Project 1 - Search Problems

Adela Morar

November 2025

1 Introducere

Algoritmii de căutare sunt folosiți pentru a găsi un drum de la o stare inițială la o stare scop. Aceștia diferă prin felul în care explorează spațiul de stări, tipul de structură de date folosit și garantiile privind optimalitatea soluției. În cele ce urmează sunt prezentate cele mai importante metode de căutare:

- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Uniform Cost Search (UCS)
- A* Search
- BONUS: Greedy Best-First Search

2 Depth-First Search (DFS)

Descriere

Algoritmul **Depth-First Search** este utilizat pentru rezolvarea unei probleme de căutare (definită de clasa **SearchProblem**). Scopul său este de a găsi o secvență de acțiuni care duce de la starea inițială la o stare scop (goal state), explorând mai întâi nodurile cele mai adânci din arborele de căutare. Utilizează o **stivă** (LIFO) pentru a gestiona frontiera de căutare.

Avantaje

- **Necesită puțină memorie:** Complexitatea spațială este proporțională cu adâncimea maximă a arborelui de căutare, $O(b \cdot d)$, unde b este factorul de ramificare (numărul maxim de succesori) și d este adâncimea soluției.

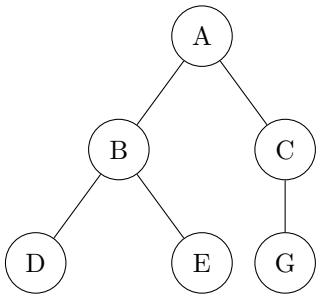
Dezavantaje

- **Poate intra în buclă infinită:** Daca spațiul de căutare este infinit sau conține cicluri, DFS poate explora o singură ramură infinită și nu va găsi niciodată scopul, chiar dacă acesta există pe o altă ramură.
- **Nu garantează soluția optimă:** Dacă prima stare scop găsită se află pe o ramură foarte adâncă și există o altă stare scop mult mai aproape de rădăcină (mai puțin adâncă), DFS va returna calea mai lungă, deoarece prioritizează adâncimea.

Pseudocod DFS

```
DFS(problem):
    // stochează elemente de forma (st_crnt, [actiuni până aici])
    creează stiva frontier
    creează setul visited // stari deja vizitate

    frontier.push( (start, []) ) // adaugăm starea inițială în frontieră

    cât timp frontier nu e goală:
        // extragem vf stivei (LIFO)
        (state, actions) = frontier.pop()

        dacă state este goal:
            return actions

        dacă state nu e în visited:
            adaugă state în visited

            //mergem mai departe
            pentru fiecare succesor(next, action, cost):
                //punem in frontiera nodul nou si drumul pana la el
                new_actions = actions + [action]
                frontier.push( (next, new_actions) )

    return []
```

Diagramă DFS



DFS explorează: $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow G$

3 Breadth-First Search (BFS)

Descriere

Algoritmul **Breadth-First Search** (Căutare în Lățime) explorează mai întâi nodurile cele mai superficiale (cele mai apropiate de rădăcină). Acest algoritm garantează că va găsi cea mai scurtă cale în termeni de număr de pași. Utilizează o **coadă** (FIFO) pentru a gestiona frontiera de căutare.

Avantaje

- **Optimalitate:** Garantează că va găsi **cea mai scurtă cale** (calea optimă) în termeni de număr de acțiuni necesare pentru a ajunge la scop (presupunând costuri uniforme).
- **Completitudine:** Dacă o soluție există, BFS o va găsi întotdeauna.

Dezavantaje

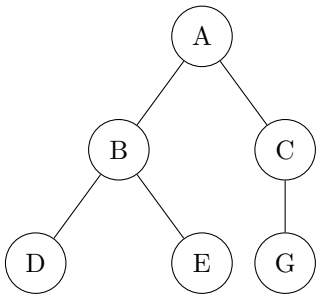
- **Necesită Multă Memorie:** Complexitatea spațială este exponențială cu adâncimea soluției, $O(b^d)$, deoarece trebuie să stocheze toate nodurile de la nivelul final al explorării în coadă.
- **Lent pe Soluții Adânci:** Timpul de execuție este, de asemenea, exponențial, $O(b^d)$, ceea ce îl face nepractic pentru spații de căutare foarte mari și adânci.

