

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII**

**AL REPUBLICII MOLDOVA**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică şi Microelectronică**

**Departamentul Informatică şi Ingineria Sistemelor**

**Teza de An**

***la cursul de “Structuri de Date si Algoritmi”***

***Tema:”Algoritmul Ford-Fulkersson”***

Elaborat:

**Chistol Maxim, gr. IA-233**

Verificat:

**Maria Guțu,** *Lector Universitar.*

Departamentul Informatică şi IS,

Facultatea FCIM, UTM

**Chișinău 2024**

**Cuprins:**

[**Întroducere:** 3](#_Toc166302674)

[**1.** **Abordari Teoretice Algoritmul Ford-Fulkersson** 4](#_Toc166302675)

[**1.1** **Scurt Istoric** 4](#_Toc166302676)

[**1.2** **Teoria Fundamentala Algoritmului Ford-Fulkersson** 5](#_Toc166302677)

[**1.3 Algoritmul Ford-Fulkerson determina fluxul** 6](#_Toc166302678)

[**1.3** **PseudoCodul Algoritmului Ford-Fulkersson** 9](#_Toc166302679)

[**2. Abordari Practice Algoritmul Ford-Fulkersson** 10](#_Toc166302680)

[**2.1 Algoritmul Ford-Fulkersson** 10](#_Toc166302681)

[**2.2 Implementarea Algoritmului Ford-Fulkersson în limbajul de programare C** 11](#_Toc166302682)

[**2.2.1 Implementarea funcției BFS in Limbajul C** 12](#_Toc166302683)

[**2.3 Algoritmului Ford Fulkerson in problema fluxului maxim** 13](#_Toc166302684)

[2.3.1 **Probleme de optimizare** 13](#_Toc166302685)

[**Concluzie** 29](#_Toc166302686)

[**Bibliografie:** 30](#_Toc166302687)

[**Anexa:** 31](#_Toc166302688)

**Întroducere:**

În domeniul informaticii, structurile de date și algoritmii reprezintă fundația esențială pentru rezolvarea eficientă a problemelor complexe. În cadrul acestei lucrări, ne vom concentra asupra unuia dintre cei mai importanți algoritmi utilizați în teoria grafurilor și în rețelele de transport: algoritmul Ford-Fulkerson.

Algoritmul Ford-Fulkerson este o metodă puternică și versatilă utilizată pentru a găsi fluxul maxim într-o rețea direcționată, fiind fundamental în rezolvarea problemelor de flux maxim, cum ar fi rețelele de transport și alte aplicații practice. În această lucrare, ne propunem să explorăm în profunzime etapele acestui algoritm, modalitățile de parcurgere a rețelei și să analizăm pseudo-codul acestuia pentru a înțelege cum funcționează și cum poate fi implementat în practică.

Pentru a înțelege în profunzime algoritmul Ford-Fulkerson, vom începe prin prezentarea conceptelor teoretice fundamentale legate de fluxurile în rețele și capacitățile arcelor. Vom explora, de asemenea, conceptele de flux residual și tăieturi minime, care stau la baza funcționării acestui algoritm.

În continuare, lucrarea noastră va include și o componentă practică, în care vom implementa algoritmul Ford-Fulkerson în cod. Vom începe cu un exemplu simplu, în care vom explica pas cu pas cum se construiește acest algoritm într-un limbaj de programare specific. Apoi, vom prezenta un cod mai complex, în care algoritmul Ford-Fulkerson va fi aplicat într-un context practic, oferind astfel o înțelegere mai profundă a modului în care poate fi utilizat în aplicații reale.

Un aspect esențial al acestei lucrări îl reprezintă prezentarea pseudo-codului algoritmului Ford-Fulkerson, care va ilustra clar modul de implementare a acestuia într-un mediu de programare. Vom discuta, de asemenea, complexitatea algoritmului și condițiile de terminare pentru asigurarea corectitudinii și eficienței acestuia.

Prin explorarea etapelor de rezolvare, parcurgerii și pseudo-codului acestui algoritm, lucrarea noastră își propune să ofere o înțelegere comprehensivă a importanței și funcționării algoritmului Ford-Fulkerson în rezolvarea problemelor de rețele și fluxuri, inclusiv o privire practică asupra aplicării sale în contexte reale.

# **Abordari Teoretice Algoritmul Ford-Fulkersson**

Algoritmul Ford-Fulkerson este o metodă clasică folosită pentru a găsi fluxul maxim într-o rețea. Acesta se bazează pe principiul augmentării fluxului într-o rețea de la sursă la destinație până când nu mai este posibilă o creștere a fluxului.

# **Scurt Istoric**

Algoritmul Ford-Fulkerson, denumit după matematicienii L.R. Ford, Jr. și D.R. Fulkerson, a fost publicat într-un articol științific în 1956. Dezvoltarea acestui algoritm a avut loc în contextul evoluției teoriei rețelelor și algoritmilor de optimizare combinatorială. Începând cu anii 1930 și 1940, matematicienii și cercetătorii au început să studieze problemele de flux în rețele, pentru a optimiza sistemele de transport și telecomunicații. Lucrările lui Ford și Fulkerson din anii 1950 au marcat un moment important, prezentând algoritmul pentru determinarea fluxului maxim într-o rețea și demonstrând eficiența sa în diverse aplicații practice. În anii 1960 și 1970, s-a continuat studiul algoritmului Ford-Fulkerson și s-au căutat modalități de îmbunătățire a eficienței și performanței acestuia. Au fost dezvoltate și alte tehnici, cum ar fi algoritmul Edmonds-Karp, o variantă optimizată a algoritmului Ford-Fulkerson. După anii 1970, interesul pentru teoria rețelelor și algoritmii de flux a crescut, odată cu avansul tehnologiei și apariția unor aplicații noi, cum ar fi rețelele de calculatoare și rețelele sociale. Algoritmul Ford-Fulkerson a rămas unul dintre algoritmii fundamentali în teoria rețelelor și algoritmii de optimizare, fiind aplicat în diverse domenii, de la inginerie și tehnologie informatică la economie și științe sociale.

# **Teoria Fundamentala Algoritmului Ford-Fulkersson**

Algoritmul Ford-Fulkerson este un algoritm fundamental în teoria grafurilor și în problemele de rețele. Este utilizat pentru a găsi fluxul maxim într-o rețea de fluxuri, adică cantitatea maximă de materiale sau informații care pot fi transmise de la sursă la destinație într-o rețea dată, respectând capacitățile arcelor.

Prezentare a algoritmului Ford-Fulkerson:

**1.** **Inițializare:** Se începe cu un flux nul și se alege un drum augmenting (un drum de la sursă la destinație în care se poate crește fluxul).

**2.** **Căutare drum augmenting:** Se folosește o metodă de căutare pentru a găsi un drum de la sursă la destinație în care se poate crește fluxul.

**3.** **Actualizare fluxului:** Se crește fluxul pe drumul augmenting găsit.

**4.** **Repetare:** Se repetă pașii 2 și 3 până când nu mai există drumuri augmenting în rețea.

**5.** **Terminare:** Se obține fluxul maxim.

Există mai multe metode de căutare a unui drum augmenting, cum ar fi căutarea în lățime (BFS) sau căutarea în adâncime (DFS). Odată ce un drum augmenting este găsit, se ajustează fluxul pe acea cale și se continuă căutarea până când nu mai există astfel de drumuri.

Un aspect important al algoritmului Ford-Fulkerson este că performanța sa poate varia în funcție de strategia de căutare a drumurilor augmenting și de modul în care sunt gestionate capacitățile reziduale ale arcelor. De asemenea, trebuie să fim atenți la posibile cicluri de flux negativ, care pot apărea în timpul procesului de căutare a drumurilor augmenting.

Algoritmul Ford-Fulkerson poate fi îmbunătățit și optimizat prin utilizarea unor tehnici precum algoritmul Edmonds-Karp, care folosește căutarea în lățime pentru a găsi un drum augmenting într-un mod mai eficient.

# **1.3 Algoritmul Ford-Fulkerson determina fluxul**

Algoritmul Ford-Fulkerson determină fluxul maxim care poate fi introdus într-o rețea de transport și se mai cunoaște și sub denumirea de problema Algoritmul Ford-Fulkerson "debitului maxim" (max flow).

Procedeul prin care se determină debitul maxim este relativ simplu:

- Fie costul inițial egal cu 0.

- Cat timp există un drum de la S la P, atunci fie D un drum de la S la P.

- Fie m muchia care este inclusă în D și are costul minim.

- Scade costul muchiei m din toate muchiile care aparțin lui D.

- Pentru fiecare muchie A->B inclusă în D adaugă o muchie B->A egală cu costul muchiei m.

- Adaugă la costul total valoarea costului muchiei m.

