La fel ca în cazul comparării algoritmilor pentru MIS, vom folosi *connected_watts_strogatz_graph* din pachetul NetworkX pentru a genera un graf conex aleatoriu de tip small world.

Cum nu există o astfel de metodă care să genereze un graf ponderat, vom asigna ponderi aleatorii uniforme muchiilor. Putem alege orice valori pentru ponderi, însă pentru simplitate vom alege valori aleatorii distincte din mulțimea $\{1, ..., m\}$, unde m este numărul de muchii al grafului.

4.4.2 Comparații

Compararea timpului total de rulare Am comparat performanța totală a algoritmilor pe 10 grafuri aleatorii de 500 de noduri. La fel ca în cazul comparației pentru problema MIS, am rulat fiecare algoritm de 3 ori și am folosit media acestor valori pentru graficul final, ilustrat în Figura 4.1.

Algoritmul GHS-SR este mai lent decât versiunea originala a GHS-ului, ceea ce explică performanța mai slabă a algoritmului Awerbuch comparativ cu GHS-ul original. Prin faptul că doar nodul central este cel ce trimite mesaje *Initiate*, GHS-SR induce o perioadă de așteptare pentru nodurile ce nu sunt centrale, scăzând performanța. Prin mecanismele

de actualizare mai rapidă a nivelului, algoritmul lui Awerbuch reușește să reducă acest efect, însă nu compensează pentru el.

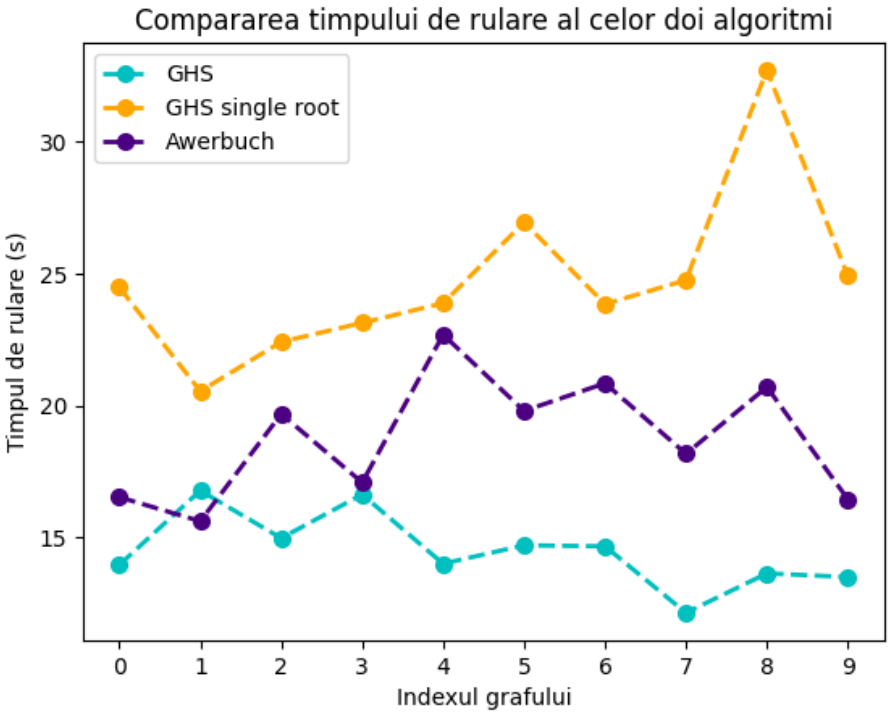

Figura 4.1: Compararea performanței algoritmilor GHS, GHS-SR și Awerbuch pentru rulările pe 10 grafuri aleatorii de 500 de noduri

Un amănunt ce trebuie luat în considerare atunci când comparăm performanța de timp a algoritmilor este variația timpului pentru același algoritm și același graf. În Figura 4.2 se observă că algoritmul GHS-SR și algoritmul lui Awerbuch au o variație foarte mare a performanței, între 16 și 22 de secunde. Algorimul GHS este mai stabil, având o variație a performanței între 13 și 15 secunde.

