### 

|  |  |
| --- | --- |
|  | **Universitatea Tehnică a Moldovei** |

**METODA BACKTRACKING : ANALIZA , IMPLEMENTARE , ARIE DE UTILIZARE.**

**A efectuat:** **st. gr. MI-222 Calancea Catalin**

**A verificat: asis.univ., dr. Gutu Maria**

Chişinău, 2023

Cuprins

[INTRODUCERE 3](#_Toc134822004)

[1. FUNDAMENTUL TEORETIC 4](#_Toc134822005)

[1.1. Definiția și caracteristicile metodei backtracking 4](#_Toc134822006)

[1.2. Caracteristici ale metodei backtracking 4](#_Toc134822007)

[1.3. Avantajele si dezavantajele metodei backtracking 5](#_Toc134822008)

[1.4. Algoritmul general pentru backtracking 6](#_Toc134822009)

[1.5. Exemple de probleme rezolvate cu ajutorul backtracking-ului 6](#_Toc134822010)

[1.6. Compararea metodei backtracking cu alte metode de rezolvare a problemelor 7](#_Toc134822011)

[1.7. Demonstrarea relației dintre backtracking și metodele de explorare a spațiului de căutare 7](#_Toc134822012)

[1.8. Complexitatea algoritmului de backtracking 7](#_Toc134822013)

[1.8.1. **Analiza complexității algoritmului de backtracking pentru diverse probleme** 8](#_Toc134822014)

[1.8.2. **Exemplificarea unor tehnici de optimizare a complexității algoritmului de backtracking** 8](#_Toc134822015)

[1.9. Implementarea eficientă a algoritmului de backtracking 9](#_Toc134822016)

[1.10. Tehnici de optimizare a algoritmului de backtracking 9](#_Toc134822017)

[1.10.1. **Ordinea de explorare a stărilor** 9](#_Toc134822018)

[1.10.2. **Euristicile de alegere a opțiunilor** 10](#_Toc134822019)

[1.10.3. **Tehnici de optimizare bazate pe informații specifice problemei** 10](#_Toc134822020)

[1.11. Tehnici de optimizare bazate pe informații specifice problemei 10](#_Toc134822021)

[1.11.1. Pruning prin verificarea condițiilor de soluție parțială. 11](#_Toc134822022)

[1.11.2. Pruning prin verificarea constrângerilor și regulelor problemei 11](#_Toc134822023)

[1.11.3. Pruning prin aplicarea limitelor superioare și inferioare 12](#_Toc134822024)

[2. APLICATII PRACTICE 13](#_Toc134822025)

[2.1. Algoritmul backtracking-ului. 13](#_Toc134822026)

[2.2. Implementarea backtracking-ului în limbajul de programare C 13](#_Toc134822027)

[2.3. Probleme de optimizare 14](#_Toc134822028)

[2.3.1. **Problema rucsacului** 14](#_Toc134822029)

[2.3.2. **Problema colorării grafului** 17](#_Toc134822030)

[2.3.3. **Problema TSP (Travelling Salesman Problem)** 19](#_Toc134822031)

[2.4. Jocuri 22](#_Toc134822032)

[2.4.1. **Sudoku** 22](#_Toc134822033)

[2.4.2. **Jocul 8-regine** 26](#_Toc134822034)

[2.5. Analiza de date 28](#_Toc134822035)

[2.5.1. **Selecția de date dintr-o bază de date** 28](#_Toc134822036)

[2.5.2. **Identificarea de pattern-uri in date** 29](#_Toc134822037)

[CONCLUZII 32](#_Toc134822038)

[BIBLIOGRAFIE 33](#_Toc134822039)

[ANEXE 34](#_Toc134822040)

# INTRODUCERE

În aceasta lucrare , vom explora conceptul și utilizarea metodei backtracking în sfera IT. Metoda backtracking este o tehnică eficientă utilizată în rezolvarea problemelor de căutare, optimizare și explorare a spațiului de soluții. Această metodă se bazează pe explorarea sistematică a tuturor posibilităților pentru a găsi soluții valide la o problemă dată.

Vom explora caracteristicile și principiile metodei backtracking, evidențiind avantajele și dezavantajele sale. De asemenea, vom analiza algoritmul general pentru backtracking și modul în care poate fi implementat în diferite limbaje de programare.

Scopul acestei teze este de a analiza în detaliu metoda backtracking, de la conceptele teoretice până la aplicațiile practice în sfera IT. Teza își propune să ofere o înțelegere profundă a metodei backtracking, precum și a modului în care poate fi implementată și utilizată eficient în rezolvarea problemelor complexe.

În cadrul acestei teze, vom explora exemple de probleme care pot fi rezolvate cu ajutorul metodei backtracking și vom analiza complexitatea algoritmului de backtracking în diverse contexte. De asemenea, vom evidenția ariile de utilizare ale metodei backtracking și vom examina modul în care poate contribui la optimizarea proceselor, rezolvarea problemelor de analiză de date și proiectarea de circuite electrice.

Prin urmare, scopul final al tezei este de a oferi cititorilor o înțelegere solidă a metodei backtracking și a capacității sale de a rezolva probleme complexe în sfera IT.

1. **FUNDAMENTUL TEORETIC**
   1. **Definiția și caracteristicile metodei backtracking**

Metoda backtracking reprezintă o tehnică de rezolvare a problemelor bazată pe încercarea sistematică a tuturor posibilităților, urmărind o serie de etape de explorare și revenire în urmă în cazul în care o soluție se dovedește invalidă. Această metodă se folosește de explorarea recursivă a unui spațiu de căutare pentru a găsi soluții valide.

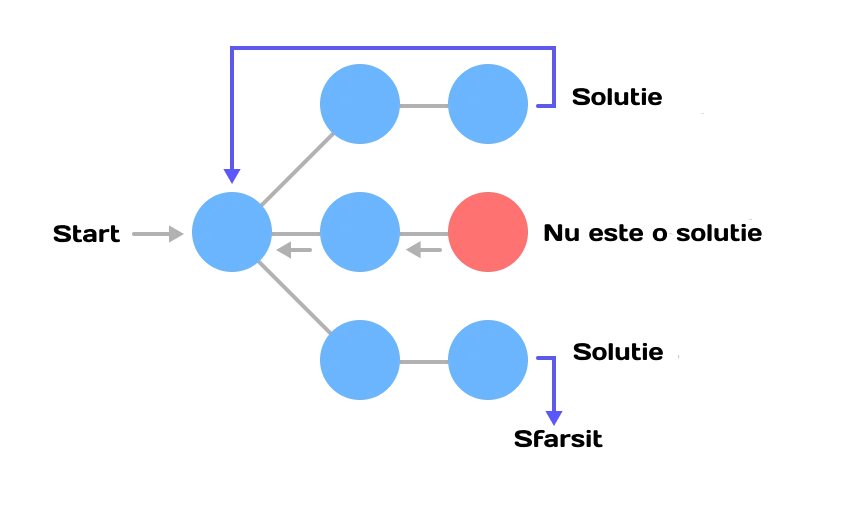


Figura 1.1 Metoda Backtracking

* 1. **Caracteristici ale metodei backtracking**
* Explorare sistematică: Metoda backtracking explorează sistematic toate posibilitățile într-un spațiu de soluții, verificând fiecare combinație și revenind în urmă în cazul în care o soluție nu este viabilă.
* Recursivitate: Implementarea metodei backtracking se bazează în mod obișnuit pe recursivitate, în care se realizează apeluri recursive pentru explorarea spațiului de soluții și revenirea în urmă în cazul în care o soluție nu este validă.
* Constrângeri și condiții de validare: Metoda backtracking funcționează în baza unor constrângeri și condiții de validare specifice fiecărei probleme, asigurându-se că soluțiile generate respectă cerințele problemei respective.

|  |  |
| --- | --- |
| Avantaje | Dezavantaje |
| * Explorare exhaustivă: Metoda backtracking permite explorarea sistematică a tuturor posibilităților, asigurând găsirea unei soluții valide în cazul în care există una. Aceasta poate fi deosebit de utilă în problemele în care este necesară acoperirea completă a spațiului de soluții. | * Complexitatea în cazul spațiilor de căutare mari: În cazul în care spațiul de căutare este foarte mare, metoda backtracking poate deveni ineficientă din punct de vedere al timpului și al memoriei necesare. Explorarea tuturor combinațiilor posibile poate duce la o complexitate exponentială. |
| * Eficiență în cazul problemelor combinatorice: Metoda backtracking este eficientă în rezolvarea problemelor combinatorice, cum ar fi generarea combinațiilor sau permutărilor. Prin utilizarea constrângerilor și a verificărilor de validitate, se poate evita generarea și evaluarea inutilă a unor soluții nevalide. | * Necesitatea unor constrângeri bine definite: Pentru a utiliza metoda backtracking, este necesară definirea clară a constrângerilor și regulilor problemei. Fără constrângeri bine definite, algoritmul poate genera soluții invalide sau poate deveni inutilizabil. |
| * Flexibilitate: Metoda backtracking este flexibilă și poate fi adaptată pentru a rezolva diverse tipuri de probleme. Algoritmul general poate fi ajustat și personalizat în funcție de specificul problemei și a constrângerilor impuse. |  |

* 1. **Avantajele si dezavantajele metodei backtracking**

Tabel 1.1 Avantajele si dezavantajele backtracking-ului

* 1. **Algoritmul general pentru backtracking**

Implementarea acestui algoritm implică crearea unei funcții recursive care să parcurgă acești pași și să realizeze explorarea și căutarea soluțiilor în spațiul de căutare specific problemei.

Algoritmul general pentru backtracking constă în următorii pași:

1. Inițializarea problemei și a datelor de intrare.
2. Verificarea condiției de oprire și returnarea soluției în cazul în care aceasta este găsită.
3. Generarea posibilelor opțiuni sau soluții parțiale.
4. Verificarea validității fiecărei opțiuni generate.
5. Realizarea unui apel recursiv pentru a explora în continuare spațiul de soluții.
6. Revenirea în urmă în cazul în care o soluție nu este validă și continuarea explorării cu alte opțiuni.
   1. **Exemple de probleme rezolvate cu ajutorul backtracking-ului**

Metoda backtracking poate fi utilizată pentru rezolvarea unor probleme diverse, printre care se numără:

* Sudoku: Problema de completare a unui joc Sudoku, în care trebuie să se găsească o soluție validă care respectă regulile jocului.
* Generarea combinațiilor și permutărilor: Problema de generare a tuturor combinațiilor sau permutărilor unui set de elemente, respectând anumite constrângeri sau reguli.
* Problema șahului (N-regine): Problema de plasare a N regine pe o tablă de șah de dimensiune dată, astfel încât să nu existe conflicte între ele (nicio regină să nu poată ataca alta).
* Căutarea unui drum într-un labirint: Problema de găsire a unui drum valid într-un labirint de la punctul de start la punctul final, evitând obstacolele și respectând regulile de mișcare.
* Problema colorării grafului: Problema de colorare a nodurilor unui graf astfel încât două noduri adiacente să nu aibă aceeași culoare.

