Inteligenta Artificiala

Strategii de cautare in Inteligenta Artificiala

Laborator 1

O strategie este definită prin alegerea ordinul de expansiune a unui nod. Strategiile sunt evaluate pe baza următoarelor dimensiuni:

- Completitudine – Intotdeauna gaseste o solutie, daca aceasta exista?
- Optimalitate – Intotdeauna gaseste o soluție de cost minimal?
- Complexitate Timp - Numărul de noduri generate / extinse
- Complexitate Spatiu - Numărul maxim de noduri salvate în memorie

Complexitatea Timp și Complexitatea Spatiu sunt măsurate în functie de :

- b - factorul maxim de ramuri din arborele de căutare
- d - adâncimea soluție de ramificare cu cel mai mic cost
- m - adâncimea maximă a spațiului de stare (poate fi ∞)

Strategii de cautare neinformate

Strategiile de cautare neinformate(Blind Search) folosesc doar informatiile existente in definirea problemei.Solutiile la problem sunt gasite prin generarea sistematica de stari noi si verificarea daca s-a ajuns la starea tinta/solutia problemei.

In parcurgerea spatiului de cautare un nod poate fi:

- *necunoscut* - nodul apartine partii neexplorate a spatiului de cautare,
- *evaluat* - nodul este cunoscut dar fie nu se cunoaste nici un succesor al lui, fie se cunosc numai o parte din succesorii lui,
- *extins* - nodul este cunoscut si, in plus, se cunosc toti succesorii lui.

Prin extinderea unui nod se intelege generarea tuturor succesorilor acelui nod. Aceasta inseamna obtinerea tuturor starilor urmatoare starii curente S, prin aplicarea tuturor operatorilor legali in starea S.

In procesul de cautare se vor folosi doua liste:

- FRONTIERA - lista nodurilor evaluate

- TERITORIU - lista nodurilor extinse

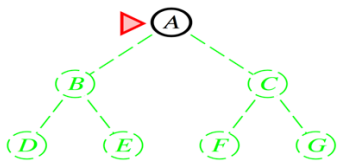
Lista FRONTIERA reprezinta frontiera spatiului de cautare parcurs (explicitat) spre partea necunoscuta a spatiului de cautare. Lista TERITORIU reprezinta partea cunoscuta a spatiului de cautare.

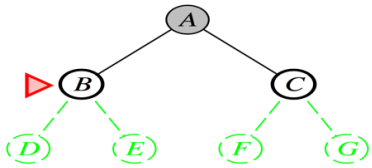
Strategiile de cautare neinformate(Blind Search) sunt:

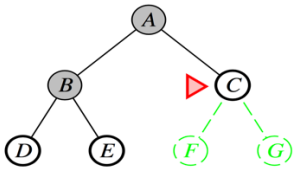
1. Breadth First Search (Cautarea pe nivel)
2. Uniform Cost Search
3. Depth First Search (Cautarea in adancime)
4. Depth-limited search
5. Iterative deepening search

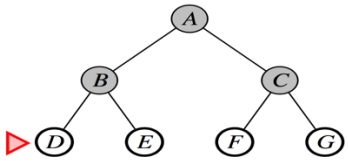
A. Breadth First Search (Cautarea pe nivel)

Cautarea pe nivel, numita si cautare in latime, este o strategie care expandeaza starile urmatoare in ordinea apropierii fata de nodul stare initiala. Cu alte cuvinte, aceasta strategie considera intai toate secventele posibile de n operatori inaintea secventelor de n+1 operatori.









Pseudocod

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)   /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Exemplu –Puzzle