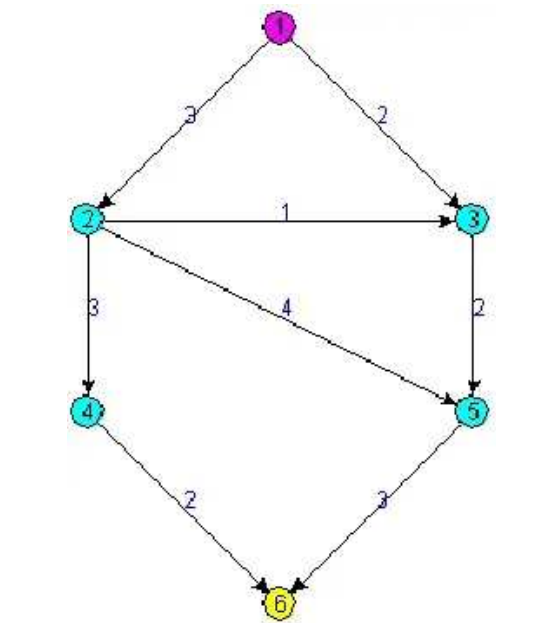
Fie f un flux introdus prin rețea la un moment dat. Se numește capacitatea reziduală a unei muchii în acel moment fluxul adițional care mai poate fi introdus prin acea muchie fără a depăși însă capacitatea acelei muchii. De exemplu, dacă capacitatea unei muchii (u,v) este C(u,v) = 10 iar prin acea muchie a fost introdus un flux F(u,v) = 7, fluxul F(u,v) = 7 atunci capacitatea reziduală a muchiei (u,v) este CF(u,v) = C(u,v) - F(u,v).

Când fluxul f(u,v) este negativ, capacitatea reziduală CF(u,v) este mai mare decât capacitatea C(u,v). De exemplu, dacă C(u,v) = 10 și f(u,v) = -4, atunci CF(u,v) = 14, ceea ce poate fi interpretat în felul următor: există un flux de 4 unități care merge de la v la u și poate fi anulat prin introducerea unui flux de 4 unități de la u la v. Apoi mai putem introduce un flux de încă 10 unități până la atingerea maximului pentru muchia respectivă. Deci, în total, putem introduce 14 unități pe flux pe o muchie de cost 10, începând de la un flux cu cost 10, fără a depăși capacitatea maximă de -4 a muchiei.

Fiind dată o rețea de transport G=(V,E), în care V este mulțimea vârfurilor (vertex) și E mulțimea muchiilor (edge), și un flux F, numim rețea reziduală a rețelei G indusă de F o rețea GF(V,Ef) în care EF={(u,v) ∈ VxV | CF(u,v) > 0}. De observat că (u,v) poate fi o muchie reziduală în EF chiar dacă nu este muchie în E. Rețeaua reziduală GF este și ea o rețea de transport cu capacitățile date de C rețea de transport cu capacitățile date de CF.

Principala problemă care apare este alegerea unui drum (augmenting path) de la S la P și în acest sens există mai multe metode de alegere a lui. Fie p un drum de la S la P. Se numește capacitate reziduală a lui p maximul de unități de flux pe care le putem introduce pe drumul p, adică CF(p) = min(CF(u,v) | (u,v) ∈ p}.

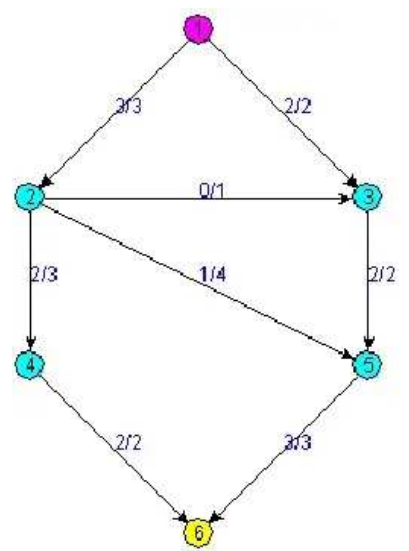
Să presupunem că avem rețeaua de transport din figura 1.3.1 și ne propunem să găsim fluxul maxim care poate traversa rețeaua de la nodul 1 la nodul 6.



**Fig.1.3.1**

Pentru început, stabilim fluxul maxim la valoarea 0 și alegem un drum de la 1 la 6. Să zicem că am găsit drumul 1-2-3-5-6 care are capacitatea reziduală 1 (datorită muchiei 2-3). Adăugăm 1 la rezultat și continuăm algoritmul. Alegem alt drum, de exemplu 1-2-4-6, care are capacitatea reziduală 2 (datorită muchiei 2-6), pe care o adăugăm la fluxul total și actualizăm costurile muchiilor.

În acest moment, nu mai există decât un singur drum de la 1 la 6, și anume drumul 1-3-5-6, care are capacitatea reziduală 1 (datorită muchiei 3-5, care avea inițial capacitatea 2, dar în urma alegerii drumului 1-2-3-5-6 a rămas doar cu o singură unitate de flux). Deci, am obținut un flux total de 4 unități, algoritmul luând sfârșit deoarece nu se mai poate ajunge, maxim deoarece după cum se vede în figura 1.3.2, în această rețea de transport se ajunge din nodul 1 în nodul 6. Dar acest rezultat nu este tocmai maxim, deoarece se poate obține un flux de 5 unități.

Așadar, pentru a obține fluxul maxim în această rețea de transport, este necesar să reevaluăm alegerile drumurilor și să continuăm algoritmul pentru a găsi un drum cu capacitatea reziduală mai mare sau egală cu 1.

**Fig.1.3.2:** Fluxul maxim prin reteaua de transport din figura 1.3.2

Pentru a vedea unde este problema sa presupunem ca in graful inițial in loc sa alegem drumul 1-2-3-5-6 am alege drumul 1-2-4-6 care are o capacitate reziduala de 2 unitați, iar apoi drumurile 1-2-5-6 si 1-3-5-6. Aceasta alegere a drumurilor ne conduce la un flux total de 5 unitati. Dar obținerea fluxului maxim n-ar trebui sa depinda de felul in care se aleg drumurile. Acesta este un exemplu de algoritm greedy care eșueaza.

Ca algoritmul sa functioneze corect introducem pentru fiecare muchie (u,v) cu fluxul f u,v  in rețeaua de transport o muchie (v,u) în rețeaua reziduală cu capacitatea F (u,v). Prin aceasta metoda îi permitem algoritmului sa faca “undo” la  propriile decizii trimitand înapoi un flux în direcția opusă. Așa dar in graful rezidual din exemplul nostru vor fi muchii în ambele direcii între vârfurile 2 si 3. Algoritmul va putea astfel, după gasirea drumului 1-2-4-6 sa gasească drumul 1-3-2-5-6 care poate fi impins înapoi pe muchia 2-3 algoritmul, gasind astfel înca un flux de cost 1 ca apoi alegand si drumul 1-3-5-7 (de capacitate reziduală 1) să ajunga la fluxul maxim egal cu 5.

# **PseudoCodul Algoritmului Ford-Fulkersson**

FORD-FULKERSON(Graf G(V,E), Nod S, Nod D)

maxflow ⇐ 0

pentru fiecare muchie (u,v) din G

f[u,v] ⇐ 0

f[v,u] ⇐ 0

atâta timp cât există un drum p de la S la D în rețeaua reziduală Gf

cp ⇐ min { cf(u,v) | (u,v) ∈ p }

maxflow ⇐ maxflow + cp

pentru fiecare muchie (u,v) din p

f[u,v] ⇐ f[u,v] + cp

f[v,u] ⇐ f[u,v]

Pseudocodul urmează pașii algoritmului Ford-Fulkerson:  
  
-Inițializează fluxul maxim maxflow la 0 și fluxul pe fiecare muchie la 0.  
-Caută un drum de la nodul de sursă S la nodul de destinație D în rețeaua reziduală Gf, unde Gf este rețeaua reziduală la pasul curent.  
-Pentru fiecare drum găsit, determină capacitatea reziduală minimă cp pe drum și actualizează fluxul maxim cu această valoare.  
-Actualizează fluxurile pe fiecare muchie de-a lungul drumului găsit.  
-Continuă acești pași până când nu mai există drumuri de la S la D în rețeaua reziduală.  
-Întoarce valoarea fluxului maxim calculat.