Metoda oferă un cadru flexibil și puternic pentru rezolvarea unor probleme complexe care implică explorarea și căutarea în spații de soluții. Implementarea metodei backtracking trebuie adaptată și personalizată în funcție de specificul fiecărei probleme și a constrângerilor impuse.

* 1. **Compararea metodei backtracking cu alte metode de rezolvare a problemelor**
* Forța brută (brute force): Metoda backtracking este similară cu abordarea de forță brută, în care se încearcă toate posibilitățile pentru a găsi o soluție validă. Cu toate acestea, backtracking-ul are avantajul de a evita generarea și evaluarea inutilă a unor soluții nevalide, prin aplicarea constrângerilor și a verificărilor de validitate.
* Algoritmi de optimizare: În comparație cu algoritmi de optimizare, cum ar fi algoritmii genetici sau algoritmi bazati pe traiectorii, metoda backtracking este mai simplă și mai ușor de implementat. Cu toate acestea, poate exista o diferență în eficiența și viteza de convergență a acestor metode, în funcție de problema specifică.
  1. **Demonstrarea relației dintre backtracking și metodele de explorare a spațiului de căutare**

Metoda backtracking și metodele de explorare a spațiului de căutare, cum ar fi BFS și DFS, sunt strâns legate și există o relație între ele. Iată cum se manifestă această relație:

* Backtracking ca o formă specifică a DFS: Metoda backtracking poate fi considerată o implementare specifică a Depth-First Search (DFS). În backtracking, se explorează recursiv toate posibilitățile, mergând cât mai adânc într-o direcție și revenind în urmă atunci când se ajunge într-o stare care nu duce la o soluție validă. Astfel, backtracking-ul este o formă specifică a DFS, care este adaptată pentru rezolvarea problemelor particulare.
* Utilizarea backtracking-ului în DFS: În implementarea algoritmului DFS, backtracking-ul poate fi folosit pentru a reveni în urmă și a explora alte stări nevizitate. După ce o anumită stare a fost explorată, backtracking-ul poate fi aplicat pentru a reveni în starea anterioară și a explora alte opțiuni neexplorate. Aceasta ajută la parcurgerea eficientă a spațiului de căutare într-un mod adânc, explorând în profunzime.

Backtracking-ul poate fi considerat o implementare specifică a DFS, iar metodele de explorare a spațiului de căutare pot fi utilizate pentru a optimiza procesul de backtracking și pentru a ghida explorarea într-un mod eficient. Aceste abordări sunt utile în rezolvarea unor probleme complexe și în căutarea soluțiilor într-un spațiu de căutare extins.

* 1. **Complexitatea algoritmului de backtracking**

Complexitatea algoritmului de backtracking poate varia în funcție de specificul problemei și de constrângerile impuse. În general, complexitatea algoritmului de backtracking este determinată de dimensiunea spațiului de căutare și de numărul de stări care trebuie explorate.

* + 1. **Analiza complexității algoritmului de backtracking pentru diverse probleme**

Pentru a analiza complexitatea algoritmului de backtracking pentru o problemă specifică, trebuie să luăm în considerare următoarele aspecte:

* Dimensiunea spațiului de căutare: Aceasta reprezintă numărul total de stări posibile în problema dată. De exemplu, în problema reginelor (N-regine), dimensiunea spațiului de căutare este determinată de numărul de poziții posibile pe tabla de șah, N2.
* Numărul de stări care trebuie explorate: Acesta reprezintă numărul de stări care trebuie parcurse pentru a găsi o soluție validă. De exemplu, în problema reginelor, trebuie să explorăm toate combinațiile posibile de plasare a reginelor pe tabla de șah.
* Constrângerile și regulele problemei: Acestea pot afecta numărul de stări posibile și pot influența complexitatea algoritmului de backtracking. De exemplu, în problema Sudoku, constrângerile referitoare la valorile și pozițiile caselor goale reduc numărul de stări care trebuie explorate.

Complexitatea algoritmului de backtracking poate fi exprimată în diferite moduri, precum în funcție de timpul de execuție (numărul de operații necesare), de spațiul ocupat în memorie sau de numărul de stări explorate.

* + 1. **Exemplificarea unor tehnici de optimizare a complexității algoritmului de backtracking**

Pentru a optimiza complexitatea algoritmului de backtracking, pot fi aplicate următoarele tehnici:

* Reducerea spațiului de căutare: În unele probleme, spațiul de căutare poate fi redus prin eliminarea stărilor care nu pot duce la o soluție validă. Acest lucru poate fi realizat prin aplicarea unor constrângeri și regule specifice problemei.
* Pruning (tăierea): Pruning-ul este o tehnică prin care se elimină explorarea unor porțiuni inutile ale spațiului de căutare. Prin identificarea anumitor proprietăți sau condiții care duc la soluții incorecte, putem evita explorarea acelor stări.
* Backtracking inteligent: În unele probleme, putem aplica o strategie inteligentă de alegere a stărilor următoare pentru a reduce numărul de stări care trebuie explorate. Aceasta poate fi realizată prin prioritizarea stărilor care sunt mai susceptibile de a duce la o soluție validă sau prin aplicarea unor euristică specifice problemei.
* Tehnici de memorizare (memorizare): Dacă în cadrul algoritmului de backtracking există suprapunere între subprobleme, putem utiliza tehnici de memorizare pentru a stoca și reutiliza rezultatele parțiale. Aceasta poate reduce semnificativ numărul de stări care trebuie explorate, evitând recalcularea acestora.
* Utilizarea algoritmilor de optimizare a căutării: Unele probleme pot beneficia de aplicarea unor algoritmi de optimizare a căutării, precum algoritmi genetici, algoritmi evolutivi sau algoritmi de căutare locală. Acești algoritmi pot furniza soluții mai rapide și mai bune în anumite scenarii.

Aplicarea acestor tehnici poate duce la îmbunătățirea performanței algoritmului, reducerea timpului de execuție și gestionarea mai eficientă a spațiului de căutare.Este important de menționat că optimizarea complexității algoritmului de backtracking depinde de specificul problemei și de constrângerile impuse. Nu există o soluție universală, iar alegerea tehnicilor potrivite depinde de analiza detaliată a problemei și de încercări iterative pentru a identifica cele mai eficiente strategii.

* 1. **Implementarea eficientă a algoritmului de backtracking**

Implementarea eficientă a algoritmului de backtracking este esențială pentru a obține rezultatele dorite într-un timp rezonabil. Backtracking-ul este o tehnică de explorare sistematică a spațiului de căutare al unei probleme, cu scopul de a găsi o soluție sau de a explora toate soluțiile posibile.

Pentru a optimiza implementarea algoritmului de backtracking, putem aplica o serie de tehnici care vizează diferite aspecte ale acestuia. Aceste tehnici includ optimizarea ordinei de explorare a stărilor, utilizarea euristicilor pentru alegerea opțiunilor și aplicarea de informații specifice problemei pentru a ghida procesul de backtracking.

* 1. **Tehnici de optimizare a algoritmului de backtracking**

Tehnicile de optimizare a algoritmului de backtracking reprezintă un aspect crucial în îmbunătățirea performanței acestui algoritm. Prin aplicarea unor strategii precum optimizarea ordinei de explorare a stărilor, utilizarea euristicilor pentru alegerea opțiunilor și integrarea de informații specifice problemei, se poate obține o implementare eficientă a algoritmului de backtracking.

* + 1. **Ordinea de explorare a stărilor**

Ordinea în care sunt explorate stările în cadrul algoritmului de backtracking poate avea un impact semnificativ asupra performanței acestuia. O ordine de explorare bine aleasă poate duce la reducerea numărului total de stări evaluate și, implicit, la o soluție mai rapidă. Există mai multe strategii posibile pentru a determina ordinea de explorare a stărilor.

Una dintre strategii este alegerea unei ordini în funcție de variabilele implicate în problema dată. De exemplu, putem prioritiza variabilele care au un impact mai mare asupra soluției finale sau care au constrângeri mai stricte asociate cu ele. Aceasta poate contribui la reducerea factorului de ramificație și poate direcționa explorarea către soluții valide mai repede.

O altă strategie este alegerea unei ordini bazate pe evaluarea și prioritizarea opțiunilor de valori pentru variabile. În funcție de problema specifică, unele valori pot fi mai promițătoare decât altele în obținerea unei soluții valide. Prin prioritatea evaluării acestor valori mai promițătoare, putem evita explorarea unor ramuri care sunt mai puțin susceptibile de a duce la o soluție.

* + 1. **Euristicile de alegere a opțiunilor**

Euristicile de alegere a opțiunilor sunt strategii utilizate pentru a decide care opțiuni să fie evaluate mai întâi în cadrul algoritmului de backtracking. Aceste euristici pot ajuta la identificarea mai rapidă a soluției optime sau la evitarea explorării ramurilor care nu pot duce la o soluție viabilă.

Un exemplu comun de euristică este euristica "Minimum Remaining Values" (MRV), care implică alegerea variabilei cu cel mai mic număr de valori posibile rămase de atribuit. Alegerea unei astfel de variabile poate reduce numărul total de stări evaluate.

* + 1. **Tehnici de optimizare bazate pe informații specifice problemei**

Tehnicile de optimizare bazate pe informații specifice problemei reprezintă o abordare avansată în îmbunătățirea performanței algoritmului de backtracking. Aceste tehnici implică utilizarea cunoștințelor și constrângerilor specifice problemei pentru a ghida și optimiza procesul de explorare a spațiului de căutare.

Una dintre tehnicile utilizate este identificarea și aplicarea de constrângeri specifice problemei pentru a reduce spațiul de căutare. Prin înțelegerea restricțiilor stricte ale problemei și relațiilor între variabile, se pot elimina opțiunile care nu respectă aceste constrângeri, ceea ce duce la o reducere semnificativă a numărului de ramificații evaluate și, implicit, la o eficiență sporită a algoritmului de backtracking.

Pe lângă utilizarea constrângerilor, informațiile parțiale și restricțiile specifice problemei pot fi folosite pentru a ghida alegerea opțiunilor în cadrul algoritmului de backtracking. Aceste informații pot fi utilizate pentru a prioritiza anumite opțiuni sau pentru a evita explorarea unor ramuri care nu pot duce la o soluție validă. Prin utilizarea înțeleaptă a acestor informații, se poate obține o explorare mai eficientă și o identificare mai rapidă a soluțiilor optime.

* 1. **Tehnici de optimizare bazate pe informații specifice problemei**

Tăierea inutilă a căilor în algoritmul de backtracking, cunoscută și sub numele de pruning, este o tehnică utilizată pentru a reduce numărul de căi explorate într-un arbore de căutare, eliminând ramuri care nu pot conduce la o soluție validă.