1	2	3
8		4
7	6	5

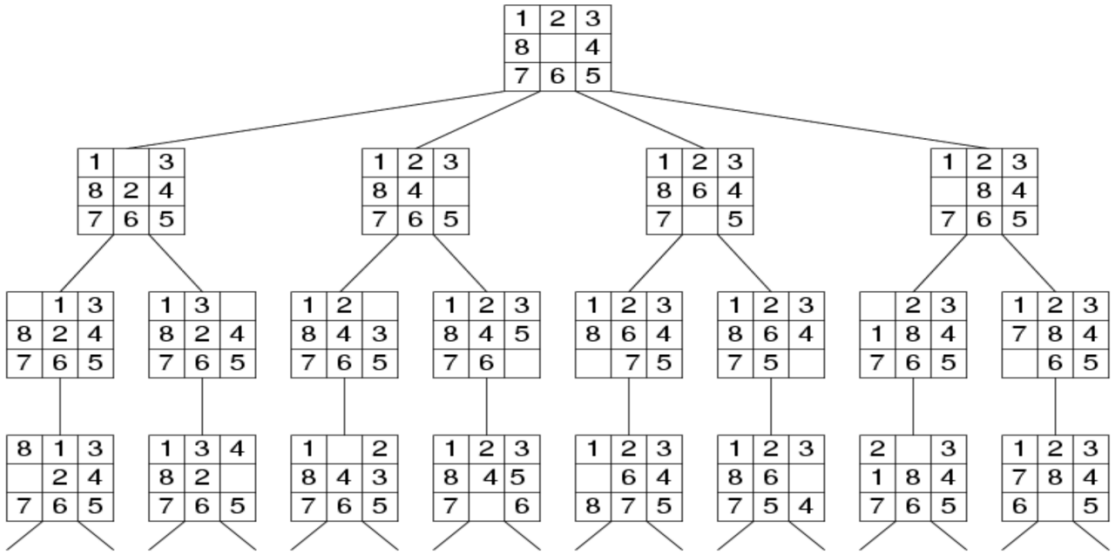
Start state

2		3
1	8	4
7	6	5

Goal state

Stari: pozitia pieselor in puzzle
Actiuni: Mutari posibile ale piesei goale (stanga, dreapta, sus , jos)
Tinta: Goal State predefinit
Costul drumului: 1 per mutare

Arborele de cautare partial pentru exemplul de mai sus



Avantajele algoritmului Breath-First Search sunt:

- Algoritmul nu se va bloca niciodata in urma explorarii cailor nefolositoare
- Daca exista o solutie, acest algoritm o va gasi
- Daca exista mai multe solutii, acest algoritm va gasi solutia minimala, cea care necesita un numar mai mic de pasi

Dezavantajele algoritmului Breath-First Search sunt:

- Necesita un spatiu de memorie mare, deoarece fiecare nivel al arborelui trebuie memorat pentru a genera urmatorul nivel.
- Daca solutia este foarte departe de radacina, acest algoritm va necesita un timp de executie indelungat.

Proprietati ale algoritmului breadth-first search

- Completitudine : Da, daca b este finit
- Timp: $1+b+b^2+b^3+...+b^d+b(b^d-1)=O(b^d)$,
- Spatiu: $O(b^d)$ (pastreaza fiecare nod in memorie)
- Optim: Da, daca costul fiecarui pas este 1, nu e optim in general

Aplicatii

- Găsirea celei mai scurte căi între cele două noduri u și v, cu lungimea căii măsurată prin numărul de muchii
- Metoda Ford-Fulkerson pentru calculul debitului maxim într-o rețea de flux
- Construcția funcției de risc a matricei de tip Aho-Corasick.
- Serializarea / Deserializarea unui arbore binar vs serializarea ordonată permite arborelui să fie reconstruit într-o manieră eficientă

B. Uniform Cost Search

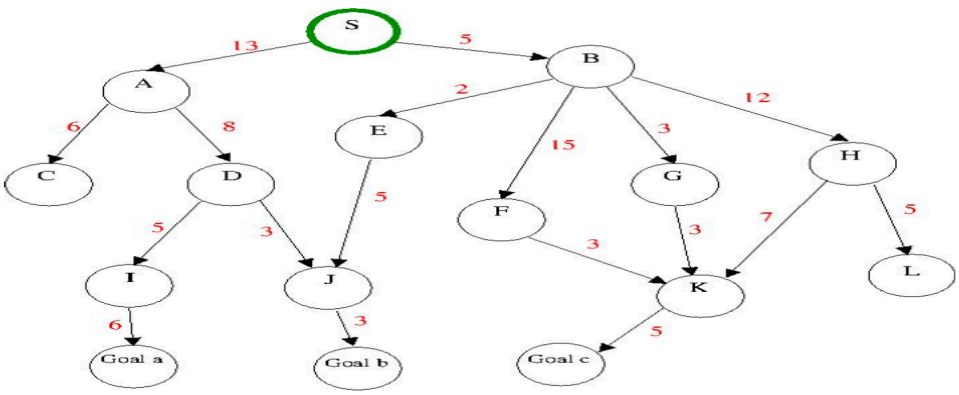
Cautare de cost uniform este un algoritm de cautare neinformata derivata din cautarea pe nivele. In cazul cautarii pe nivele, se extend nodurile care sunt cele mai apropiate de starea inițială în ceea ce privește numărul de acțiuni în calea lor.

Dar, în căutarea de cost uniform vom extinde nodurile care sunt cele mai apropiate de starea inițială în ceea ce privește costul cailor lor. În consecință, nodurile de pe margini vor avea costurile aproximativ egale. În cazul în care costul tuturor acțiunilor este aceeași, căutare de cost uniform este echivalent cu cautarea pe nivele.