# **2. Abordari Practice Algoritmul Ford-Fulkersson**

# **2.1 Algoritmul Ford-Fulkersson**

FORD-FULKERSON(Graf G(V,E), Nod S, Nod D)

maxflow ⇐ 0

pentru fiecare muchie (u,v) din E[G] executa

f[u,v] ⇐ 0

f[v,u] ⇐ 0

atât timp cât există un drum p de la S la D în rețeaua reziduală Gf executa

cp ⇐ min { cf(u,v) | (u,v) ∈ p }

maxflow ⇐ maxflow + cp

pentru fiecare muchie (u,v) din p executa

f[u,v] ⇐ f[u,v] + cp

f[v,u] ⇐ f[u,v]

întoarce maxflow

1. Funcția Ford-Fulkersson primește graful care prezintă graful problemei și nodul inițial și nodul final
2. Inițializăm maxflow cu 0 asa cum inițial fluxul este 0
3. Egalam fluxul pe muchii cu zero
4. Se apelează funcția BFS( care calculează drumul in adâncime) se execute atâta timp cat existra drumuri de la S la D
5. După fiecare executare a funcției BFS capacității muchilor se aduna cp (Teta minimă)

# **2.2** **Implementarea Algoritmului Ford-Fulkersson în limbajul de programare C**

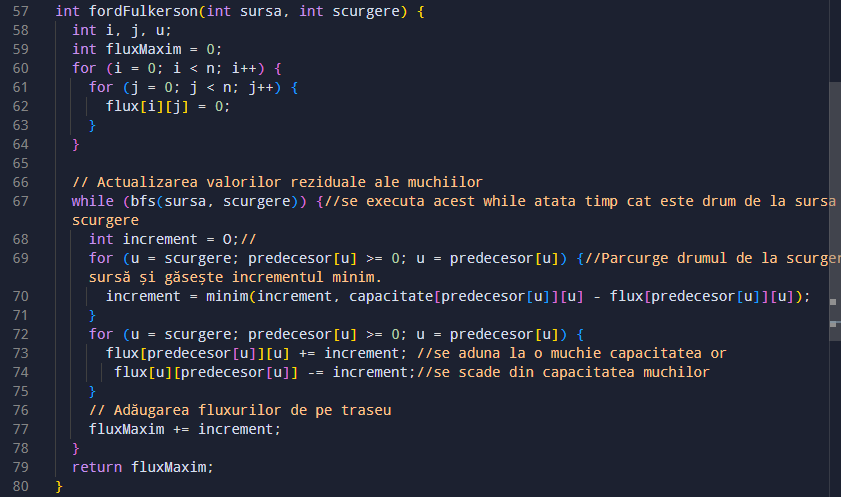


Fig. 2.2.1 Implementarea algoritmului Ford-Fulkersson în Limbajul C

1. Funcția Ford-Fulkerson este definită cu două variabile "sursa" și "scurgere" (int sursa, int scurgere). Aceste variabile reprezintă nodul inițial (nodul A) și nodul final (nodul B) în rețeaua de transport.

2. Prima buclă "for" și a doua buclă "for" inițializează matricea flux cu zero, adică fiecare element flux[i][j] este setat la 0.

3. Bucla "while" are ca și condiție funcția "bfs", care se execută atâta timp cât există un drum de la sursă la scurgere în rețeaua reziduală.

4. Incrementul este inițializat cu zero, deoarece va reprezenta capacitatea minimă a muchiilor pe drumul găsit.

5. Bucla "for" parcurge drumul găsit și determină incrementul minim, apoi îl atribuie variabilei "increment".

6. În urma parcurgerii buclei "for", incrementul este adunat la capacitatea muchiilor care alcătuiesc drumul și scăzut din capacitatea muchiilor invers orientate.

7. La finalul parcurgerii etapelor 4-6, fluxul maxim este actualizat prin adăugarea incrementului calculat, adică "fluxMaxim += increment".

# **2.2.1 Implementarea funcției BFS in Limbajul C**

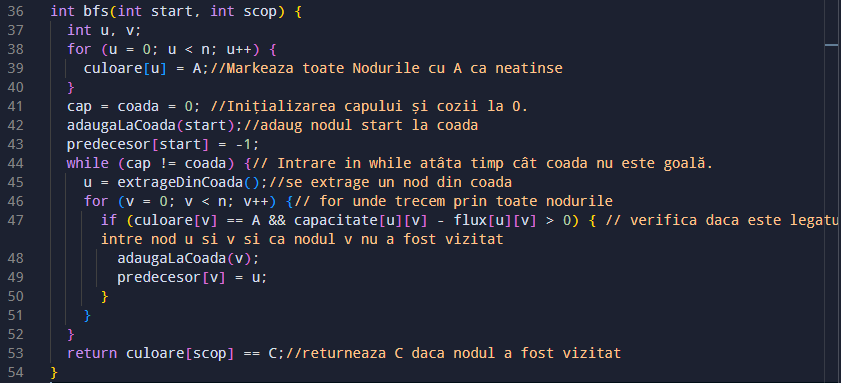


Fig.2.2.2: Implimentarea funcția BFS in Limbajul C

1. Funcția „bfs” este definită cu 2 variabile “start” si “scop”
2. Bucla „for” parcurcurge vectorul si notiaza toate nodurile cu „A” ca neatinse
3. Initializam capul si coada cu zero
4. Variabila „predecesor” se notiaza cu (-1) pentru a arata ca primul nod nu are predecesor
5. Parcurgerea buclei „while” care zice ca cat timp cap este deferit de coada se va parcurge urmatoarele instructiuni
6. Se extrage un nod din coada si va parcurge bucla „for” care trece prin toate nodurile si cauta drum.
7. Parcurge bucla „IF” care verifica daca au ligatura intre sine acese noduri si daca na fost vizitat nodul,urmaa carui il adauga in coda si noteaza predecesorul
8. Altfel va nota nodul cu „C” si il va returna.

# **2.3 Algoritmului Ford Fulkerson in problema fluxului maxim**

Implementarea codului în limbajul C, problema fluxului maxim cu ajutorul Algoritmului Ford-Fulkersson.

Această implimentarea va calcula fluxul maxim care trece printrun graf.Datele necesare vor fi preluate din fisierul “Input.txt” in care se află dimensiunile grafului si muchile de pornire si de final si capacitatea muchiei.După parcurgerea codului fluxul maxim va fi afișat la consolă și in fișier „output.txt”

2.3.1 **Probleme de optimizare**

Problemele de optimizare pot fi abordate și pentru algoritmului de Ford-Fulkersson, așa cum algoritmul Ford-Fulkerson este o metodă clasică pentru găsirea fluxului maxim într-o rețea. Există mai multe modalități de a optimiza acest algoritm și de a-l face mai eficient.

1. Selectarea unei metode de căutare a căii crescătoare: Algoritmul Ford-Fulkerson necesită găsirea unei căi crescătoare de la sursă la destinație în fiecare iterație. Alegerea unei metode eficiente pentru această căutare poate îmbunătăți semnificativ performanța algoritmului. O opțiune este să fie folosit căutarea în lățime (BFS) sau căutarea în adâncime (DFS). Ambele au avantaje și dezavantaje, iar alegerea depinde de structura specifică a rețelei.

2. Implementarea unei structuri de date eficiente: Folosirea unei structuri de date eficiente pentru a stoca grafurile și fluxurile poate reduce timpul de execuție al algoritmului. De exemplu, o listă de adiacență poate fi mai rapidă decât o matrice de adiacență în anumite cazuri.

3. Tăierea grafurilor: Identificarea și eliminarea muchiilor care nu pot face parte dintr-un flux maximal poate reduce numărul de iterații necesare pentru a găsi un flux maximal. Aceasta poate fi realizată prin aplicarea unui algoritm de tăiere a grafurilor, cum ar fi algoritmul Edmonds-Karp.

4. Îmbunătățirea scalabilității: În rețele mari, scalabilitatea poate deveni o problemă. Puteți explora tehnici de optimizare a memoriei sau de paralelizare a algoritmului pentru a îmbunătăți performanța acestuia în cazul unor rețele mai mari.

5. Optimizarea actualizării fluxului: În timpul actualizării fluxului în fiecare iterație, puteți căuta modalități de a optimiza acest proces. De exemplu, puteți folosi o coadă de priorități pentru a identifica cea mai mică capacitate residuală de pe calea crescătoare, reducând astfel timpul de căutare.

6. Paralelizarea algoritmului: În anumite cazuri, algoritmul Ford-Fulkerson poate fi paralelizat pentru a profita de arhitecturile hardware moderne, cum ar fi procesoarele multi-nucleu sau sistemele distribuite.

8. Explorarea altor algoritmi: În unele cazuri, algoritmul Ford-Fulkerson poate fi înlocuit cu algoritmi alternativi, cum ar fi algoritmul Edmonds-Karp sau algoritmul Dinic, care pot avea performanțe mai bune în anumite scenarii.

2.3.2 Implimentarea problemei fluxul maxim in limbaj C

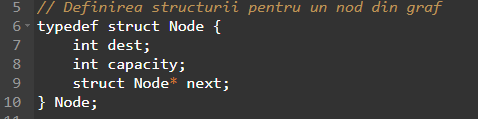


Fig.2.3.1:Definirea structurii pentru un nod din graf.

Această secțiune definește o structură numită “Node”, care va fi utilizată pentru a reprezenta un nod din graf. Structura conține următoarele câmpuri:  
1. “dest”: Un întreg care reprezintă destinația acestui nod în cadrul unei muchii.  
2. “capacity”: Un întreg care reprezintă capacitatea muchiei asociate acestui nod.  
3. “next”: Un pointer către următorul nod din listă. Acest lucru permite crearea unei liste de adiacență sau a unei liste de vecini pentru fiecare nod din graf.

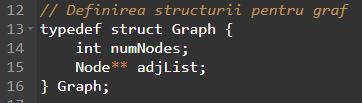


Fig.2.3.2:Definirea structurii pentru graf.

Această secțiune definește o structură numită Graph, care este utilizată pentru a reprezenta întregul graf. Structura conține următoarele câmpuri:

1. numNodes: Un întreg care reprezintă numărul de noduri din graf.  
2. adjList: Un pointer la un pointer de tip Node. Aceasta este o listă de pointeri către noduri (Node\*), care reprezintă lista de adiacență pentru fiecare nod din graf. Prin lista de adiacență, putem accesa vecinii fiecărui nod și capacitatea muchiilor care le leagă.