Această tehnică se bazează pe faptul că anumite ramuri ale arborelui de căutare nu pot duce la o soluție validă, fie pentru că nu respectă anumite constrângeri ale problemei, fie pentru că nu pot satisface condițiile de soluționare. Prin eliminarea acestor ramuri, algoritmul poate evita explorarea unor căi care oricum nu ar duce la o soluție validă, economisind astfel timp și resurse.

Există diferite moduri de a aplica tăierea inutilă a căilor în algoritmul de backtracking, în funcție de problema specifică. De exemplu, în problema celor opt regine (unde se caută o aranjare a opt regine pe o tablă de șah astfel încât să nu se atace reciproc), se poate evita explorarea unor căi inutile prin verificarea constrângerilor pentru fiecare regină adăugată în soluție. Dacă o regină nu poate fi plasată într-o anumită poziție fără a ataca o altă regină deja plasată, atunci se poate elimina acea ramură din arborele de căutare.

Tăierea inutilă a căilor în algoritmul de backtracking este esențială în multe probleme combinatoriale complexe, unde numărul de căi posibile este foarte mare. Prin eliminarea căilor inutile, algoritmul poate ajunge mai rapid la o soluție validă sau poate determina că nu există nicio soluție posibilă, economisind astfel timp și resurse.

* + 1. Pruning prin verificarea condițiilor de soluție parțială.

Acest tip de pruning implică verificarea condițiilor de soluție parțială pentru a elimina căile care nu pot duce la o soluție validă. În timpul parcurgerii arborelui de căutare, algoritmul poate verifica dacă starea curentă a căii parțiale respectă anumite condiții necesare pentru a fi considerată o soluție parțială validă. Dacă condițiile nu sunt îndeplinite, se poate elimina acea ramură din arbore.

De exemplu, în problema colorării hartiei, în care trebuie să se atribuie culori diferite unor regiuni vecine, algoritmul poate verifica dacă colorarea parțială curentă respectă regula de a avea culori diferite pentru regiuni vecine. Dacă o regiune are deja o culoare atribuită, iar una dintre vecinele sale are aceeași culoare, atunci se poate elimina acea ramură din arbore.

* + 1. Pruning prin verificarea constrângerilor și regulelor problemei

Acest tip de pruning implică verificarea constrângerilor și regulelor specifice problemei pentru a elimina căile care nu pot satisface acestea. În multe probleme combinatoriale, există anumite constrângeri și regule care trebuie respectate pentru ca o soluție să fie validă.

De exemplu, în problema labirintului, algoritmul de backtracking poate verifica dacă o mișcare propusă într-o anumită direcție este validă, adică nu iese din limitele labirintului sau nu trece prin pereți. Dacă mișcarea propusă nu este validă conform regulelor problemei, atunci se poate elimina acea ramură din arbore.

* + 1. Pruning prin aplicarea limitelor superioare și inferioare

Acest tip de pruning implică utilizarea limitelor superioare și inferioare pentru a elimina căile care nu pot conduce la o soluție optimă sau care nu pot îndeplini anumite criterii de calitate a soluției.

De exemplu, în problema rucsacului, în care trebuie să se selecteze un set de obiecte cu greutatea totală maximă posibilă, algoritmul de backtracking poate utiliza o limită superioară a greutății pentru a evita explorarea căilor care nu pot duce la o greutate mai mare decât această limită. De asemenea, algoritmul poate utiliza o limită inferioară bazată pe o euristică pentru a tăia căile care nu pot duce la o soluție mai bună decât cea obținută până în acel moment.

Pruning prin aplicarea limitelor superioare și inferioare poate contribui la accelerarea algoritmului și la obținerea unei soluții mai eficiente din punct de vedere al timpului și resurselor.

1. **APLICATII PRACTICE**

Backtracking-ul are aplicații practice într-o varietate de domenii și poate fi folosit în rezolvarea unor probleme complexe. Algoritmul poate explora recursiv toate posibilitățile și încerca să găsească soluții valide prin aplicarea unor reguli și constrângeri specifice fiecărei probleme.

* 1. **Algoritmul backtracking-ului.**

Backtrack(x)

if x is not a solution

return false

if x is a new solution

add to list of solutions

backtrack(expand x)

1. Funcția Backtrack(x) primește un parametru x, care reprezintă starea curentă a algoritmului de backtracking.
2. Dacă starea x nu este o soluție validă, algoritmul se oprește și returnează false. Acest lucru înseamnă că nu există nicio soluție posibilă în continuarea explorării și se face un pas înapoi în căutare de alte opțiuni.
3. Dacă starea x este o soluție nouă și validă, aceasta este adăugată la lista de soluții. Acest pas este optional și poate fi adaptat în funcție de necesități. Poate fi adăugat un alt tip de procesare sau salvare a soluției, în funcție de problema specifică.
4. Se continuă explorarea spațiului de căutare prin expandarea stării x. Acest lucru presupune generarea și evaluarea tuturor posibilelor opțiuni următoare. Algoritmul se bazează pe recursivitate pentru a explora toate opțiunile posibile, apelând funcția Backtrack pe fiecare opțiune în parte.
5. Algoritmul de backtracking va continua recursivitatea până când toate soluțiile posibile au fost identificate și procesate.
   1. **Implementarea backtracking-ului în limbajul de programare C**

void backtracking(params...) {

if (soluție este completă) {

return;

}

for (toate opțiunile posibile) {

if (opțiunea este validă) {

backtracking(params...);

}

}

}

void solveBacktracking(params...) {

backtracking(params...);

}

1. Funcția backtracking este definită cu un set de parametri (params...). Acest set de parametri poate include informații despre starea curentă a căii parțiale și alte informații necesare pentru rezolvarea problemei.
2. Prima verificare este dacă soluția este completă. Acest lucru se realizează prin verificarea unei condiții care indică că toate cerințele problemei au fost îndeplinite. Dacă condiția este adevărată, se poate procesa soluția găsită și se returnează din funcție.
3. Urmează o buclă for care parcurge toate opțiunile posibile pentru a extinde soluția curentă. Aceste opțiuni pot fi generate sau selectate în funcție de cerințele problemei.
4. În interiorul buclei, se verifică dacă opțiunea curentă este validă, adică respectă constrângerile și regulele problemei. Dacă opțiunea nu este validă, se trece la următoarea iterație a buclei.
5. Dacă opțiunea este validă, starea curentă este actualizată pentru a reflecta adăugarea opțiunii la soluția parțială.
6. Apoi, se face un apel recursiv la funcția backtracking, cu aceleași parametri (params...) sau cu parametri actualizați, pentru a explora opțiunea următoare.
7. După apelul recursiv, se anulează modificările la starea curentă, pentru a pregăti următoarea iterație a buclei și explorarea altor opțiuni.
8. Funcția solveBacktracking este definită pentru a inițializa starea inițială și pentru a face apelul inițial al funcției backtracking cu parametrii corespunzători.
   1. **Probleme de optimizare**

Problemele de optimizare pot fi abordate și cu ajutorul algoritmului de backtracking, deși acesta nu este întotdeauna cel mai eficient algoritm pentru rezolvarea lor. Cu toate acestea, backtracking-ul poate fi util în anumite scenarii specifice.

Este important de menționat că, în multe cazuri, backtracking-ul poate fi ineficient pentru rezolvarea problemelor de optimizare, mai ales în cazul problemelor de dimensiuni mari. În astfel de situații, pot fi preferate alte tehnici de optimizare, cum ar fi programarea dinamică sau algoritmi evolutivi.

* + 1. **Problema rucsacului**

Problema rucsacului reprezintă un exemplu clasic de problemă de optimizare combinatorială în care trebuie să decidem ce obiecte să selectăm pentru a maximiza valoarea totală, ținând cont de o capacitate maximă a rucsacului.



**Figura 2.3.1.1** Problema rucsacului.

Algoritmul de rezolvare a problemei Rucsacului folosind backtracking constă în parcurgerea recursivă a tuturor posibilităților de selectare și ne-selectare a obiectelor, ținând cont de capacitatea maximă a rucsacului și obiectele disponibile.

1. #define MAX\_N 100
2. #define MAX\_WEIGHT 1000
3. int max\_value = 0;
4. int best\_solution[MAX\_N];
5. int weights[MAX\_N];
6. int values[MAX\_N];
7. Inițializarea variabilelor: La începutul programului, se initializează variabilele ‘max\_value’ și ‘best\_solution’ cu valorile inițiale corespunzătoare.
8. int main() {
9. int n;
10. int capacity;
11. printf("Introduceți numărul de obiecte: ");
12. scanf("%d", &n);
13. printf("Introduceți capacitatea rucsacului: ");
14. scanf("%d", &capacity);
15. printf("Introduceți greutățile obiectelor:\n");
16. for (int i = 0; i < n; i++) {
17. scanf("%d", &weights[i]);
18. }
19. printf("Introduceți valorile obiectelor:\n");
20. for (int i = 0; i < n; i++) {
21. scanf("%d", &values[i]);
22. }
23. Citirea datelor de intrare: Programul citește numărul de obiecte ‘(n)’, capacitatea rucsacului ‘(capacity)’, greutățile obiectelor ‘(weights)’ și valorile obiectelor ‘(values)’ de la utilizator.
24. void backtrack(int current\_item, int current\_weight, int current\_value, int n, int capacity) {
25. if (current\_weight > capacity) {
26. return;
27. }
28. if (current\_value > max\_value) {
29. max\_value = current\_value;
30. for (int i = 0; i < n; i++) {
31. best\_solution[i] = (current\_item >> i) & 1;
32. }
33. }
34. if (current\_item == n) {
35. return;
36. }
37. backtrack(current\_item + 1, current\_weight + weights[current\_item], current\_value + values[current\_item], n, capacity);
38. backtrack(current\_item + 1, current\_weight, current\_value, n, capacity);}
39. Funcția backtrack: Această funcție implementează algoritmul de backtracking. Ea primește ca argumente starea curentă a algoritmului: ‘current\_item’ (obiectul curent), ‘current\_weight’ (greutatea curentă a obiectelor selectate), ‘current\_value’ (valoarea curentă a obiectelor selectate), n (numărul total de obiecte) și ‘capacity’ (capacitatea rucsacului).
40. Verificarea condițiilor de oprire:

* Dacă greutatea obiectelor selectate (‘current\_weight’) depășește capacitatea rucsacului, se returnează din funcție deoarece aceasta nu reprezintă o soluție validă.
* Dacă valoarea obiectelor selectate (‘current\_value’) este mai mare decât ‘max\_value’, se actualizează ‘max\_value’ cu valoarea curentă și se salvează configurația curentă a obiectelor în ‘best\_solution’.