Pseudocod BFS

```
BFS(problem):
    // stochează elemente de forma (st_crnt, [actiuni până aici])
    creează coada frontier
    creează setul visited // stari deja vizitate
```

```
frontier.push( (start, []) ) // adaugăm starea inițială

cât timp frontier nu e goală:
    // extragem primul element (FIFO)
    (state, actions) = frontier.pop()

    dacă state este goal:
        return actions

    dacă state nu e în visited:
        adaugă state în visited

        //mergem mai departe
        pentru fiecare succesor(next, action, cost):
            new_actions = actions + [action]
            frontier.push( (next, new_actions) )

return []
```

Diagramă BFS



Ordinea de explorare: $A \to B \to C \to D \to E \to G$

4 Uniform Cost Search (UCS)

Descriere

Algoritmul **Uniform Cost Search** (Căutare după Cost Uniform) explorează întotdeauna nodul care are cel mai mic cost total cumulat de la starea inițială ($g(n)$). Obiectivul său este să găsească calea cu costul minim. Utilizează o **coadă cu prioritate** ('PriorityQueue'), unde prioritatea este costul total.

Structuri de Date Cheie

- **Frontiera:** Coadă cu Prioritate (prioritate = cost total $g(n)$).
- **Costuri Stocate (`cost_so_far`):** Dicționar care reține cel mai mic cost cunoscut pentru a ajunge la fiecare stare, esențial pentru **optimalitate** și prevenirea re-explorării ineficiente.

Avantaje

- **Optimalitate Garantată:** Găsește calea cu **cel mai mic cost total** pentru costuri de acțiune neuniforme (pozitive).
- **Adecvat pentru Costuri Variabile:** Cel mai bun algoritm pentru grafuri cu costuri ponderate.

Dezavantaje

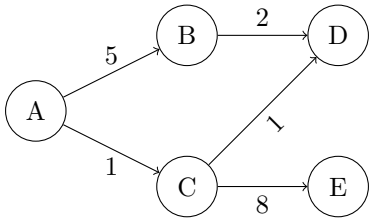
- **Ineficient:** Similar cu BFS, este ineficient în timp și spațiu pentru spații mari, deoarece nu folosește euristică.
- **Sensibil la Costuri Mici:** Poate explora un număr mare de noduri cu costuri marginal diferite înainte de a progresa spre scop.

Pseudocod UCS

```
UCS(problem):
    creează Coada cu Prioritate frontier
    creează cost_so_far // {state : cost_minim_cunoscut}

    initial_node = Node(parent = None, start_state, action = None, cost=0)
    frontier.push(initial_node, 0)
    cost_so_far[start_state] = 0

    cât timp frontier nu e goală:
        current_node = frontier.pop()
        current_state = current_node.state
        current_cost = current_node.cost

        dacă current_state este goal:
            return reconstruiește_calea(current_node)

        pentru fiecare succesor, action, step_cost:
            new_cost = current_cost + step_cost

            dacă succesor nu e în cost_so_far SAU new_cost < cost_so_far[succesor]:
                cost_so_far[succesor] = new_cost
                new_node = Node(parent=current_node, state=succesor,
```

```
                    action, new_cost)
                frontier.push(new_node, new_cost)

    return []
```

Diagramă UCS



Start A. Nodul țintă este D.

1. Frontiera conține: {`(A, cost 0)`}
2. Extrage A. Adaugă succesori: {`(C, cost 1)`, `(B, cost 5)`}
3. Extrage **C** (cost minim 1). Adaugă succesori: {`(D, cost 1+1=2)`, `(E, cost 1+8=9)`}
4. Frontiera conține: {`(D, cost 2)`, `(B, cost 5)`, `(E, cost 9)`}
5. Extrage **D** (cost minim 2). Nodul țintă a fost găsit.

Calea Optima: A → C → D (Cost Total 2).

5 A* Search (A-Star)

Descriere

Algoritmul **A* Search** (A-Star) este cel mai popular algoritm de căutare "informată" (informed search). El combină avantajele algoritmului **Uniform Cost Search (UCS)** cu o funcție de estimare euristică $h(n)$. A* explorează nodul care are cea mai mică valoare a funcției de evaluare $f(n)$, definită ca:

$$f(n) = g(n) + h(n)$$

unde:

- $g(n)$: Costul real al căii de la starea inițială la nodul curent n.
- $h(n)$: Costul estimat (euristica) de la nodul curent n la starea scop.