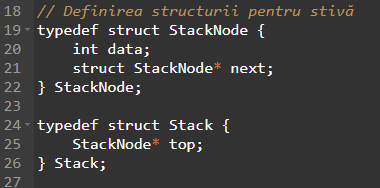
w

Fig.2.3.3:Definirea structurii pentru stivă.

Această secțiune definește două structuri:  
  
 1. StackNode: Aceasta reprezintă un nod în stivă. Fiecare nod conține un câmp data care reține informația stocată în nod și un pointer către următorul nod în stivă (next).  
2. Stack: Aceasta este structura stivei în sine. Aceasta conține un singur câmp, top, care este un pointer către vârful stivei. Prin intermediul acestui pointer, putem accesa vârful stivei și potențialele elemente de acolo

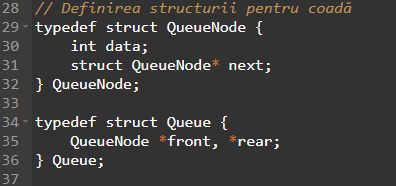


Fig.2.3.4

Această secțiune definește două structuri:  
1.QueueNode: Aceasta reprezintă un nod în coadă. Fiecare nod conține un câmp data care reține informația stocată în nod și un pointer către următorul nod din coadă (next).  
2.Queue: Aceasta este structura coadei în sine. Aceasta conține două câmpuri, front și rear, care sunt pointeri către începutul și sfârșitul coadei, respectiv. Prin intermediul acestor pointeri, putem accesa atât începutul, cât și sfârșitul coadei și potențialele elemente stocate acolo.

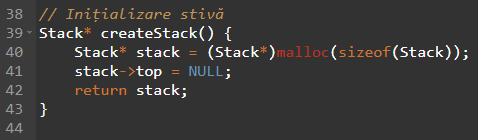


Fig.2.3.5

Această funcție createStack() returnează un pointer la o stivă nou creată și inițializată. Ea face următoarele lucruri:  
  
1.Alocă spațiu pentru o nouă structură Stack folosind funcția malloc(sizeof(Stack)). Aceasta asigură că avem suficient spațiu pentru a stoca o stivă.  
2.Inițializează câmpul top al stivei cu valoarea NULL. Acest lucru indică că stiva este inițial vidă.  
3.Returnează pointerul la stiva nou creată.

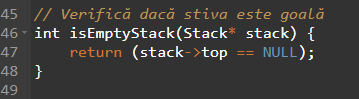


Fig.2.3.6

Această funcție isEmptyStack() primește ca argument un pointer către o stivă și returnează un întreg. Ea verifică dacă câmpul top al stivei este NULL. Dacă acest lucru este adevărat, înseamnă că stiva este goală și funcția returnează 1 (adevărat). În caz contrar, funcția returnează 0 (fals), indicând că stiva nu este goală.

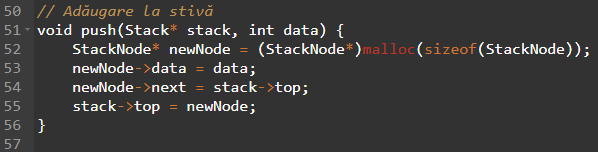


Fig.2.3.7

Alocă memorie pentru un nou nod în stivă folosind funcția malloc(sizeof(StackNode)). Acest nod va conține datele pe care dorim să le adăugăm la stivă.  
Setează câmpul data al noului nod cu valoarea data dată ca argument.  
Setează câmpul next al noului nod să pointeze către vârful stivei existente. Astfel, noul nod devine noul vârf al stivei, iar vechiul vârf devine următorul nod în stivă.  
Actualizează vârful stivei pentru a fi noul nod adăugat.

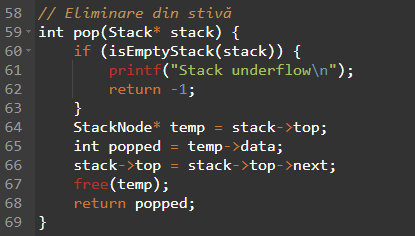


Fig.2.3.8

Această funcție primește ca argument un pointere către stivă

1.Verifică dacă stiva este goală utilizând funcția isEmptyStack(). Dacă stiva este goală, afișează un mesaj de eroare "Stack underflow" și returnează -1 pentru a indica o eroare.  
2.Altfel, salvează pointerul către vârful stivei într-o variabilă temporară temp.  
3.Extrage datele stocate în vârful stivei și le salvează într-o variabilă popped.  
4.Actualizează vârful stivei pentru a fi următorul nod în stivă.  
5.Eliberează memoria ocupată de nodul extras din stivă.  
6.Returnează datele extrase din vârful stivei.

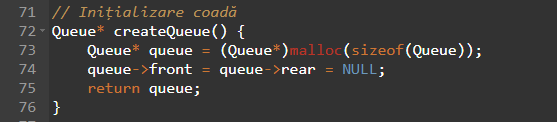


Fig.2.3.9

Alocă memorie pentru o nouă structură Queue folosind funcția malloc(sizeof(Queue)). Acest lucru asigură că avem suficient spațiu pentru a stoca o coadă.  
Inițializează câmpurile front și rear ale cozii cu valoarea NULL. Acest lucru indică că coada este inițial vidă.  
Returnează pointerul la coada nou creată.

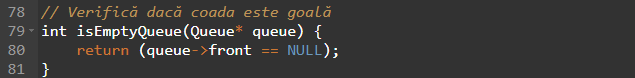


Fig.2.3.10

Această funcție isEmptyQueue() primește ca argument un pointer către o coadă și returnează un întreg.

1. Ea verifică dacă câmpul front al cozii este NULL. Dacă acest lucru este adevărat, înseamnă că coada este goală și funcția returnează 1 (adevărat). În caz contrar, funcția returnează 0 (fals), indicând că coada nu este goală.

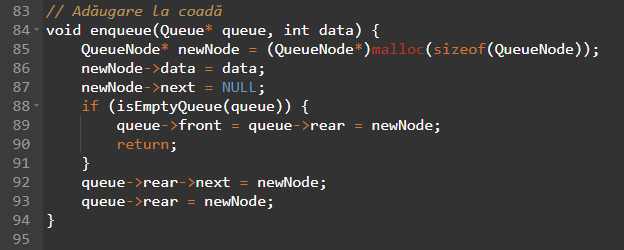


Fig.2.3.11

1.Alocă memorie pentru un nou nod în coadă folosind funcția malloc(sizeof(QueueNode)). 2.Acest nod va conține datele pe care dorim să le adăugăm la coadă.  
3.Inițializează câmpurile data și next ale noului nod cu valorile corespunzătoare.  
4.Verifică dacă coada este goală utilizând funcția isEmptyQueue(). Dacă coada este goală, atât front cât și rear sunt actualizate pentru a pointa către noul nod. În caz contrar, noul nod este adăugat la sfârșitul cozii, iar rear este actualizat pentru a pointa către noul nod.

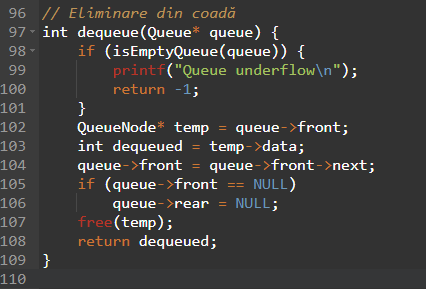


Fig.2.3.12

Alocă memorie pentru un nou nod în coadă folosind funcția malloc(sizeof(QueueNode)).

1.Acest nod va conține datele pe care dorim să le adăugăm la coadă.  
Inițializează câmpurile data și next ale noului nod cu valorile corespunzătoare.  
Verifică dacă coada este goală utilizând funcția isEmptyQueue(). Dacă coada este goală, atât front cât și rear sunt actualizate pentru a pointa către noul nod. În caz contrar, noul nod este adăugat la sfârșitul cozii, iar rear este actualizat pentru a pointa către noul nod.

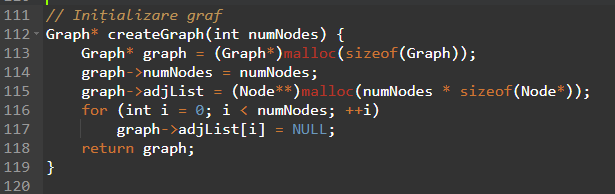


Fig.2.3.13

1. Alocă memorie pentru o nouă structură Graph folosind funcția malloc(sizeof(Graph)). 2 2. Aceasta asigură că avem suficient spațiu pentru a stoca un graf.  
   3. Initializează câmpul numNodes al grafului cu valoarea numNodes dată ca argument.  
   4.Alocă memorie pentru lista de adiacență (adjList) a grafului. Pentru fiecare nod din graf, se alocă memorie pentru un pointer către un nod (Node\*). Această listă este inițializată cu NULL pentru fiecare nod, indicând că inițial graful nu are muchii.  
   Returnează pointerul la graf nou creat.