1. Explorarea spațiului de căutare:

* Dacă s-au explorat toate obiectele (current\_item este egal cu n), nu mai există opțiuni de selectare și se returnează din funcție.
* Altfel, se efectuează două apeluri recursive ale funcției backtrack:
  + Un apel pentru a lua obiectul curent și a continua explorarea cu obiectul următor (‘current\_item’ + 1), greutatea și valoarea actualizate (‘current\_weight’ + ‘weights[current\_item]’, ‘current\_value + values[current\_item]’).
  + Un apel pentru a nu lua obiectul curent și a continua explorarea cu obiectul următor, fără modificări ale greutății și valorii (‘current\_item + 1’, ‘current\_weight’, ‘current\_value’).

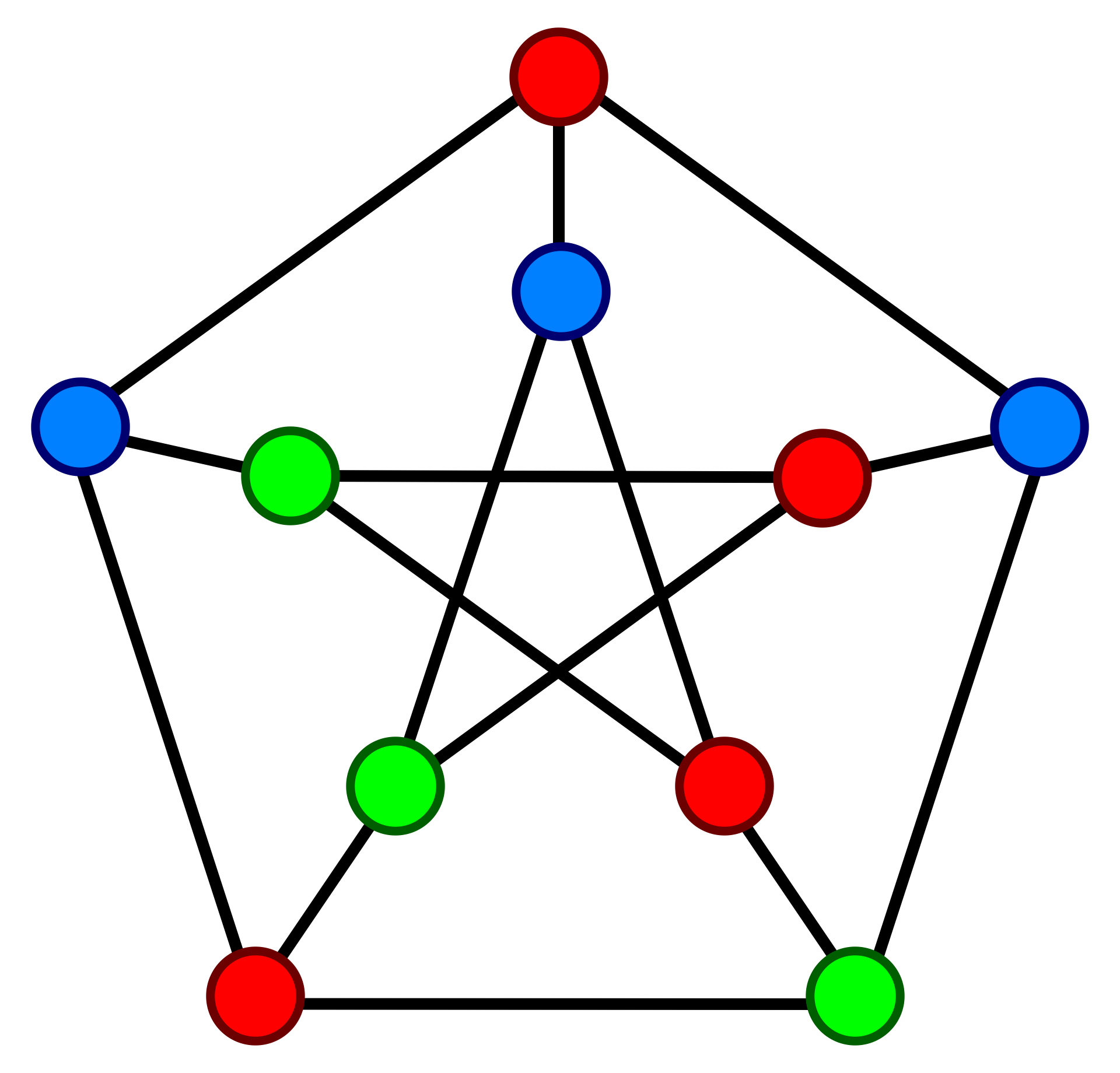
1. printf("Soluția optimă este:\n");
2. for (int i = 0; i < n; i++) {
3. if (best\_solution[i]) {
4. printf("Obiectul %d\n", i + 1);
5. }
6. }
7. printf("Valoarea maximă obținută: %d\n", max\_value);
8. return 0;}
9. Afișarea rezultatelor: După finalizarea explorării spațiului de căutare, programul afișează soluția optimă ‘(best\_solution)’ prin parcurgerea vectorului și afișarea obiectelor selectate. De asemenea, afișează și valoarea maximă obținută ‘(max\_value)’.

Metoda backtracking explorează recursiv toate posibilitățile de selectare și ne-selectare a obiectelor pentru a găsi configurația optimă care maximizează valoarea totală a obiectelor selectate, respectând limita de capacitate a rucsacului.

Avantajul principal al metodei backtracking este că poate genera soluții valide pentru diverse configurații de obiecte și capacitatea rucsacului. Ea garantează găsirea soluției optime, dar poate necesita un timp mai îndelungat de execuție în cazul unui spațiu de căutare mare.

* + 1. **Problema colorării grafului**

Problema colorării grafului este o problemă clasică în teoria grafurilor, care constă în găsirea unei colorări a nodurilor unui graf, astfel încât nici două noduri adiacente să nu aibă aceeași culoare. Aceasta este o problemă NP-dificilă și nu există o soluție cunoscută care să poată rezolva această problemă pentru orice graf.



**Figura 2.3.2.1** Colorarea corectă a nodurilor a graficului Petersen cu 3 culori.

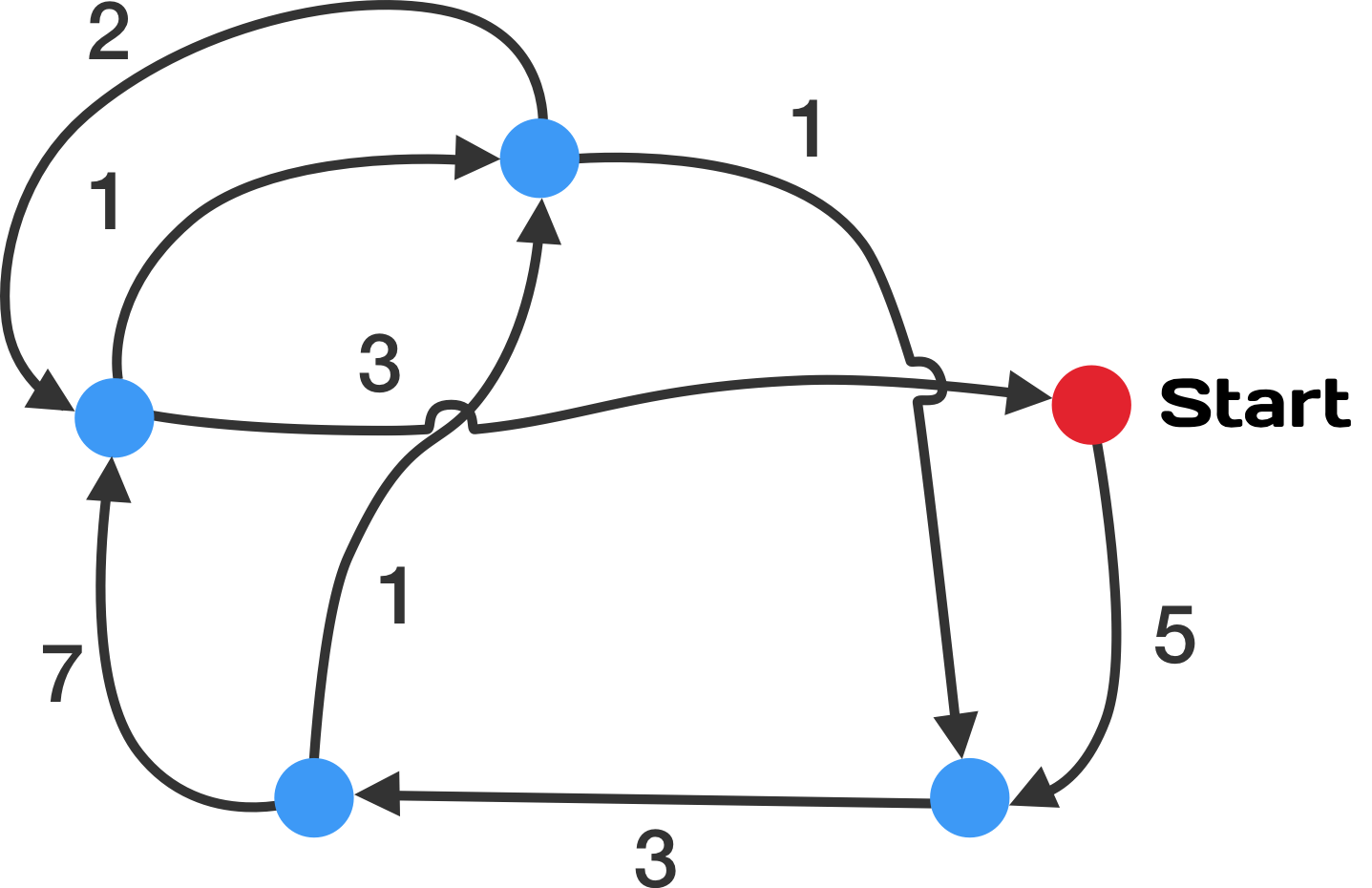
* + - 1. #define MAX\_NODES 100
      2. int graph[MAX\_NODES][MAX\_NODES];
      3. int colors[MAX\_NODES];
      4. int numNodes;