A* utilizează o **coadă cu prioritate** prioritatea fiind dată de valoarea $f(n)$.

Structuri de Date Cheie

- **Frontieră (Coadă cu Prioritate):** Prioritizează nodurile pe baza costului total estimat $f(n) = g(n) + h(n)$.
- **Costuri Reale (‘cost_so_far‘):** Un dicționar care stochează costul real minim $g(n)$ găsit până în acel moment pentru a ajunge la fiecare stare. Acest lucru este esențial pentru a verifica dacă a fost descoperită o cale mai ieftină spre o stare deja vizitată.

Avantaje

- **Optimalitate Garantată:** Dacă euristica $h(n)$ este **admisibilă** (nu supraestimează niciodată costul real până la scop) și **consistentă**, A* este garantat să găsească calea cu **cel mai mic cost total**.
- **Eficiență:** Datorită euristicii, A* este mult mai eficient decât UCS și evită explorarea ramurilor inutile care duc la costuri mari.

Dezavantaje

- **Complexitate Spațială:** Ca și BFS și UCS, A* poate necesita o cantitate mare de memorie, deoarece trebuie să stocheze toate nodurile generate în frontieră.
- **Depinde de Euristică:** Performanța și optimalitatea depind critic de calitatea euristicii $h(n)$. O euristică slabă poate degenera A* în UCS.

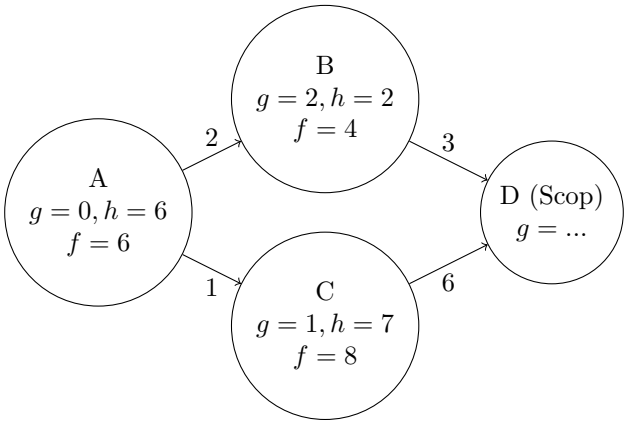
Pseudocod A* Search

```
A*Search(problem, heuristic):
    creează Coada cu Prioritate frontier (prioritate = f(n) = g(n) + h(n))
    creează cost_so_far // map {state : cost_real_g(n)}

    initial_state = problem.getStartState()
    // f(start) = 0 + h(start)
    initial_priority = heuristic(initial_state, problem)
    frontier.push(initial_node, initial_priority)
    cost_so_far[initial_state] = 0

    cât timp frontier nu e goală:
        current_node = frontier.pop() // extrage nodul cu f(n) minim
        current_state = current_node.state
        g_cost = current_node.cost

        dacă current_state este goal:
            return reconstruiește_calea(current_node)
```

```
        // Explorarea succesorilor
        pentru fiecare succesor, action, step_cost:
            new_g = g_cost + step_cost
            h = heuristic(succesor, problem)
            f = new_g + h

            // Dacă s-a găsit o cale mai ieftină (g) sau e un nod nou
            dacă succesor nu e în cost_so_far SAU new_g < cost_so_far[succesor]:
                cost_so_far[succesor] = new_g  // actualizează costul real
                new_node = Node(parent=current_node, state=succesor,
                                action, new_g)
                // prioritizează pe baza lui f(n)
                frontier.push(new_node, f)
    return []
```

Diagramă A* Search (Exemplu de Evaluare)

Acest graf ilustrează modul în care A* folosește $f(n) = g(n) + h(n)$ pentru a prioritiza explorarea. În acest caz, deși nodul B are un cost real $g(n)$ mai mare decât C, estimarea euristică $h(n)$ îl face să pară mai promițător.



Decizie A* la pasul 1:
Nodul B: $f(B) = g(B) + h(B) = 2 + 2 = 4$
Nodul C: $f(C) = g(C) + h(C) = 1 + 7 = 8$

Extragere → **B** (f=4), deoarece costul estimat total $f(n)$ este cel mai mic.
UCS ar fi extras C (cost real $g = 1$).