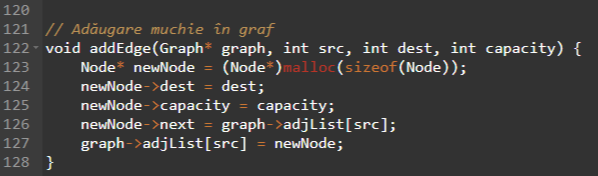


Fig.2.3.14

Această funcție primește ca argument graful,capacitatea src si destinați.

1.Alocă memorie pentru un nou nod (Node) care va reprezenta o nouă muchie în graf.  
Inițializează câmpurile dest și capacity ale noului nod cu valorile corespunzătoare destinației și capacitatei date ca argumente.  
2.Setează câmpul next al noului nod să pointeze către lista de adiacență a nodului sursă (src). Astfel, noul nod este adăugat la începutul listei de adiacență a nodului sursă.  
3.Actualizează lista de adiacență a nodului sursă (src) pentru a include noul nod adăugat.

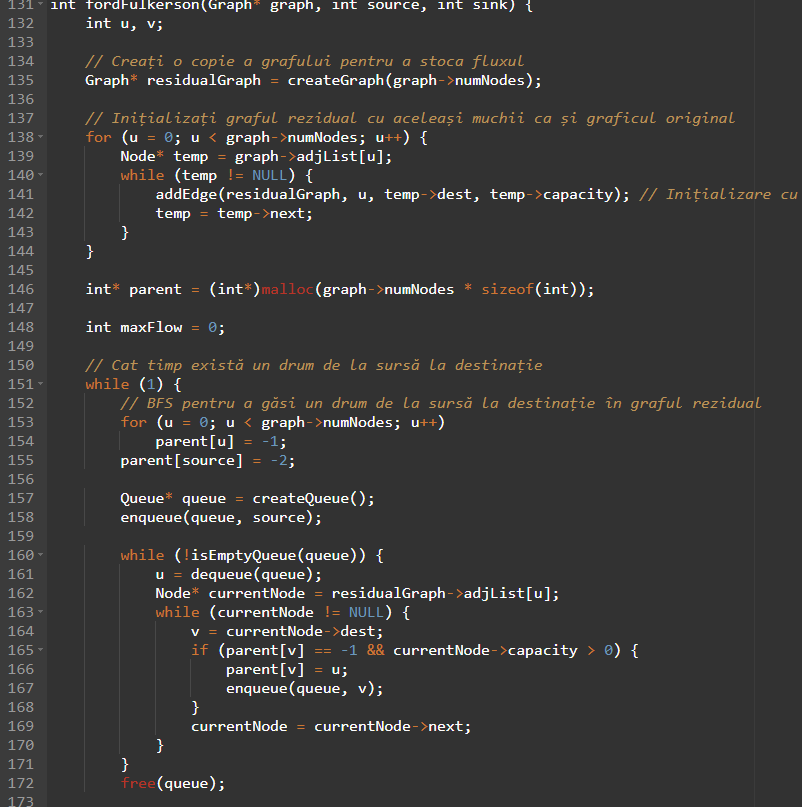


Fig.2.3.15

1.Creează o copie a grafului inițial pentru a stoca graful rezidual. Graful rezidual este utilizat pentru a găsi fluxul maxim prin graf.  
2.Inițializează graful rezidual cu aceleași muchii ca și graful original, dar cu capacitățile inițiale.  
3.Inițializează un tablou parent pentru a stoca drumul de la sursă la destinație găsit în BFS.  
4.Inițializează variabila maxFlow care va stoca fluxul maxim găsit.  
5.Aplică BFS (breadth-first search) pe graful rezidual pentru a găsi un drum de la sursă la destinație.  
6.După ce se termină BFS, se eliberează memoria utilizată pentru coadă (queue).

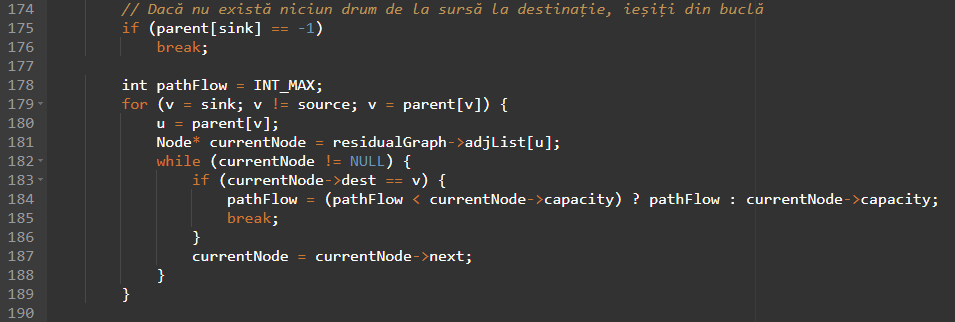


Fig.2.3.16

1.Verifică dacă s-a găsit vreun drum de la sursă la destinație în graful rezidual. Dacă nu există niciun drum, se iese din buclă pentru a termina algoritmul.  
2.Inițializează variabila pathFlow cu o valoare mare, în acest caz, INT\_MAX, pentru a determina fluxul maxim posibil pe cale.  
3.Parcurge drumul de la destinație către sursă (utilizând tabloul parent) și găsește fluxul minim pe acest drum.  
4.Acest flux minim reprezintă capacitatea minimă a muchiilor pe acest drum și este folosit pentru a determina cât de mult flux poate fi adăugat la fluxul total.

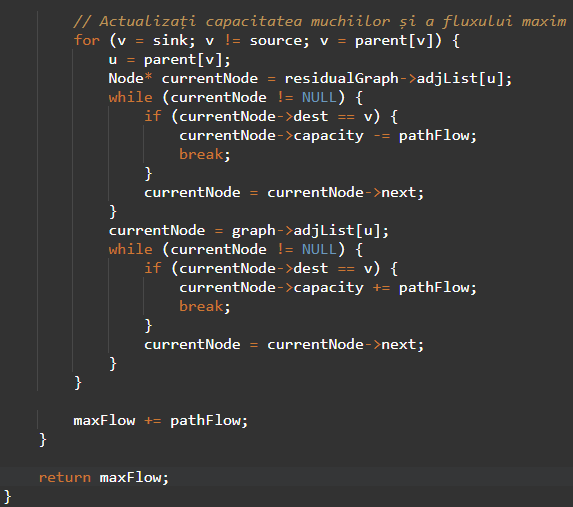


Fig.2.3.17

Această parte a algoritmului realizează următoarele operații:  
  
1.Actualizează capacitatea muchiilor în graful rezidual și capacitatea fluxului în graf în funcție de fluxul determinat pe drumul de la sursă la destinație.  
1.Pentru fiecare muchie pe drumul de la sursă la destinație, se scade fluxul determinat (pathFlow) din capacitatea muchiei corespunzătoare în graful rezidual.  
3.De asemenea, se adaugă fluxul determinat la capacitatea muchiei corespunzătoare în graf.  
4.Fluxul total (maxFlow) este actualizat prin adăugarea fluxului determinat pe drumul curent.

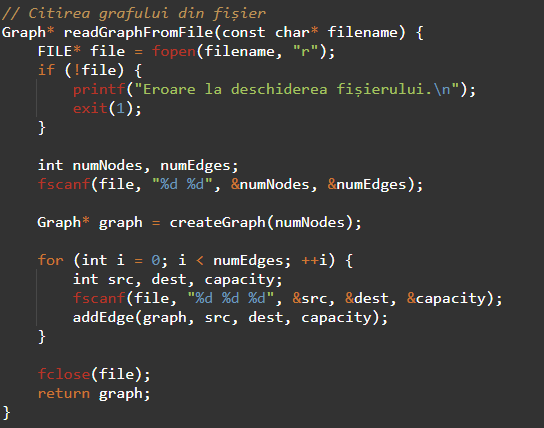


Fig.2.3.18

Această funcție readGraphFromFile() primește ca argument numele fișierului din care trebuie citit graf. Ea face următoarele lucruri:  
  
1.Deschide fișierul în modul "r" (doar pentru citire).  
2.Verifică dacă deschiderea fișierului a avut succes. În caz contrar, afișează un mesaj de eroare și încheie programul cu exit(1).  
3.Citeste numărul de noduri și muchii din fișier utilizând fscanf().  
4.Creează un graf folosind funcția createGraph() cu numărul de noduri citit anterior.  
5.Parcurge fiecare muchie din fișier și adaugă-o în graf folosind funcția addEdge().  
6.Închide fișierul după ce citirea este terminată.  
7.Returnează pointerul către graful citit.