1. Se declareaza matricea ‘graph’ pentru a reprezenta graful și vectorul ‘colors’ pentru a stoca culoarea atribuită fiecărui nod si numarul total de noduri ‘numNodes’.
2. int isSafe(int node, int color, int colorAssignment[]) {
3. for (int i = 0; i < numNodes; i++) {
4. if (graph[node][i] && colorAssignment[i] == color) {
5. return 0;
6. }
7. }
8. return 1;
9. }
10. Funcția ‘isSafe’ este folosită pentru a verifica dacă se poate atribui o anumită culoare unui nod fără să existe conflicte cu vecinii săi. Aceasta parcurge toți vecinii nodului și verifică dacă au aceeași culoare.
11. int graphColoringUtil(int numColors, int colorAssignment[], int node) {
12. if (node == numNodes) {
13. return 1;
14. }
15. for (int color = 1; color <= numColors; color++) {
16. if (isSafe(node, color, colorAssignment)) {
17. colorAssignment[node] = color;
18. if (graphColoringUtil(numColors, colorAssignment, node + 1) == 1) {
19. return 1;
20. }
21. colorAssignment[node] = 0;
22. }
23. }
24. return 0;
25. }
26. Funcția ‘graphColoringUtil’ este funcția principală de backtracking. Aceasta primește numărul maxim de culori, vectorul ‘colorAssignment’, și nodul curent. Verifică cazul de oprire: dacă toate nodurile au fost colorate, returnează 1 pentru a indica găsirea unei soluții valide. În caz contrar, încercă toate culorile pentru nodul curent. Pentru fiecare culoare, verifică dacă poate fi atribuită nodului și apoi apelează recursiv ‘graphColoringUtil’ pentru a colora nodurile rămase. Dacă găsește o soluție validă, returnează 1. Dacă nici o colorare nu duce la o soluție validă, resetează culoarea nodului curent și continuă explorarea cu alte culori.
27. void graphColoring(int numColors) {
28. int colorAssignment[MAX\_NODES];
29. for (int i = 0; i < numNodes; i++) {
30. colorAssignment[i] = 0;
31. }
32. if (graphColoringUtil(numColors, colorAssignment, 0) == 0) {
33. printf("Nu există o soluție de colorare cu %d culori.\n", numColors);
34. } else {
35. printf("Soluția de colorare cu %d culori este:\n", numColors);
36. for (int i = 0; i < numNodes; i++) {
37. printf("Nodul %d: Culoare %d\n", i, colorAssignment[i]);
38. }
39. }
40. }
41. Funcția ‘graphColoring’ inițializează vectorul ‘colorAssignment’ cu 0 și apelează ‘graphColoringUtil’ pentru nodul de start 0. Dacă ‘graphColoringUtil’ returnează 0, se afișează un mesaj că nu există o soluție de colorare. În caz contrar, se afișează soluția de colorare pentru fiecare nod.
42. int main() {
43. int numEdges;
44. int node1, node2;
45. printf("Introduceți numărul de noduri în graf: ");
46. scanf("%d", &numNodes);
47. printf("Introduceți numărul de muchii în graf: ");
48. scanf("%d", &numEdges);
49. printf("Introduceți muchiile (perechea de noduri):\n");
50. for (int i = 0; i < numEdges; i++) {
51. scanf("%d %d", &node1, &node2);
52. graph[node1][node2] = 1;
53. graph[node2][node1] = 1;
54. }
55. int numColors;
56. printf("Introduceți numărul maxim de culori: ");
57. scanf("%d", &numColors);
58. graphColoring(numColors);
59. return 0;
60. }
61. Funcția ‘main’ primește input-ul utilizatorului pentru numărul de noduri și muchii în graf. Apoi, se citește perechea de noduri pentru fiecare muchie și se construiește graful în matricea ‘graph’. Se cere utilizatorului să introducă numărul maxim de culori și se apelează funcția ‘graphColoring’ pentru a rezolva problema de colorare a grafului.

Astfel, algoritmul de backtracking este evidentiat în programul prin funcțiile graphColoringUtil și graphColoring. Acestea explorează recursiv toate posibilitățile de colorare a nodurilor și utilizează tehnicile de backtracking pentru a găsi o soluție validă.

* + 1. **Problema TSP (Travelling Salesman Problem)**

Problema TSP (Travelling Salesman Problem) este o problemă clasică de optimizare combinatorială în care se dorește determinarea celui mai scurt circuit hamiltonian într-un graf complet ponderat. Circuitul hamiltonian reprezintă un ciclu în graful în care fiecare nod este vizitat o singură dată, iar costul total al muchiilor care alcătuiesc circuitul trebuie să fie minim.



**Figura 2.3.3.1** Rețeaua de orașe

Algoritmul de backtracking poate fi utilizat pentru a rezolva problema TSP, dar din cauza complexității ridicate a problemei, aceasta devine impractică pentru instanțe mari. Cu toate acestea, este utilă în înțelegerea conceptelor și abordărilor de rezolvare.

1. #define MAX 10
2. int graph[MAX][MAX]; // Matricea de adiacență pentru graf
3. int visited[MAX]; // Vectorul de noduri vizitate
4. int n; // Numărul de noduri în graf
5. int costMinim = INT\_MAX; // Costul minim găsit până în present
6. Inițializarea variabilelor: Programul începe prin inițializarea matricei de adiacență ‘graph’ a grafului, vectorului de noduri vizitate ‘visited’, numărului de noduri ‘n’ și costului minim găsit până în present ‘costMinim’.
7. void tsp(int current, int count, int cost, int path[]) {
8. int i;
9. visited[current] = 1; // Se marchează nodul curent ca fiind vizitat
10. path[count] = current; // Se adaugă nodul în calea curentă
11. // Cazul de oprire: toate nodurile au fost vizitate
12. if (count == n - 1) {
13. // Se adaugă costul muchiei de la ultimul nod la primul nod
14. cost += graph[current][0];
15. // Verificăm dacă este o soluție mai bună decât minimul curent
16. if (cost < costMinim) {
17. costMinim = cost;
18. printf("Cost minim: %d\n", costMinim);
20. // Se afișează calea minimă
21. printf("Cale minimă: ");
22. for (i = 0; i < n; i++) {
23. printf("%d ", path[i]);
24. }
25. printf("0\n"); // Adăugăm nodul de start la final
26. }
27. }
28. else {
29. // Se continuă explorarea pentru toate nodurile nevizitate
30. for (i = 1; i < n; i++) {
31. if (!visited[i]) {
32. tsp(i, count + 1, cost + graph[current][i], path);
33. }
34. }
35. }
36. // Se marchează nodul curent ca nevizitat pentru a putea fi explorat din nou în alte ramuri
37. visited[current] = 0;
38. }
39. Funcția ‘tsp’ primește ca parametri nodul curent, numărul de noduri vizitate până în prezent, costul curent și calea parcursă până în acel moment.
40. Se marchează nodul curent ca fiind vizitat și se adaugă în calea curentă.
41. Cazul de oprire: dacă s-au vizitat toate nodurile, se adaugă costul muchiei de la ultimul nod la primul nod și se verifică dacă acesta este mai mic decât costul minim găsit până în prezent. Dacă da, se actualizează costul minim și se afișează calea minimă.
42. În caz contrar, se continuă explorarea pentru toate nodurile nevizitate, prin apelul recursiv al funcției tsp pentru fiecare nod nevizitat.
43. După terminarea explorării unei ramuri, se marchează nodul curent ca nevizitat pentru a putea fi explorat din nou în alte ramuri.
44. int main() {
45. int i, j;
46. printf("Introduceți numărul de noduri în graf: ");
47. scanf("%d", &n);
49. printf("Introduceți matricea de adiacență pentru graf:\n");
50. for (i = 0; i < n; i++) {
51. for (j = 0; j < n; j++) {
52. scanf("%d", &graph[i][j]);
53. }
54. }
55. În funcția ‘main’, se primește numărul de noduri în graf și matricea de adiacență de la utilizator.
56. int path[MAX];
57. // Se initializează vectorul de noduri vizitate cu 0 (nevizitate)
58. for (i = 0; i < n; i++) {
59. visited[i] = 0;
60. }
61. // Se pornește algoritmul de backtracking de la nodul de start 0
62. tsp(0, 0, 0, path);
63. return 0;
64. }
65. Se inițializează vectorul de noduri vizitate cu 0 (nevizitate).
66. Se pornește algoritmul de backtracking apelând funcția ‘tsp’ cu nodul de start 0 și valorile inițiale pentru numărul de noduri vizitate, costul și calea.
67. La final, se afișează rezultatele, inclusiv costul minim și calea minimă găsită.

În acest algoritm, backtracking-ul este realizat prin utilizarea recursivității. La fiecare pas, se alege un nod nevizitat și se explorează toate posibilitățile de continuare a călătoriei. Dacă se ajunge la o soluție validă (toate nodurile sunt vizitate), se verifică dacă costul obținut este mai mic decât costul minim găsit până în prezent și se actualizează rezultatele. Dacă nu se ajunge la o soluție validă, se continuă explorarea în profunzime, alegând un alt nod nevizitat. Prin marcarea și debifarea nodurilor vizitate, se asigură că fiecare nod este vizitat o singură dată și că se explorează toate posibilitățile de călătorie posibile.

Este important de menționat că aceasta este doar o implementare simplificată și nu include toate optimizările sau heuristici posibile pentru o rezolvare eficientă a problemei.

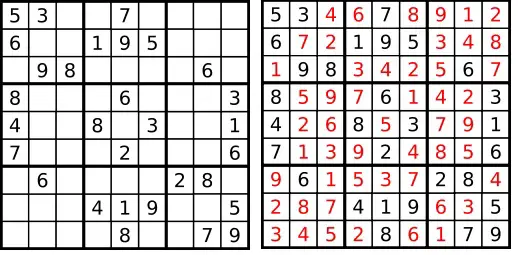
* 1. **Jocuri**

Backtracking-ul are o aplicabilitate semnificativă în rezolvarea jocurilor. Jocurile pot fi modelate ca probleme de decizie și pot fi rezolvate folosind algoritmi de backtracking pentru a explora toate posibilitățile și a găsi soluții valide.

* + 1. **Sudoku**

Sudoku este un joc de logică care poate fi rezolvat eficient cu ajutorul algoritmului de backtracking. În rezolvarea unui puzzle Sudoku, obiectivul este de a completa o tablă 9x9 împărțită în subgrile de 3x3, astfel încât fiecare rând, fiecare coloană și fiecare subgrilă să conțină toate cifrele de la 1 la 9, fără repetiții.

Algoritmul de backtracking este ideal pentru rezolvarea problemelor de tipul Sudoku datorită naturii sale exploratorii și posibilității de a reveni în urmă pentru a încerca alte combinații.