6 BONUS: Greedy Best-First Search (GBFS)

Descriere

Algoritmul **Greedy Best-First Search** este un algoritm de căutare "informată" (informed search) care se concentrează exclusiv pe estimarea euristică. GBFS explorează întotdeauna nodul care este estimat a fi cel mai aproape de starea scop. Funcția de evaluare este definită ca:

$$f(n) = h(n)$$

unde:

- $h(n)$: Costul estimat (euristica) de la nodul curent n la starea scop. (Costul real $g(n)$ nu este luat în considerare în prioritizare.)

Structuri de Date Cheie

- **Frontieră (Coadă cu Prioritate):** Prioritizează nodurile pe baza valorii **euristice** $h(n)$. Nodul cu cea mai mică valoare $h(n)$ este extras primul, fiind considerat cel mai promițător.
- **Stări Vizitate ('visited_states'):** Un set care înregistrează stările care au fost deja expandate, asigurând astfel că algoritmul nu re-explorează stări.

Avantaje

- **Viteză:** De obicei, găsește o soluție mult mai repede decât algoritmii neinformați (BFS, DFS, UCS), deoarece este orientat direct spre scop prin intermediul euristicii.
- **Eficiență în Timp:** În scenarii ideale, poate fi extrem de eficient, explorând doar o mică parte din spațiul de căutare.

Dezavantaje

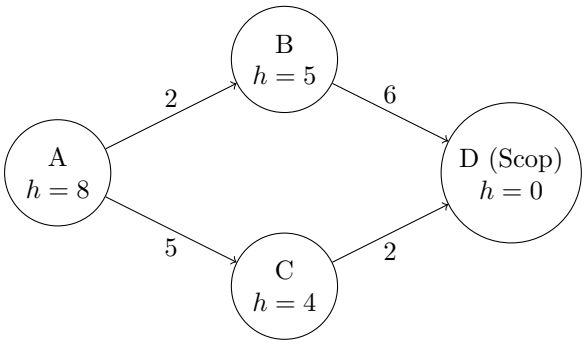
- **Lipsă de Optimalitate:** GBFS **nu este garantat** să găsească calea cu cel mai mic cost total, deoarece ignoră costul real $g(n)$ al căii parcurse până la nodul curent. Poate rămâne blocat pe o cale cu cost mare dacă aceasta are un $h(n)$ inițial mic.
- **Incompletitudine Potențială:** Similar cu DFS, dacă euristica direcționează căutarea pe o cale infinită sau foarte lungă, algoritmul poate eșua în găsirea scopului.
- **Sensibilitate la Euristică:** Performanța depinde în totalitate de calitatea și precizia funcției euristice $h(n)$.

Pseudocod Greedy Best-First Search

```
GBFS(problem, heuristic):
    creează Coada cu Prioritate frontier (prioritate = h(n))
    creează setul visited

    initial_priority = heuristic(start)
    frontier.push(initial_node, initial_priority)

    cât timp frontier nu e goală:
        current_node = frontier.pop()
        current_state = current_node.state

        adaugă current_state la visited

        dacă current_state este goal:
            return reconstruiește_calea(current_node)

        // Explorarea succesorilor
        pentru fiecare succesor, action, cost:
            dacă succesor nu e în visited:
                new_cost = current_node.cost + cost
                new_f = heuristic(succesor) // Se bazează doar pe h
                new_node = Node(parent=current_node, state=succesor,
                           action, new_cost)
                frontier.push(new_node, new_f)

    return []
```

Diagramă GBFS (Exemplu de Prioritizare)

În acest exemplu, deși calea A $\rightarrow$ C $\rightarrow$ D este mai scumpă (cost total 7), GBFS alege nodul C mai întâi (dacă $h(C)$ este mai mic decât $h(B)$), urmând exclusiv euristica.



Decizie GBFS la pasul 1:

Nodul B: $f(B) = h(B) = 5$

Nodul C: $f(C) = h(C) = 4$

Extragere $\rightarrow$ **C** (f=4).

Atenție: Deși calea A $\rightarrow$ B $\rightarrow$ D (cost total 8) este mai ieftină decât A $\rightarrow$ C $\rightarrow$ D (cost total 7), GBFS o va găsi pe cea bazată pe $h(n)$.