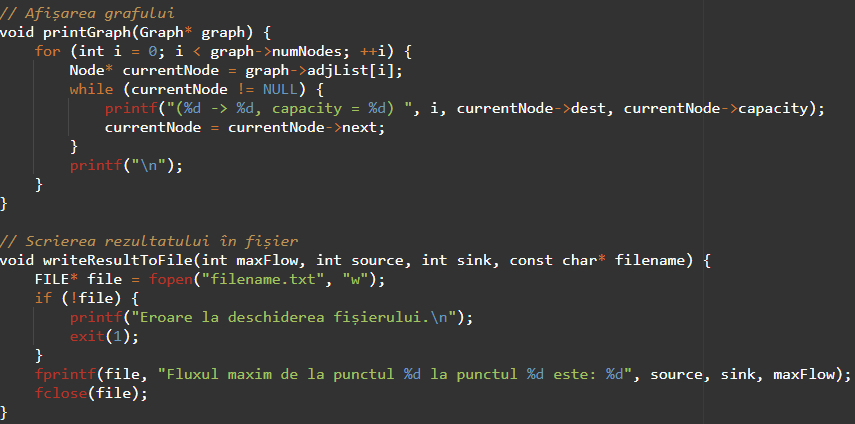


Fig.2.3.19

Această funcție printGraph() primește un pointer către un graf și afișează fiecare nod și muchie din graf. Parcurge lista de adiacență a fiecărui nod și afișează destinația și capacitatea fiecărei muchii.

Această funcție writeResultToFile() primește fluxul maxim găsit, nodul sursă, nodul destinație și numele fișierului în care trebuie scris rezultatul. Ea deschide fișierul pentru scriere și scrie rezultatul în format specificat (fluxul maxim și punctele sursă și destinație).

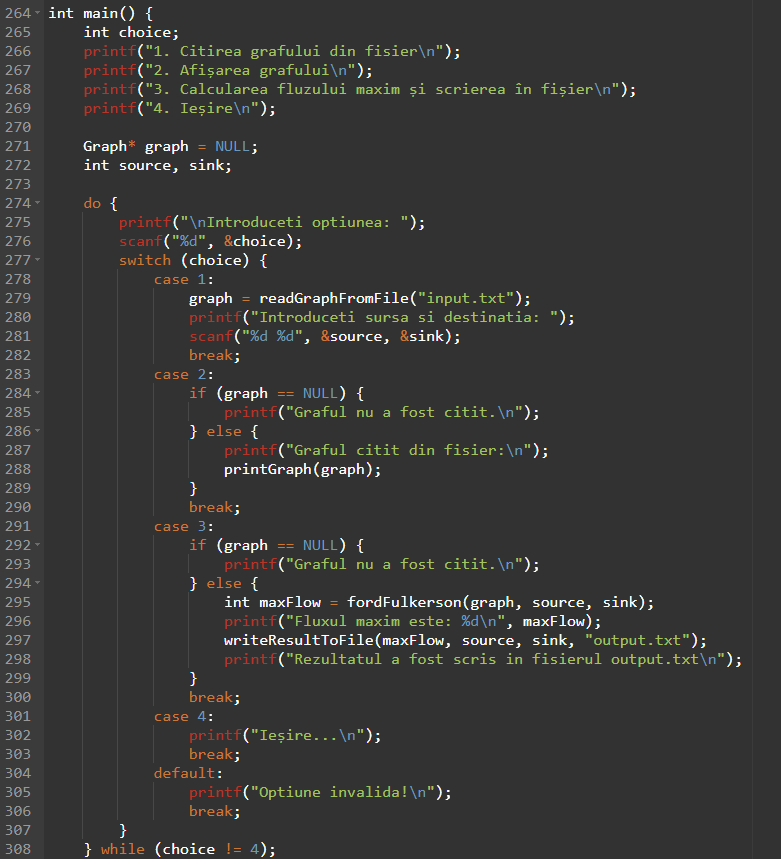


Fig.2.3.20

1.Afiseaza un meniu pentru utilizator, oferind patru opțiuni: citirea grafului din fișier, afișarea grafului, calcularea fluxului maxim și ieșirea din program.  
2.Utilizează o buclă do-while pentru a permite utilizatorului să selecteze și să execute diferite opțiuni până când alege să iasă din program.  
3.Pentru fiecare opțiune selectată, utilizează un switch-case pentru a executa codul corespunzător.  
4.În funcție de opțiunea aleasă, se citesc graful și nodurile sursă și destinație, se afișează graful, se calculează fluxul maxim și se scrie rezultatul într-un fișier, sau programul se încheie.

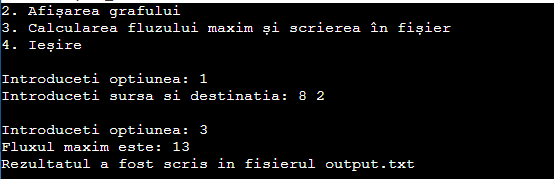


Fig.2.3.21 Outputul din console.

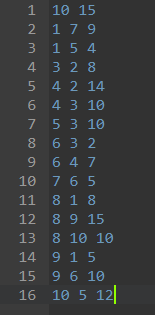


Fig.2.3.22 Fișierul de Input.

# **Concluzie**

Problema fluxului maxim în teoria grafurilor reprezintă o temă fundamentală cu aplicații vaste și profunde în diverse domenii. Această problemă apare în contextul rețelelor de transport, a telecomunicațiilor, a fluxurilor de date în computere, a planificării rețelelor și a multor alte domenii. O soluție eficientă pentru această problemă este esențială pentru optimizarea utilizării resurselor în rețelele moderne.

Algoritmul Ford-Fulkerson este una dintre cele mai utilizate metode pentru rezolvarea problemei fluxului maxim. Bazat pe ideea de căutare a unui drum de la sursă la destinație într-un graf rezidual, acest algoritm oferă o abordare eficientă și flexibilă. Implementarea algoritmului Ford-Fulkerson necesită gestionarea grafurilor reziduale, actualizarea capacităților muchiilor și o strategie eficientă de căutare a drumului de la sursă la destinație.

Codul prezentat oferă o implementare concretă a algoritmului Ford-Fulkerson, împreună cu funcții auxiliare pentru citirea și afișarea grafului. Acest cod servește ca o bază solidă pentru înțelegerea și aplicarea algoritmului în diverse scenarii practice.

În concluzie, algoritmul Ford-Fulkerson este o unealtă puternică pentru rezolvarea problemei fluxului maxim într-un graf și este esențial pentru optimizarea rețelelor în diverse domenii. Înțelegerea acestui algoritm și capacitatea de a-l aplica în mod eficient pot aduce beneficii semnificative în proiectele care implică gestionarea fluxurilor de resurse în rețele.

# **Bibliografie:**

1. Ford, L. R., Jr., & Fulkerson, D. R. - Articolul lor intitulat "Maximal flow through a network" a fost publicat în Canadian Journal of Mathematics în 1956.  
  
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - Cartea "Introduction to Algorithms" a fost publicată în 2009 de MIT Press.  
  
3. Christos H. Papadimitriou & Kenneth Steiglitz - Cartea "Combinatorial Optimization: Algorithms and Complexity" a fost publicată în 1982 de Prentice-Hall.  
  
4. Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin - Cartea "Network Flows: Theory, Algorithms, and Applications" a fost publicată în 1993 de Prentice Hall.  
  
5. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman - Cartea "The Design and Analysis of Computer Algorithms" a fost publicată în 1974 de Addison-Wesley.  
  
6. Alexander Schrijver - Cartea "Combinatorial Optimization: Polyhedra and Efficiency" a fost publicată în 2003 de Springer Science & Business Media.

# **Anexa:**

1. Algoritmul Ford-Fulkersson implemintarea în limbajul C.

#include <stdio.h>  
  
#define A 0  
  
#define B 1  
  
#define C 2  
  
#define MAX\_NODURI 1000  
  
#define O 1000000000  
  
int n = 6;  
  
int capacitate[MAX\_NODURI][MAX\_NODURI];  
  
int flux[MAX\_NODURI][MAX\_NODURI];  
  
int culoare[MAX\_NODURI];  
  
int predecesor[MAX\_NODURI];  
  
int minim(int x, int y) {  
  
  return x < y ? x : y;  
  
}  
int cap, coada;  
  
int q[MAX\_NODURI + 2];  
void adaugaLaCoada(int x) {  
  
  q[coada] = x;  
  
  coada++;  
  
  culoare[x] = B;// nod in  coada  
  
}  
  
   
int extrageDinCoada() {  
  
  int x = q[cap];  
  
  cap++;  
  
  culoare[x] = C; // nod extras din nod  
  
  return x;  
  
}  
  
   
  
// Folosind BFS ca algoritm de căutare  
  
int bfs(int start, int scop) {  
  
  int u, v;  
  
  for (u = 0; u < n; u++) {  
  
    culoare[u] = A;//Markeaza toate Nodurile cu A ca neatinse  
  
  }  
  
  cap = coada = 0; //Inițializarea capului și cozii la 0.  
  
  adaugaLaCoada(start);//adaug nodul start la coada  
  
  predecesor[start] = -1;  
  
  while (cap != coada) {// Intrare in while atâta timp cât coada nu este goală.  
  
    u = extrageDinCoada();//se extrage un nod din coada  
  
    for (v = 0; v < n; v++) {// for unde trecem prin toate nodurile  
  
      if (culoare[v] == A && capacitate[u][v] - flux[u][v] > 0) { // verifica daca este legatura intre nod u si v si ca nodul v nu a fost vizitat  
  
        adaugaLaCoada(v);  
  
        predecesor[v] = u;  
  