**Figura 2.4.1.1** Sudoku solver utilizand backtracking-ul

1. #include <stdio.h>
2. #include <stdbool.h>
3. #define SIZE 9
4. int main() {
5. int grid[SIZE][SIZE] = {
6. {
7. {5, 3, 0, 0, 7, 0, 0, 0, 0},
8. {6, 0, 0, 1, 9, 5, 0, 0, 0},
9. {0, 9, 8, 0, 0, 0, 0, 6, 0},
10. {8, 0, 0, 0, 6, 0, 0, 0, 3},
11. {4, 0, 0, 8, 0, 3, 0, 0, 1},
12. {7, 0, 0, 0, 2, 0, 0, 0, 6},
13. {0, 6, 0, 0, 0, 0, 2, 8, 0},
14. {0, 0, 0, 4, 1, 9, 0, 0, 5},
15. {0, 0, 0, 0, 8, 0, 0, 7, 9}
16. };
17. printf("Sudoku initial:\n");
18. printSudoku(grid);
19. if (solveSudoku(grid)) {
20. printf("\nSudoku rezolvat:\n");
21. printSudoku(grid);
22. } else {
23. printf("\nNu exista o solutie valida pentru Sudoku!\n");
24. }
25. Se definește tabla Sudoku inițială, reprezentată de o matrice bidimensională de dimensiune 9x9. Valorile inițiale ale celulelor sunt reprezentate prin numere de la 1 la 9, iar valorile necunoscute sunt reprezentate prin zero.
26. bool solveSudoku(int grid[SIZE][SIZE]) {
27. int row, col;
28. if (!findEmptyCell(grid, &row, &col)) {
29. // Nu mai sunt celule goale, tabla este completă
30. return true;
31. }
32. // Testează valorile de la 1 la 9 pentru celula curentă
33. for (int num = 1; num <= SIZE; num++) {
34. if (isSafe(grid, row, col, num)) {
35. grid[row][col] = num;
36. if (solveSudoku(grid)) {
37. // Tabla este completă, s-a găsit o soluție
38. return true;
39. }
40. // Soluția nu este validă, se revine la valoarea anterioară
41. grid[row][col] = 0;
42. }
43. }
44. // Nu există o soluție validă pentru tabla Sudoku
45. return false;
46. }
47. Funcția solveSudoku() primește tabla Sudoku ca parametru și implementează algoritmul de backtracking pentru rezolvarea problemei.
48. În interiorul funcției solveSudoku(), se utilizează o structură de control ‘for’ pentru a itera prin fiecare celulă a tablei. Pentru fiecare celulă, se verifică dacă valoarea este zero, ceea ce indică o celulă goală.
49. Dacă se găsește o celulă goală, se încearcă valorile de la 1 la 9 în interiorul unui alt ciclu for. Pentru fiecare valoare, se verifică dacă aceasta poate fi plasată în celula curentă fără a încălca regulile Sudoku. Acest lucru se realizează prin apelul funcției ‘isSafe()’.
50. bool isSafe(int grid[SIZE][SIZE], int row, int col, int num) {
51. // Verifică dacă numărul se regăsește deja în aceeași linie
52. for (int i = 0; i < SIZE; i++) {
53. if (grid[row][i] == num) {
54. return false;
55. }
56. }
57. // Verifică dacă numărul se regăsește deja în aceeași coloană
58. for (int i = 0; i < SIZE; i++) {
59. if (grid[i][col] == num) {
60. return false;
61. }
62. }
63. // Verifică dacă numărul se regăsește deja în aceeași sub-grilă de 3x3
64. int startRow = row - (row % 3);
65. int startCol = col - (col % 3);
66. for (int i = 0; i < 3; i++) {
67. for (int j = 0; j < 3; j++) {
68. if (grid[i + startRow][j + startCol] == num) {
69. return false;
70. }
71. }
72. }
73. return true;
74. }
75. Funcția isSafe() primește tabla Sudoku, rândul și coloana celulei curente, precum și valoarea pe care dorim să o plasăm. Verifică dacă valoarea respectivă respectă regulile Sudoku pentru rândul, coloana și subgrila asociate celulei curente. Dacă valoarea poate fi plasată în celula curentă fără a încălca regulile, funcția returnează true. În caz contrar, returnează false.
76. În cazul în care o valoare poate fi plasată în celula curentă, se atribuie acea valoare în tabla Sudoku și se face o apelare recursivă a funcției solveSudoku() pentru a completa restul tablei.
77. Apelul recursiv va explora recursiv toate posibilitățile de completare a celulelor goale rămase. Dacă se găsește o soluție validă, funcția solveSudoku() returnează true. În caz contrar, se revine la valoarea anterioară a celulei curente și se continuă explorarea cu o altă valoare posibilă.
78. Dacă se parcurge toată tabla Sudoku și nu se găsește o soluție validă, funcția solveSudoku() returnează false, semnalând că nu există o soluție.
79. În funcția main(), se definește tabla Sudoku inițială și se apelează funcția solveSudoku() pentru a rezolva problema. Dacă se găsește o soluție validă, tabla Sudoku rezolvată este afișată. În caz contrar, se afișează un mesaj de eroare.
80. bool findEmptyCell(int grid[SIZE][SIZE], int \*row, int \*col) {
81. for (\*row = 0; \*row < SIZE; (\*row)++) {
82. for (\*col = 0; \*col < SIZE; (\*col)++) {
83. if (grid[\*row][\*col] == 0) {
84. return true;
85. }
86. }
87. }
88. return false;
89. }
90. Funcția primește tabla Sudoku și coordonatele rândului și coloanei unei celule ca parametri.
91. În interiorul funcției, se verifică valoarea din celula corespunzătoare rândului și coloanei primite ca parametri. Dacă valoarea este zero, înseamnă că celula este goală, deci funcția returnează true.
92. Dacă valoarea este diferită de zero, înseamnă că celula are deja o valoare atribuită și nu este goală, deci funcția returnează false.
93. Funcția isEmptyCell() este utilizată în cadrul algoritmului de rezolvare a Sudoku pentru a verifica dacă o celulă este goală înainte de a încerca să-i atribuim o valoare.

Prin utilizarea recursivității, algoritmul explorează în mod recursiv toate posibilitățile și backtracking-ul permite revenirea la starea anterioară pentru a încerca alte combinații de valori. Astfel, se continuă procesul până când se găsește o soluție validă sau se parcurge întreaga tabla Sudoku fără a găsi o soluție.

* + 1. **Jocul 8-regine**

Jocul 8-regine este o problemă clasică de colocare a reginelor pe o tablă de șah de dimensiune 8x8, astfel încât niciuna dintre ele să nu se atace reciproc. Fiecare regină trebuie plasată pe o coloană și o linie astfel încât să nu existe nicio alta regină pe aceeași linie, coloană sau diagonală.

Pentru a rezolva problema jocului 8-regine, putem utiliza algoritmul de backtracking. Algoritmul de backtracking explorează recursiv toate posibilitățile de plasare a reginelor, începând cu prima coloană și avansând către următoarele coloane.

1. #define N 8
2. void printSolution(int board[N][N]) {
3. for (int i = 0; i < N; i++) {
4. for (int j = 0; j < N; j++) {
5. printf("%d ", board[i][j]);
6. }
   1. printf("\n");
7. }
8. printf("\n");
9. }
10. La început, definim dimensiunea tablei de șah ‘N’ și declarăm funcțiile ‘printSolution()’, ‘isSafe()’ și ‘solveNQueens()’.
11. Funcția ‘printSolution()’ afișează o soluție validă găsită pentru jocul 8-regine. Parcurgem tabla de șah și afișăm valorile din fiecare celulă.
12. int isSafe(int board[N][N], int row, int col) {
13. // Verificăm rândul în stânga
14. for (int i = 0; i < col; i++) {
15. if (board[row][i])
16. return 0;
17. }
18. // Verificăm diagonală superioară-stânga
19. for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
20. if (board[i][j])
21. return 0;
22. }
23. // Verificăm diagonală inferioară-stânga
24. for (int i = row, j = col; i < N && j >= 0; i++, j--) {
25. if (board[i][j])
26. return 0;
27. }
28. return 1;}
29. Funcția isSafe() verifică dacă o regină poate fi plasată într-o anumită poziție fără să se atace cu celelalte regine deja plasate. Aceasta parcurge rândul în stânga și verifică dacă există vreo regină deja plasată acolo. Apoi, verifică cele două diagonale (superioară-stânga și inferioară-stânga) pentru a detecta prezența altor regine.
30. int solveNQueens(int board[N][N], int col) {
31. if (col >= N) {
32. printSolution(board);
33. return 1;
34. }
35. int res = 0;
36. for (int i = 0; i < N; i++) {
37. if (isSafe(board, i, col)) {
38. board[i][col] = 1;
39. res += solveNQueens(board, col + 1);
40. board[i][col] = 0; // backtracking
41. }
42. }
43. return res;
44. }
45. Funcția principală solveNQueens() primește tabela de șah board și coloana curentă col ca parametri. Dacă coloana curentă depășește dimensiunea tablei de șah, înseamnă că am găsit o soluție completă și apelăm funcția printSolution() pentru a o afișa.
46. În caz contrar, parcurgem rândurile de la 0 la N-1 și verificăm dacă regina poate fi plasată în acea poziție folosind funcția isSafe(). Dacă regina poate fi plasată acolo, setăm valoarea 1 în tabela de șah pentru acea poziție și apelăm recursiv solveNQueens() pentru a găsi soluțiile în continuare.
47. După apelul recursiv, revenim și resetăm valoarea la 0 în tabela de șah pentru a permite explorarea altor posibilități (backtracking). Acest lucru ne permite să explorăm toate combinațiile posibile.
48. În final, funcția solveNQueens() returnează numărul total de soluții găsite. Aceasta este suma tuturor soluțiilor găsite în toate ramurile recursivității.
49. int main() {
50. int board[N][N] = {0};
52. int solutions = solveNQueens(board, 0);
53. printf("Numărul total de soluții: %d\n", solutions);
55. return 0;
56. }
57. În funcția principală main(), inițializăm tabela de șah board cu valorile inițiale (0 pentru celule goale).
58. Apelăm funcția solveNQueens() cu tabela de șah și coloana de start 0. Rezultatul returnat este numărul total de soluții găsite.
59. Afișăm rezultatul final, adică numărul total de soluții.

Acest cod implementează algoritmul de rezolvare a jocului 8-regine folosind backtracking. Funcția solveNQueens() utilizează recursivitate pentru a explora toate posibilitățile de plasare a reginelor pe tabla de șah. Funcția isSafe() este utilizată pentru a verifica dacă o regină poate fi plasată într-o anumită poziție fără să se atace cu celelalte regine deja plasate.

* 1. **Analiza de date**

În cadrul analizei de date, backtracking-ul poate fi utilizat pentru a explora diferite combinații, permutări sau submulțimi ale datelor în vederea analizei și evaluării acestora.

Un exemplu de utilizare a backtracking-ului în analiza de date este determinarea tuturor combinațiilor posibile ale unui set de valori pentru a găsi cea mai bună combinație care satisface anumite criterii sau restricții. Acest lucru poate fi util în domenii precum optimizarea problemelor de programare, probleme de planificare, planificarea resurselor și altele.

* + 1. **Selecția de date dintr-o bază de date**

Selecția de date dintr-o bază de date este unul dintre aspectele fundamentale în analiza de date. Backtracking-ul poate fi utilizat pentru a explora diferite combinații de filtre și criterii de selecție în vederea extragerii datelor relevante dintr-o bază de date.