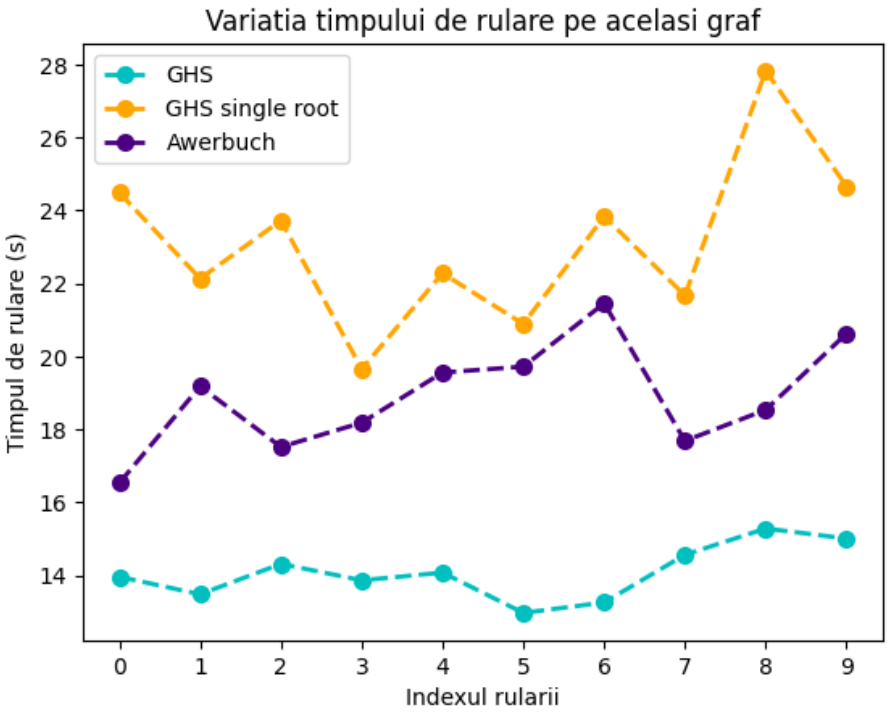

Figura 4.2: Variația timpului de rulare pe același graf de 500 de noduri pentru algoritmii GHS, GHS-SR și Awerbuch

Capitolul 5

Concluzii

Așadar, studiul unui algoritm în mediu distribuit simulat este extrem de util. Chiar și cu resurse limitate, o simulare oferă o oportunitate de studiu mult aprofundată față de introducerea teoretică a pseudocodului. Implementarea reală oferă o claritate superioară chiar și celor mai bune descrieri, iar multe dintre lucrările studiate de mine ([4], [11], [13]) ar fi beneficiat din includerea unei implementări, în orice limbaj de programare. Implementările în limbajul DistAlgo au beneficiul clarității și simplității, făcându-le ideale pentru expunerea într-o lucrare de cercetare.

În urma studiului meu, este surprinzătoare corectitudinea și claritatea obținută de lucrarea inițială pentru algoritmul GHS din 1983 [12], claritate ce nu a mai fost egalată de nicio altă rezolvare ulterioară a problemei MST.

De asemenea, implementarea algoritmilor într-un mediu simulat ne oferă o imagine asupra performanței. Comparația a scos la iveală o creștere a complexității pentru algoritmul GHS-SR care ar fi putut fi trecută cu vederea inițial. Însă performanța algoritmilor implementați este influențată de mulți factori, de la eficiența codului scris, al limbajului și chiar de performanța mașinii la momentul rulării. Astfel, comparațiile într-un mediu simulat pot fi utile, însă trebuie să fie însoțite de o analiză teoretică riguroasă.

5.1 Direcții viitoare

5.1.1 Vizualizarea grafică a evoluției algoritmilor

Pentru o mai buna analiză a algoritmilor distribuiți este utilă o metodă de vizualizare a evoluției lor. Putem folosi o librărie de vizualizare a grafurilor pentru a afișa în timp real și intuitiv acest proces.

5.1.2 Finalizarea implementării algoritmului lui Awerbuch

Procedura Test-Update nu este completă în implementarea mea deoarece induce o rulare infinită. Motivul nu este evident, însă îmi doresc să identific problema și să completez implementarea procedurii. Aceasta ar fi prima implementare a algoritmului lui Awerbuch de la publicarea lui, din cunoștințele mele.