      }  
  
    }  
  
  }  
  
  return culoare[scop] == C;//returneaza C daca nodul a fost vizitat  
}  
  
// Aplicarea algoritmului Ford-Fulkerson  
  
int fordFulkerson(int sursa, int scurgere) {  
  
  int i, j, u;  
  
  int fluxMaxim = 0;  
  
  for (i = 0; i < n; i++) {  
  
    for (j = 0; j < n; j++) {  
  
      flux[i][j] = 0;  
  
    }  
  
 }  
  
  // Actualizarea valorilor reziduale ale muchiilor  
  
  while (bfs(sursa, scurgere)) {//se executa acest while atata timp cat este drum de la sursa la scurgere  
  
    int increment = O;//  
  
    for (u = scurgere; predecesor[u] >= 0; u = predecesor[u]) {//Parcurge drumul de la scurgere la sursă și găsește incrementul minim.  
  
      increment = minim(increment, capacitate[predecesor[u]][u] - flux[predecesor[u]][u]);  
  
    }  
  
    for (u = scurgere; predecesor[u] >= 0; u = predecesor[u]) {  
  
      flux[predecesor[u]][u] += increment; //se aduna la o muchie capacitatea or  
  
       flux[u][predecesor[u]] -= increment;//se scade din capacitatea muchilor  
  
    }  
  
    // Adăugarea fluxurilor de pe traseu  
  
    fluxMaxim += increment;  
  
  }  
  
  return fluxMaxim;  
  
}  
int main() {  
  
  for (int i = 0; i < n; i++) {  
  
    for (int j = 0; j < n; j++) {  
  
      capacitate[i][j] = 0;  
  
    }  
  
  }  
  capacitate[0][1] = 8;  
  
  capacitate[0][4] = 3;  
  
  capacitate[1][2] = 9;  
  
  capacitate[2][4] = 7;  
  
  capacitate[2][5] = 2;  
  
  capacitate[3][5] = 5;  
  capacitate[4][2] = 7;  
  
  capacitate[4][3] = 4;  
  
  int sursa = 0, scurgere = 5;  
  
  printf("Fluxul maxim: %d\n", fordFulkerson(sursa, scurgere));  
  
  return 0;  
  
}

1. Problema Fluxul maxim implimentarea ]n limabjul C.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Definirea structurii pentru un nod din graf

typedef struct Node {

int dest;

int capacity;

struct Node\* next;

} Node;

// Definirea structurii pentru graf

typedef struct Graph {

int numNodes;

Node\*\* adjList;

} Graph;

// Definirea structurii pentru stivă

typedef struct StackNode {

int data;

struct StackNode\* next;

} StackNode;

typedef struct Stack {

StackNode\* top;

} Stack;

// Definirea structurii pentru coadă

typedef struct QueueNode {

int data;

struct QueueNode\* next;

} QueueNode;

typedef struct Queue {

QueueNode \*front, \*rear;

} Queue;

// Inițializare stivă

Stack\* createStack() {

Stack\* stack = (Stack\*)malloc(sizeof(Stack));

stack->top = NULL;

return stack;

}

// Verifică dacă stiva este goală

int isEmptyStack(Stack\* stack) {

return (stack->top == NULL);

}

// Adăugare la stivă

void push(Stack\* stack, int data) {

StackNode\* newNode = (StackNode\*)malloc(sizeof(StackNode));

newNode->data = data;

newNode->next = stack->top;

stack->top = newNode;

}

// Eliminare din stivă

int pop(Stack\* stack) {

if (isEmptyStack(stack)) {

printf("Stack underflow\n");

return -1;

}

StackNode\* temp = stack->top;

int popped = temp->data;

stack->top = stack->top->next;

free(temp);

return popped;

}

// Inițializare coadă

Queue\* createQueue() {

Queue\* queue = (Queue\*)malloc(sizeof(Queue));

queue->front = queue->rear = NULL;

return queue;

}

// Verifică dacă coada este goală

int isEmptyQueue(Queue\* queue) {

return (queue->front == NULL);

}

// Adăugare la coadă

void enqueue(Queue\* queue, int data) {

QueueNode\* newNode = (QueueNode\*)malloc(sizeof(QueueNode));

newNode->data = data;

newNode->next = NULL;

if (isEmptyQueue(queue)) {

queue->front = queue->rear = newNode;

return;

}

queue->rear->next = newNode;

queue->rear = newNode;

}

// Eliminare din coadă

int dequeue(Queue\* queue) {

if (isEmptyQueue(queue)) {

printf("Queue underflow\n");

return -1;

}

QueueNode\* temp = queue->front;

int dequeued = temp->data;

queue->front = queue->front->next;

if (queue->front == NULL)

queue->rear = NULL;

free(temp);

return dequeued;

}

// Inițializare graf

Graph\* createGraph(int numNodes) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->numNodes = numNodes;

graph->adjList = (Node\*\*)malloc(numNodes \* sizeof(Node\*));

for (int i = 0; i < numNodes; ++i)

graph->adjList[i] = NULL;

return graph;

}

// Adăugare muchie în graf

void addEdge(Graph\* graph, int src, int dest, int capacity) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->dest = dest;

newNode->capacity = capacity;

newNode->next = graph->adjList[src];

graph->adjList[src] = newNode;

}

// Algoritmul Ford-Fulkerson pentru găsirea fluxului maxim

int fordFulkerson(Graph\* graph, int source, int sink) {

int u, v;

// Creați o copie a grafului pentru a stoca fluxul

Graph\* residualGraph = createGraph(graph->numNodes);

// Inițializați graful rezidual cu aceleași muchii ca și graficul original

for (u = 0; u < graph->numNodes; u++) {

Node\* temp = graph->adjList[u];

while (temp != NULL) {

addEdge(residualGraph, u, temp->dest, temp->capacity); // Inițializare cu capacitatea din graficul original

temp = temp->next;

}

}

int\* parent = (int\*)malloc(graph->numNodes \* sizeof(int));

int maxFlow = 0;

// Cat timp există un drum de la sursă la destinație

while (1) {

// BFS pentru a găsi un drum de la sursă la destinație în graful rezidual

for (u = 0; u < graph->numNodes; u++)

parent[u] = -1;

parent[source] = -2;

Queue\* queue = createQueue();

enqueue(queue, source);

while (!isEmptyQueue(queue)) {

u = dequeue(queue);

Node\* currentNode = residualGraph->adjList[u];

while (currentNode != NULL) {

v = currentNode->dest;

if (parent[v] == -1 && currentNode->capacity > 0) {

parent[v] = u;

enqueue(queue, v);

}

currentNode = currentNode->next;

}

}

free(queue);

// Dacă nu există niciun drum de la sursă la destinație, ieșiți din buclă

if (parent[sink] == -1)

break;

int pathFlow = INT\_MAX;

for (v = sink; v != source; v = parent[v]) {

u = parent[v];

Node\* currentNode = residualGraph->adjList[u];

while (currentNode != NULL) {

if (currentNode->dest == v) {

pathFlow = (pathFlow < currentNode->capacity) ? pathFlow : currentNode->capacity;

break;

}

currentNode = currentNode->next;

}

}

// Actualizați capacitatea muchiilor și a fluxului maxim

for (v = sink; v != source; v = parent[v]) {

u = parent[v];

Node\* currentNode = residualGraph->adjList[u];

while (currentNode != NULL) {

if (currentNode->dest == v) {

currentNode->capacity -= pathFlow;

break;

}

currentNode = currentNode->next;

}

currentNode = graph->adjList[u];

while (currentNode != NULL) {

if (currentNode->dest == v) {

currentNode->capacity += pathFlow;

break;

}

currentNode = currentNode->next;

}

}

maxFlow += pathFlow;

}

return maxFlow;

}

// Citirea grafului din fișier

Graph\* readGraphFromFile(const char\* filename) {

FILE\* file = fopen(filename, "r");

if (!file) {

printf("Eroare la deschiderea fișierului.\n");

exit(1);

}

int numNodes, numEdges;

fscanf(file, "%d %d", &numNodes, &numEdges);

Graph\* graph = createGraph(numNodes);

for (int i = 0; i < numEdges; ++i) {

int src, dest, capacity;

fscanf(file, "%d %d %d", &src, &dest, &capacity);

addEdge(graph, src, dest, capacity);

}

fclose(file);

return graph;

}

// Afișarea grafului

void printGraph(Graph\* graph) {

for (int i = 0; i < graph->numNodes; ++i) {

Node\* currentNode = graph->adjList[i];

while (currentNode != NULL) {

printf("(%d -> %d, capacity = %d) ", i, currentNode->dest, currentNode->capacity);

currentNode = currentNode->next;

}

printf("\n");

}

}

// Scrierea rezultatului în fișier

void writeResultToFile(int maxFlow, int source, int sink, const char\* filename) {

FILE\* file = fopen("filename.txt", "w");

if (!file) {

printf("Eroare la deschiderea fișierului.\n");

exit(1);