Pentru a utiliza backtracking-ul în selecția de date dintr-o bază de date, urmăm următorii pași:

1. Definirea criteriilor de selecție: Stabilim criteriile și filtrele necesare pentru a selecta datele dorite din baza de date. Acestea pot include condiții logice, restricții pe anumite coloane sau alte criterii specifice.
2. Definirea spațiului de căutare: Identificăm parametrii relevanți și setul de date din care vom extrage informațiile. Aceasta poate include numele tabelului sau tabelurilor, coloanele implicate și orice restricții sau condiții suplimentare.
3. Implementarea funcției de backtracking: Definim o funcție de backtracking care explorează spațiul de căutare și generează combinații de filtre pentru selecția de date. În fiecare pas, verificăm dacă combinația curentă satisface criteriile de selecție și, în funcție de rezultat, continuăm să explorăm sau să abandonăm ramura curentă.
4. Interogarea bazei de date: În fiecare pas al procesului de backtracking, construim interogări SQL corespunzătoare pentru a selecta datele din baza de date conform combinației de filtre generate. Aceste interogări pot fi construite dinamic în funcție de filtrele activate sau dezactivate în combinația curentă.
5. Extragerea și manipularea datelor: Odată ce am obținut rezultatele interogărilor, putem extrage și manipula datele în funcție de nevoile și scopurile noastre. Acest lucru poate include analiza și agregarea datelor, calcularea unor statistici relevante sau generarea de rapoarte și vizualizări.
6. Terminarea și afișarea rezultatelor: Procesul de backtracking continuă până când toate combinațiile de filtre au fost explorate sau am obținut rezultatele dorite. La final, afișăm rezultatele selecției de date efectuate, cum ar fi rândurile și coloanele selectate, statistici sau alte informații relevante.

Prin utilizarea backtracking-ului în selecția de date dintr-o bază de date, putem explora eficient combinațiile de filtre și criterii de selecție pentru a găsi seturile de date relevante pentru analiza și procesarea ulterioară.

* + 1. **Identificarea de pattern-uri in date**

Identificarea de pattern-uri în date este un aspect important al analizei de date, care poate fi realizată cu ajutorul metodei backtracking. Această metodă ne permite să explorăm diferite combinații de caracteristici sau variabile în setul de date pentru a descoperi modele sau tendințe semnificative.

Pentru a utiliza backtracking-ul în identificarea de pattern-uri în date, urmăm următorii pași:

1. Definirea problemelor de identificare a pattern-urilor: Stabilim ce tip de pattern-uri dorim să identificăm în datele noastre. Acestea pot fi modele de comportament, tendințe, asocieri sau orice alt tip de relații între variabilele din setul de date.
2. Definirea spațiului de căutare: Identificăm variabilele relevante și setul de date pe care vom aplica backtracking-ul. Aceasta poate include selectarea anumitor coloane sau combinații de variabile pe care dorim să le investigăm în căutarea pattern-urilor.
3. Implementarea funcției de backtracking: Definim o funcție de backtracking care explorează spațiul de căutare și generează combinații de variabile pentru analiza pattern-urilor. În fiecare pas, verificăm dacă combinația curentă prezintă un anumit pattern sau îndeplinește anumite criterii de pattern identificate anterior.
4. Analiza datelor și identificarea pattern-urilor: În fiecare pas al procesului de backtracking, analizăm combinația curentă de variabile și efectuăm calcule sau verificări pentru a identifica pattern-uri relevante. Acest lucru poate include compararea valorilor, calcularea de statistici sau aplicarea unor metode specifice de identificare a pattern-urilor.
5. Extrageți și interpretați pattern-urile identificate: Odată ce am obținut pattern-urile din date, trebuie să le extragem și să le interpretăm în contextul problemei noastre. Acest lucru poate implica identificarea tendințelor, asocierilor sau comportamentului specific în setul de date și extragerea de concluzii sau luarea de decizii pe baza acestor pattern-uri identificate.
6. Terminarea și raportarea rezultatelor: Procesul de backtracking continuă până când toate combinațiile de variabile au fost explorate sau am identificat pattern-urile dorite. La final, raportăm rezultatele identificării de pattern-uri, cum ar fi descrierea pattern-urilor identificate, frecvența lor sau alte informații relevante.

Utilizarea backtracking-ului în identificarea de pattern-uri în date ne permite să explorăm eficient combinațiile de variabile pentru a descoperi relații și tendințe semnificative în datele noastre. Aceasta ne ajută să obținem o înțelegere mai profundă a datelor și să luăm decizii informate în funcție de pattern-urile identificate.

Pentru a exemplifica identificarea de pattern-uri în date dintr-un fișier text în limbajul de programare C, vom lua în considerare un scenariu simplificat. Presupunem că avem un fișier text numit "data.txt" care conține datele noastre și dorim să identificăm pattern-uri specifice în aceste date.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. void identifyPatterns(const char\* filename) {
5. FILE\* file = fopen(filename, "r");
6. if (file == NULL) {
7. printf("Eroare la deschiderea fisierului.");
8. return;
9. }
10. În funcția ‘identifyPatterns’, se deschide fișierul specificat prin parametrul filename utilizând funcția fopen. Dacă fișierul nu poate fi deschis cu succes, programul va afișa un mesaj de eroare și se va returna din funcție.
11. const char\* pattern = "cuvant\_cheie";
12. Se definește pattern-ul căutat în variabila pattern. În exemplul dat, am folosit un pattern simplu, "cuvant\_cheie", dar acesta poate fi înlocuit cu orice alt pattern specific pe care doriți să îl căutați în datele din fișier.
13. char line[100];
14. Se inițializează un buffer de caractere line pentru a citi fiecare linie din fișier.
15. int lineIndex = 1;
16. Se definește o variabilă lineIndex care va reprezenta numărul liniei curente în fișier. Aceasta va fi folosită pentru a afișa în mesajul de identificare a pattern-ului în ce linie s-a găsit acesta.
17. while (fgets(line, sizeof(line), file)) {
18. Într-o buclă while, se citesc linii succesive din fișier utilizând funcția fgets. Fiecare linie este stocată în buffer-ul line.
19. char\* position = strstr(line, pattern);
20. if (position != NULL) {
21. printf("Pattern identificat in linia %d: %s", lineIndex, line);
22. }
23. Se utilizează funcția strstr pentru a căuta pattern-ul pattern în fiecare linie. Funcția strstr returnează un pointer către prima apariție a pattern-ului în linie sau NULL dacă pattern-ul nu este găsit.
24. lineIndex++;
25. }
26. fclose(file);
27. }
28. Dacă pattern-ul este găsit într-o linie, se afișează un mesaj care indică numărul liniei și conținutul liniei respective prin utilizarea variabilei lineIndex și a buffer-ului line.
29. După fiecare linie citită, se incrementează lineIndex pentru a indica următoarea linie.
30. După ce au fost procesate toate liniile din fișier, se închide fișierul utilizând funcția fclose.
31. int main() {
32. const char\* filename = "data.txt";
33. identifyPatterns(filename);
34. return 0;
35. }
36. În funcția ‘main’, se definește numele fișierului filename și se apelează funcția ‘identifyPatterns’ pentru a identifica pattern-urile în datele din fișierul respectiv.

Aceasta este structura generală a programului pentru identificarea de pattern-uri în date dintr-un fișier text. Vă puteți adapta și extinde logica programului pentru a satisface cerințele și specificațiile dvs. specifice.

# CONCLUZII

În concluzie, tema backtracking și analiza sa detaliată în această teză de curs au adus în prim plan o metodă de rezolvare eficientă a problemelor în sfera IT. Backtracking-ul este un algoritm recursiv puternic, care se bazează pe explorarea exhaustivă a spațiului de căutare pentru a găsi soluții optime sau satisfăcătoare. A fost prezentată o definiție a metodei backtracking și s-au evidențiat caracteristicile sale principale, precum și algoritmul general pentru implementarea sa.

În cadrul tezei, s-au discutat avantajele și dezavantajele utilizării backtracking-ului, comparându-l cu alte metode de rezolvare. S-au evidențiat exemple concrete de probleme care pot fi rezolvate cu ajutorul backtracking-ului, precum problema rucsacului, colorarea grafului, jocul Sudoku și jocul 8-regine.

Un aspect important a fost analiza complexității algoritmului de backtracking și identificarea tehnicilor de optimizare pentru reducerea acesteia. S-au explorat tehnici precum ordinea de explorare a stărilor, euristicile de alegere a opțiunilor și optimizarea bazată pe informații specifice problemei. Aceste tehnici pot fi aplicate pentru a reduce timpul și spațiul de căutare necesare în rezolvarea problemelor.

De asemenea, s-a discutat despre aplicarea backtracking-ului în analiza de date, inclusiv selecția de date dintr-o bază de date și identificarea de pattern-uri în date. Aceasta arată că metoda backtracking nu este limitată doar la problemele clasice de optimizare, ci poate fi aplicată și în alte domenii ale informaticii.

În final, teza a evidențiat că backtracking-ul este o metodă puternică și versatilă pentru rezolvarea problemelor. Cu toate acestea, eficiența și performanța algoritmului pot fi influențate de implementarea corectă a tehnicilor de optimizare și de natura specifică a problemei. Este important să se înțeleagă caracteristicile și limitările backtracking-ului înainte de a-l aplica într-un anumit context.

În ansamblu, teza a oferit o perspectivă cuprinzătoare asupra metodei backtracking, analizând atât aspectele teoretice, cât și aplicațiile practice ale acesteia. Acest studiu contribuie la dezvoltarea cunoștințelor și abilităților în sfera IT și poate servi drept bază pentru investigații ulterioare în domeniu.

# BIBLIOGRAFIE

* 1. Skiena, S. S. (2008). The Algorithm Design Manual. Springer Science & Business Media.
  2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
  3. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (1983). Data Structures and Algorithms. Addison-Wesley.
  4. Knuth, D. E. (2011). The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability. Addison-Wesley Professional.
  5. Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-Wesley Professional.
  6. Beigel, R., & Eppstein, D. (1993). 3-coloring in time O(1.3289^n). Journal of Algorithms, 14(2), 302-316.
  7. Cormode, G., & Muthukrishnan, S. (2004). An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms, 55(1), 58-75.
  8. Norvig, P. (2009). Solving Every Sudoku Puzzle.
  9. Crook, E. D. (2009). Solving Sudoku: A Graph Coloring Approach.
  10. Le, P. (2012). Sudoku Solver Using Backtracking Algorithm. International Journal of Computer Science and Information Technologies, 3(5), 5001-5003.
  11. Chaurasia, V., & Kumar, R. (2016). Sudoku Solver using Backtracking. International Journal of Scientific and Research Publications, 6(4), 425-428.
  12. Crook, E. D. (2017). Backtracking Algorithms. În Handbook of Graph Theory (Ed. by Gross, J. L., Yellen, J., Zhang, P.), CRC Press.
  13. Tovey, C. A. (1992). A Simplified NP-Complete Soduko Puzzle. Complexity International, 4(1), 111-112.
  14. Russell, S. J., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Pearson.