5.1.3 Event-B

Event-B este un sistem de modelare și analiză a metodelor formale. [27] În 2020 Alexis Grall a publicat lucrarea Automatic Generation of DistAlgo Programs from Event-B Models [15] care prezintă o modalitate de a transforma modelele din Event-B în programe DistAlgo pentru a fi simulate în context distribuit. Plugin-ul de Rodin (IDE-ul pentru modele Event-B) care efectuează această transformare este disponibil online. Un avantaj propus de Event-B este caracterul formal al modelelor, ce poate asigura o fidelitate mai mare a implementării algoritmilor teoretici. Ar fi interesantă o analiză a acestor transformări și o comparație a implementărilor prezentate în lucrarea mea cu transformările modelelor din Event-B.

Bibliografie

[1] Wahabou Abdou, Nesrine Ouled Abdallah și Mohamed Mosbah, „Visidia: A java framework for designing, simulating, and visualizing distributed algorithms”, în *2014 IEEE/ACM 18th International Symposium on Distributed Simulation and Real Time Applications*, IEEE, 2014, pp. 43–46.

[2] Ioannis P. Androulakis, *Asynchronous Distributed Optimization Algorithms*, 2001, DOI: 10.1007/0-306-48332-7_14, URL: http://dx.doi.org/10.1007/0-306-48332-7_14.

[3] Damir Arbula și Kristijan Lenac, „Pymote: High Level Python Library for Event-Based Simulation and Evaluation of Distributed Algorithms.”, în *International Journal of Distributed Sensor Networks* (2013).

[4] Baruch Awerbuch, „Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems”, în *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 1987, pp. 230–240.

[5] Ioannis Chatzigiannakis, Athanasios Kinalis, Athanasios Poulakidas, Grigorios Prasinos și Christos Zaroliagis, „DAP: A generic platform for the simulation of distributed algorithms”, în *37th Annual Simulation Symposium, 2004. Proceedings.* IEEE, 2004, pp. 167–177.

[6] Francis Chin și HF Ting, „An almost linear time and O (nlogn+ e) messages distributed algorithm for minimum-weight spanning trees”, în *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, IEEE, 1985, pp. 257–266.

[7] Wikipedia contributors, *Monte Carlo algorithm*, Mar. 2023, URL: https://en.wikipedia.org/wiki/Monte_Carlo_algorithm.

[8] Wikipedia contributors, „Six degrees of separation”, în *Wikipedia* (Apr. 2023), URL: https://en.wikipedia.org/wiki/Six_degrees_of_separation.

[9] Zuleyha AKUSTA DAGDEVIREN, „A Minimum Spanning Tree based Clustering Algorithm for Cloud based Large Scale Sensor Networks”, în *Avrupa Bilim ve Teknoloji Dergisi* 26 (2021), pp. 415–420.

[10] Duncan Eddy și Mykel J Kochenderfer, „A maximum independent set method for scheduling earth-observing satellite constellations", în *Journal of Spacecraft and Rockets* 58.5 (2021), pp. 1416–1429.

[11] Michalis Faloutsos și Mart Molle, „A linear-time optimal-message distributed algorithm for minimum spanning trees.", în *Distributed Computing* 17.2 (2004).

[12] Robert G. Gallager, Pierre A. Humblet și Philip M. Spira, „A distributed algorithm for minimum-weight spanning trees", în *ACM Transactions on Programming Languages and systems (TOPLAS)* 5.1 (1983), pp. 66–77.

[13] Mohsen Ghaffari, „An improved distributed algorithm for maximal independent set", în *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, SIAM, 2016, pp. 270–277.

[14] Mohsen Ghaffari, „Improved distributed algorithms for fundamental graph problems", Teză de doct., Massachusetts Institute of Technology, 2017.

[15] Alexis Grall, „Automatic generation of DistAlgo programs from Event-B models", în *Rigorous State-Based Methods: 7th International Conference, ABZ 2020, Ulm, Germany, May 27–29, 2020, Proceedings 7*, Springer, 2020, pp. 414–417.

[16] Mohammad Shafkat Islam, *Luby's MIS Algorithm Lecture Notes*, [Online; accessed 19-April-2023], 2019, URL: https://homepage.cs.uiowa.edu/~sriram/5350/fall19/notes/10.1/10.1.pdf.