}

fprintf(file, "Fluxul maxim de la punctul %d la punctul %d este: %d", source, sink, maxFlow);

fclose(file);

}

int main() {

int choice;

printf("1. Citirea grafului din fisier\n");

printf("2. Afișarea grafului\n");

printf("3. Calcularea fluzului maxim și scrierea în fișier\n");

printf("4. Ieșire\n");

Graph\* graph = NULL;

int source, sink;

do {

printf("\nIntroduceti optiunea: ");

scanf("%d", &choice);

switch (choice) {

case 1:

graph = readGraphFromFile("input.txt");

printf("Introduceti sursa si destinatia: ");

scanf("%d %d", &source, &sink);

break;

case 2:

if (graph == NULL) {

printf("Graful nu a fost citit.\n");

} else {

printf("Graful citit din fisier:\n");

printGraph(graph);

}

break;

case 3:

if (graph == NULL) {

printf("Graful nu a fost citit.\n");

} else {

int maxFlow = fordFulkerson(graph, source, sink);

printf("Fluxul maxim este: %d\n", maxFlow);

writeResultToFile(maxFlow, source, sink, "output.txt");

printf("Rezultatul a fost scris in fisierul output.txt\n");

}

break;

case 4:

printf("Ieșire...\n");

break;

default:

printf("Optiune invalida!\n");

break;

}

} while (choice != 4);

return 0;

}

Problema 3:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

//Structura pentru a reprezenta un nod al grafului

typedef struct Node {

int destination; //Nodul destinație al arcului

int capacity; //Capacitatea maximă a arcului

int flow; //Fluxul curent pe arc

struct Node \*next; //Pointer către următorul arc în listă

} Node;

//Funcție pentru a crea un nod nou

Node\* createNode(int destination, int capacity) {

Node\* newNode = (Node\*)malloc(sizeof(Node)); // Alocă memorie pentru un nou nod

newNode->destination = destination; // Inițializează nodul cu destinația dată

newNode->capacity = capacity; // Inițializează capacitatea nodului cu capacitatea dată

newNode->flow = 0; // Inițializează fluxul nodului cu 0

newNode->next = NULL; // Inițializează următorul nod cu NULL

return newNode; // Returnează noul nod creat

}

//Prototipuri de funcții

void addEdge(Node\*\* graph, int src, int destination, int capacity); // Prototip pentru funcția de adăugare a unui arc la graf

bool dfs(Node\*\* graph, int source, int sink, int parent[], bool visited[]); // Prototip pentru funcția de căutare DFS

int fordFulkerson(Node\*\* graph, int numNodes, int source, int sink); // Prototip pentru funcția de găsire a fluxului maxim

int main() {

//Definește numărul de noduri în graf

int numNodes = 8;

//Crearea listei de adiacență pentru graful nostru

Node\*\* graph = (Node\*\*)malloc(numNodes \* sizeof(Node\*)); // Alocă memorie pentru lista de adiacență

for (int i = 0; i < numNodes; i++) {

graph[i] = NULL; // Inițializează fiecare element al listei cu NULL

}

//Adăugarea arcelor la graf

addEdge(graph, 0, 1, 140);

addEdge(graph, 0, 2, 60);

addEdge(graph, 0, 3, 40);

addEdge(graph, 1, 4, 100);

addEdge(graph, 1, 5, 80);

addEdge(graph, 1, 6, 20);

addEdge(graph, 2, 5, 40);

addEdge(graph, 2, 6, 80);

addEdge(graph, 3, 6, 160);

addEdge(graph, 3, 7, 160);

//Se definește sursa și destinația

int source = 0;

int sink = numNodes - 1;

//Se găsește fluxul maxim în graf

int maxFlow = fordFulkerson(graph, numNodes, source, sink);

//Se deschide fișierul pentru a scrie rezultatul, apoi se închide

FILE \*f = fopen("main.txt", "w");

if (f == NULL) {

printf("Eroare la deschiderea fisierului de iesire.\n");

return 1; // Încheie programul cu cod de eroare

}

fprintf(f, "Fluxul maxim de marfuri ce poate fi transportat este: %d\n", maxFlow);

fclose(f);

printf("Fluxul maxim de marfuri ce poate fi transportat este: %d\n", maxFlow); // Afișează rezultatul în consolă

//Se eliberează memoria alocată pentru graf

for (int i = 0; i < numNodes; i++) {

Node\* curr = graph[i];

while (curr != NULL) {

Node\* next = curr->next;

free(curr);

curr = next;

}

}

free(graph);

return 0;

}

//Funcție pentru a adăuga un arc la graf

void addEdge(Node\*\* graph, int src, int destination, int capacity) {

Node\* newNode = createNode(destination, capacity); // Creează un nou nod

if (graph[src] == NULL) { // Dacă nu există arce pentru nodul sursă

graph[src] = newNode; // Atribuie noul nod ca primul nod din listă pentru sursa dată

} else { // Altfel

Node\* temp = graph[src]; // Atribuie nodul sursă curent

while (temp->next != NULL) { // Cat timp există un următor nod în listă

temp = temp->next; // Avansează la următorul nod

}

temp->next = newNode; // Adaugă noul nod la sfârșitul listei

}

}

//Funcția de parcurgere în adâncime

bool dfs(Node\*\* graph, int source, int sink, int parent[], bool visited[]) {

visited[source] = true; // Marchează nodul ca vizitat

for (Node\* curr = graph[source]; curr != NULL; curr = curr->next) { // Iterează prin vecinii nodului sursă

int v = curr->destination; // Atribuie destinația arcului curent

if (!visited[v] && curr->capacity > 0) { // Dacă nodul destinație nu a fost vizitat și capacitatea arcului este mai mare decât 0

parent[v] = source; // Marchează nodul sursă ca părinte al nodului destinație

if (v == sink) // Dacă nodul destinație este destinația finală

return true; // Returnează adevărat

if (dfs(graph, v, sink, parent, visited)) // Caută recursiv în josul arborelui

return true; // Returnează adevărat dacă s-a găsit drum până la destinație

}

}

return false; // Returnează fals dacă nu s-a găsit niciun drum de creștere

}

//Funcția pentru a găsi fluxul maxim în graf folosind algoritmul Ford-Fulkerson

int fordFulkerson(Node\*\* graph, int numNodes, int source, int sink) {

int maxFlow = 0; // Inițializează fluxul maxim cu 0

int parent[numNodes]; // Creează un vector pentru a urmări părinții nodurilor

bool visited[numNodes]; // Creează un vector pentru a marca nodurile vizitate

//Inițializare

for (int i = 0; i < numNodes; i++) { // Iterează prin fiecare nod al grafului

visited[i] = false; // Marchează nodul ca nevizitat

parent[i] = -1; // Inițializează părintele nodului cu -1

}

//Căutăm drumuri de creștere și actualizăm fluxul până când nu mai există astfel de drumuri

while (dfs(graph, source, sink, parent, visited)) { // Cat timp se găsesc drumuri de creștere

int pathFlow = INT\_MAX; // Inițializează fluxul drumului cu o valoare maximă posibilă

for (int v = sink; v != source; v = parent[v]) { // Iterează prin nodurile de la destinație înapoi către sursă

int u = parent[v]; // Atribuie părintele nodului curent

for (Node\* curr = graph[u]; curr != NULL; curr = curr->next) { // Iterează prin arcele nodului părinte

if (curr->destination == v) { // Dacă destinația arcului este nodul curent

pathFlow = (curr->capacity < pathFlow) ? curr->capacity : pathFlow; // Atribuie fluxul minim dintre capacitatea arcului și fluxul curent

break; // Ieși din bucla for

}

}

}

//Actualizăm capacitatea și fluxul pe fiecare arc de-a lungul drumului de creștere

for (int v = sink; v != source; v = parent[v]) { // Iterează prin nodurile de la destinație înapoi către sursă

int u = parent[v]; // Atribuie părintele nodului curent

for (Node\* curr = graph[u]; curr != NULL; curr = curr->next) { // Iterează prin arcele nodului părinte

if (curr->destination == v) { // Dacă destinația arcului este nodul curent

curr->capacity -= pathFlow; // Scade fluxul drumului din capacitatea arcului

curr->flow += pathFlow; // Adaugă fluxul drumului la fluxul arcului

break; // Ieșire din bucla for

}

}

for (Node\* curr = graph[v]; curr != NULL; curr = curr->next) { // Iterează prin arcele nodului curent

if (curr->destination == u) { // Dacă destinația arcului este părintele nodului curent

curr->capacity += pathFlow; // Adaugă fluxul drumului la capacitatea arcului

curr->flow -= pathFlow; // Scade fluxul drumului din fluxul arcului

break; // Ieșire din bucla for

}

}

}

//Adăugăm fluxul de-a lungul drumului de creștere la fluxul total maxim

maxFlow += pathFlow; // Adaugă fluxul drumului la fluxul total maxim

//Resetează marcările pentru următoarea căutare DFS

for (int i = 0; i < numNodes; i++) { // Iterează prin fiecare nod al grafului

visited[i] = false; // Marchează nodul ca nevizitat

}

}

return maxFlow; // Returnează fluxul maxim găsit

}