# ANEXE

**1.Codul pentru problema rucsacului:**

#include <stdio.h>

#define MAX\_N 100

#define MAX\_WEIGHT 1000

int max\_value = 0; // Valoarea maximă obținută

int best\_solution[MAX\_N]; // Cea mai bună soluție găsită

int weights[MAX\_N]; // Vectorul de greutăți ale obiectelor

int values[MAX\_N]; // Vectorul de valori ale obiectelor

void backtrack(int current\_item, int current\_weight, int current\_value, int n, int capacity) {

if (current\_weight > capacity) {

return; // Depășirea capacității rucsacului, nu este o soluție validă

}

if (current\_value > max\_value) {

max\_value = current\_value; // Actualizăm valoarea maximă obținută

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

best\_solution[i] = (current\_item >> i) & 1; // Salvăm configurația curentă ca fiind cea mai bună soluție

}

}

if (current\_item == n) {

return; // Am explorat toate obiectele, nu mai avem opțiuni de selectare

}

// Alegem să luăm obiectul curent și continuăm explorarea

backtrack(current\_item + 1, current\_weight + weights[current\_item], current\_value + values[current\_item], n, capacity);

// Alegem să nu luăm obiectul curent și continuăm explorarea

backtrack(current\_item + 1, current\_weight, current\_value, n, capacity);

}

int main() {

int n; // Numărul de obiecte disponibile

int capacity; // Capacitatea maximă a rucsacului

printf("Introduceți numărul de obiecte: ");

scanf("%d", &n);

printf("Introduceți capacitatea rucsacului: ");

scanf("%d", &capacity);

printf("Introduceți greutățile obiectelor:\n");

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

scanf("%d", &weights[i]);

}

printf("Introduceți valorile obiectelor:\n");

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

scanf("%d", &values[i]);

}

backtrack(0, 0, 0, n, capacity); // Apelăm funcția de backtracking

printf("Soluția optimă este:\n");

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

if (best\_solution[i]) {

printf("Obiectul %d\n", i + 1);

}

}

printf("Valoarea maximă obținută: %d\n", max\_value);

return 0;**}**

**Codul pentru problema colorarii grafului:**

#include <stdio.h>

#define MAX\_NODES 100

int graph[MAX\_NODES][MAX\_NODES];

int colors[MAX\_NODES];

int numNodes;

// Funcție pentru a verifica dacă o culoare poate fi atribuită unui nod

int isSafe(int node, int color, int colorAssignment[]) {

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

if (graph[node][i] && colorAssignment[i] == color) {

return 0;

}

}

return 1;

}

// Funcție pentru a colora graful folosind backtracking

int graphColoringUtil(int numColors, int colorAssignment[], int node) {

// Cazul de oprire: toate nodurile au fost colorate

if (node == numNodes) {

return 1;

}

// Încercăm toate culorile pentru nodul curent

for (int color = 1; color <= numColors; color++) {

// Verificăm dacă putem atribui culoarea 'color' nodului 'node'

if (isSafe(node, color, colorAssignment)) {

colorAssignment[node] = color;

// Recursivitate pentru a colora nodurile rămase

if (graphColoringUtil(numColors, colorAssignment, node + 1) == 1) {

return 1;

}

// Dacă colorarea nodului nu duce la o soluție, resetăm culoarea

colorAssignment[node] = 0;

}

}

return 0;

}

// Funcție pentru a rezolva problema de colorare a grafului

void graphColoring(int numColors) {

int colorAssignment[MAX\_NODES];

// Inițializăm toate nodurile cu 0 (nicio culoare atribuită)

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

colorAssignment[i] = 0;

}

// Apelăm funcția de backtracking pentru nodul 0

if (graphColoringUtil(numColors, colorAssignment, 0) == 0) {

printf("Nu există o soluție de colorare cu %d culori.\n", numColors);

} else {

printf("Soluția de colorare cu %d culori este:\n", numColors);

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

printf("Nodul %d: Culoare %d\n", i, colorAssignment[i]);

}

}

}

int main() {

int numEdges;

int node1, node2;

printf("Introduceți numărul de noduri în graf: ");

scanf("%d", &numNodes);

printf("Introduceți numărul de muchii în graf: ");

scanf("%d", &numEdges);

printf("Introduceți muchiile (perechea de noduri):\n");

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

scanf("%d %d", &node1, &node2);

graph[node1][node2] = 1;

graph[node2][node1] = 1;

}

int numColors;

printf("Introduceți numărul maxim de culori:**”);**

scanf("%d", &numColors);

graphColoring(numColors);

return 0;

**Codul pentru problema** TSP (Travelling Salesman Problem)**:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 10

int graph[MAX][MAX]; // Matricea de adiacență pentru graf

int visited[MAX]; // Vectorul de noduri vizitate

int n; // Numărul de noduri în graf

int costMinim = INT\_MAX; // Costul minim găsit până în prezent

void tsp(int current, int count, int cost, int path[]) {

int i;

visited[current] = 1; // Se marchează nodul curent ca fiind vizitat

path[count] = current; // Se adaugă nodul în calea curentă

// Cazul de oprire: toate nodurile au fost vizitate

if (count == n - 1) {

// Se adaugă costul muchiei de la ultimul nod la primul nod

cost += graph[current][0];

// Verificăm dacă este o soluție mai bună decât minimul curent

if (cost < costMinim) {

costMinim = cost;

printf("Cost minim: %d\n", costMinim);

// Se afișează calea minimă

printf("Cale minimă: ");

for (i = 0; i < n; i++) {

printf("%d ", path[i]);

}

printf("0\n"); // Adăugăm nodul de start la final

}

}

else {

// Se continuă explorarea pentru toate nodurile nevizitate

for (i = 1; i < n; i++) {

if (!visited[i]) {

tsp(i, count + 1, cost + graph[current][i], path);

}

}

}

// Se marchează nodul curent ca nevizitat pentru a putea fi explorat din nou în alte ramuri

visited[current] = 0;

}

int main() {

int i, j;

printf("Introduceți numărul de noduri în graf: ");

scanf("%d", &n);

printf("Introduceți matricea de adiacență pentru graf:\n");

for (i = 0; i < n; i++) {

for (j = 0; j < n; j++) {

scanf("%d", &graph[i][j]);

}

}

int path[MAX];

// Se initializează vectorul de noduri vizitate cu 0 (nevizitate)

for (i = 0; i < n; i++) {

visited[i] = 0;

}

// Se pornește algoritmul de backtracking de la nodul de start 0

tsp(0, 0, 0, path);

return 0;

}

**Codul pentru jocul Sudoku:**

#include <stdio.h>

#include <stdbool.h>

#define SIZE 9

// Verifică dacă o valoare poate fi plasată într-o anumită poziție a tablei Sudoku

bool isSafe(int grid[SIZE][SIZE], int row, int col, int num) {

// Verifică dacă numărul se regăsește deja în aceeași linie

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

if (grid[row][i] == num) {

return false;

}

}

// Verifică dacă numărul se regăsește deja în aceeași coloană

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

if (grid[i][col] == num) {

return false;

}

}

// Verifică dacă numărul se regăsește deja în aceeași sub-grilă de 3x3

int startRow = row - (row % 3);

int startCol = col - (col % 3);

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (grid[i + startRow][j + startCol] == num) {

return false;

}

}

}

return true;

}

// Găsește o celulă goală în tablă și returnează rândul și coloana acesteia

bool findEmptyCell(int grid[SIZE][SIZE], int \*row, int \*col) {

for (\*row = 0; \*row < SIZE; (\*row)++) {

for (\*col = 0; \*col < SIZE; (\*col)++) {

if (grid[\*row][\*col] == 0) {

return true;

}

}

}

return false;

}

// Rezolvă tabla Sudoku utilizând backtracking

bool solveSudoku(int grid[SIZE][SIZE]) {

int row, col;

if (!findEmptyCell(grid, &row, &col)) {

// Nu mai sunt celule goale, tabla este completă

return true;

}

// Testează valorile de la 1 la 9 pentru celula curentă

for (int num = 1; num <= SIZE; num++) {

if (isSafe(grid, row, col, num)) {

grid[row][col] = num;

if (solveSudoku(grid)) {

// Tabla este completă, s-a găsit o soluție

return true;

}

// Soluția nu este validă, se revine la valoarea anterioară

grid[row][col] = 0;

}

}

// Nu există o soluție validă pentru tabla Sudoku

return false;

}

// Funcție pentru afișarea tablei Sudoku

void printSudoku(int grid[SIZE][SIZE]) {

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

for (int j = 0; j < SIZE; j++) {

printf("%d ", grid[i][j]);

}

printf("\n");

}

}

int main() {

int grid[SIZE][SIZE] = {

{

{5, 3, 0, 0, 7, 0, 0, 0, 0},

{6, 0, 0, 1, 9, 5, 0, 0, 0},

{0, 9, 8, 0, 0, 0, 0, 6, 0},

{8, 0, 0, 0, 6, 0, 0, 0, 3},

{4, 0, 0, 8, 0, 3, 0, 0, 1},

{7, 0, 0, 0, 2, 0, 0, 0, 6},

{0, 6, 0, 0, 0, 0, 2, 8, 0},

{0, 0, 0, 4, 1, 9, 0, 0, 5},

{0, 0, 0, 0, 8, 0, 0, 7, 9}

};

printf("Sudoku initial:\n");

printSudoku(grid);

if (solveSudoku(grid)) {

printf("\nSudoku rezolvat:\n");

printSudoku(grid);

} else {

printf("\nNu exista o solutie valida pentru Sudoku!\n");

}

return 0;**}**

**Codul pentru jocul 8-regine:**

#include <stdio.h>

#define N 8

void printSolution(int board[N][N]) {

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

for (int j = 0; j < N; j++) {

printf("%d ", board[i][j]);

}

printf("\n");

}

printf("\n");

}

int isSafe(int board[N][N], int row, int col) {

// Verificăm rândul în stânga

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

if (board[row][i])

return 0;

}

// Verificăm diagonală superioară-stânga

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j])

return 0;

}

// Verificăm diagonală inferioară-stânga

for (int i = row, j = col; i < N && j >= 0; i++, j--) {

if (board[i][j])

return 0;

}

return 1;

}

int solveNQueens(int board[N][N], int col) {

if (col >= N) {

printSolution(board);

return 1;

}

int res = 0;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

res += solveNQueens(board, col + 1);

board[i][col] = 0; // backtracking

}

}

return res;

}

int main() {

int board[N][N] = {0};

int solutions = solveNQueens(board, 0);

printf("Numărul total de soluții: %d\n", solutions);

return 0;

}

**Codul pentru i**dentificarea de pattern-uri în date**:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void identifyPatterns(const char\* filename) {

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

if (file == NULL) {

printf("Eroare la deschiderea fisierului.");

return;

}

// Definim pattern-ul căutat

const char\* pattern = "cuvant\_cheie";

// Citim datele din fisier linie cu linie

char line[100];

int lineIndex = 1;

while (fgets(line, sizeof(line), file)) {

// Identificăm pattern-ul în fiecare linie

char\* position = strstr(line, pattern);

if (position != NULL) {

printf("Pattern identificat in linia %d: %s", lineIndex, line);

}

lineIndex++;

}

fclose(file);

}

int main() {

const char\* filename = "data.txt";

identifyPatterns(filename);

return 0;

}