[17] Yanhong A. Liu, Scott D. Stoller și Bo Lin, „From Clarity to Efficiency for Distributed Algorithms", în *ACM Trans. Program. Lang. Syst.* 39.3 (Mai 2017), ISSN: 0164-0925, DOI: 10.1145/2994595, URL: https://doi.org/10.1145/2994595.

[18] M Luby, „A Simple Parallel Algorithm for the Maximal Independent Set Problem", în *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, Providence, Rhode Island, USA: Association for Computing Machinery, 1985, pp. 1–10, ISBN: 0897911512, DOI: 10.1145/22145.22146, URL: https://doi.org/10.1145/22145.22146.

[19] Jeff Magee, „Analyzing Synchronous Distributed Algorithms", în ().

[20] *networkx.org*, [Online; accessed 15-June-2023], URL: https://networkx.org/.

[21] Tim Nieberg, *Independent and dominating sets in wireless communication graphs.* University of Twente, Enschede, Netherlands, 2006.

[22] Gethin Norman, „Analysing randomized distributed algorithms", în *Validation of Stochastic Systems: A Guide to Current Research* (2004), pp. 384–418.

[23] Rainer Oechsle și Tim Gottwald, „DisASTer (distributed algorithms simulation terrain) a platform for the implementation of distributed algorithms", în *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, 2005, pp. 44–48.

[24] Gopal Pandurangan, „Distributed network algorithms", în *Book. url: https://sites. google. com/site/gopalpandurangan/dnabook. pdf (visited on 10/01/2019)* (2018).

[25] Wolfgang Schreiner, *DAJ–A Toolkit for the Simulation of Distributed Algorithms in Java*, rap. teh., Citeseer, 1997.

[26] Pradeep Kumar Sinha și Sunil R Dhore, „Multi-agent optimized load balancing using spanning tree for mobile services", în *International Journal of Computer Applications* 1.6 (2010), pp. 35–42.

[27] Dependable Systems și Software Engineering Research Group, *B.org*, [Online; accessed 15-June-2023], URL: http://www.event-b.org/.

[28] Peter Urban, Xavier Défago și André Schiper, „Neko: A single environment to simulate and prototype distributed algorithms", în *Proceedings 15th International Conference on Information Networking*, IEEE, 2001, pp. 503–511.

[29] Contributors to Wikimedia projects, *Conexitate (teoria grafurilor)*, Nov. 2022, URL: https://ro.wikipedia.org/wiki/Conexitate_(teoria_grafurilor).

[30] Wikipedia, *Arbore minim de acoperire — Wikipedia, The Free Encyclopedia*, http://ro.wikipedia.org/w/index.php?title=Arbore%20minim%20de%20acoperire&oldid=15308729, [Online; accessed 19-April-2023], 2023.

[31] Wikipedia, *Distributed algorithm — Wikipedia, The Free Encyclopedia*, http://en.wikipedia.org/w/index.php?title=Distributed%20algorithm&oldid=1137109418, [Online; accessed 30-May-2023], 2023.

[32] Wikipedia contributors, *Independent set (graph theory) — Wikipedia, The Free Encyclopedia*, [Online; accessed 15-June-2022], 2022, URL: https://en.wikipedia.org/w/index.php?title=Independent_set_(graph_theory)&oldid=1082877025.

[33] Wikipedia contributors, *Maximal independent set — Wikipedia, The Free Encyclopedia*, [Online; accessed 15-June-2022], 2022, URL: https://en.wikipedia.org/w/index.php?title=Maximal_independent_set&oldid=1072268765.

[34] Zhiyong Yu, Da Huang, Haijun Jiang, Cheng Hu și Wenwu Yu, „Distributed consensus for multiagent systems via directed spanning tree based adaptive control", în *SIAM Journal on Control and Optimization* 56.3 (2018), pp. 2189–2217.

[35] Jian Zhou, Lusheng Wang, Weidong Wang și Qingfeng Zhou, „Efficient graph-based resource allocation scheme using maximal independent set for randomly-deployed small star networks", în *Sensors* 17.11 (2017), p. 2553.