Pseudocod

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Exemplu

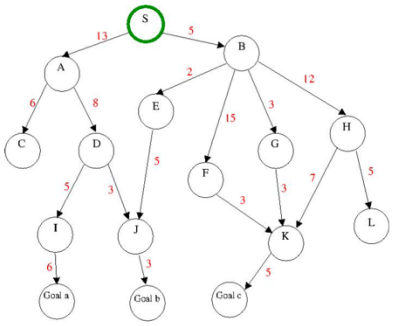


Stare de start: S

Stari tinta: Goal a, Goal b, Goal c

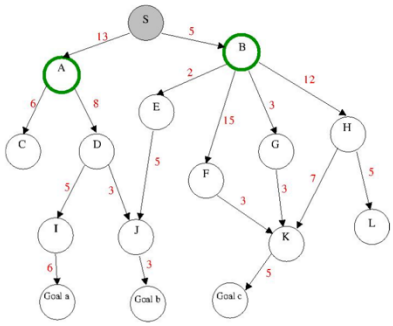
Construirea arborelui de cautare pentru algoritmul Uniform Cost Search:

Uniform Cost Search 1



S
0

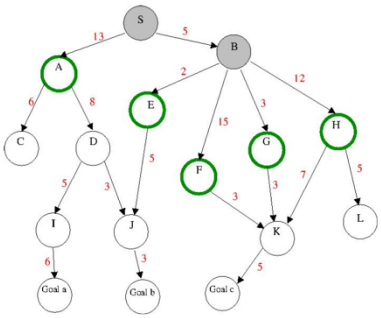
Uniform Cost Search 2



S	A	B
0	13	5

S

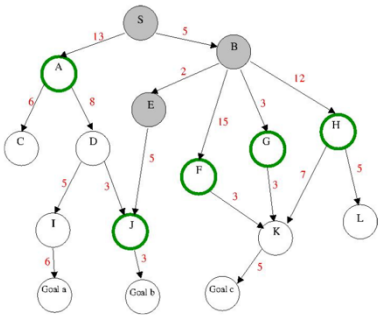
Uniform Cost Search 3



S	A	B	E	F	G	H
0	13	5	7	20	8	17

S, B

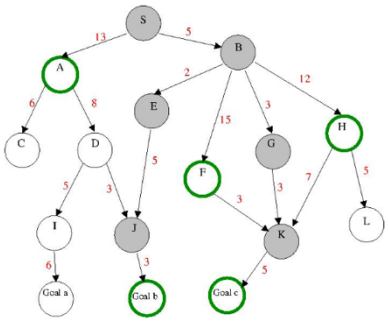
Uniform Cost Search 4



S	A	B	E	F	G	H	J
0	13	5	7	20	8	17	12

S, B, E

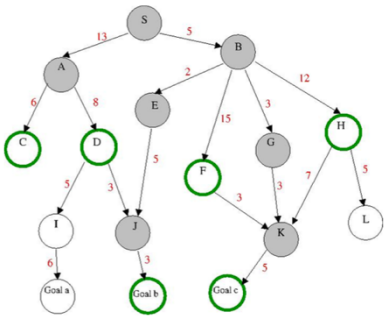
Uniform Cost Search 7



S	A	B	E	F	G	H	J	K	Goal c	Goal B
0	13	5	7	20	8	17	12	11	16	15

S, B, E, G, K, J

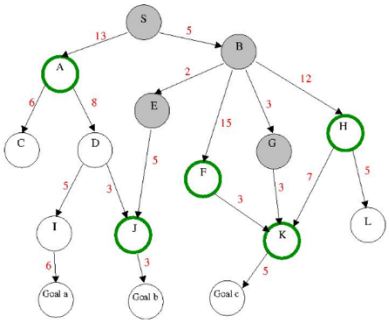
Uniform Cost Search 8



S	A	B	E	F	G	H	J	K	Goal c	Goal b	C	D
0	13	5	7	20	8	17	12	11	16	15	19	21

S, B, E, G, K, J, A

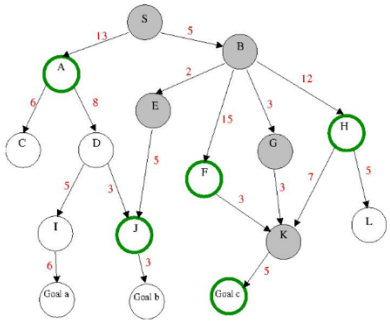
Uniform Cost Search 5



S	A	B	E	F	G	H	J	K
0	13	5	7	20	8	17	12	11

S, B, E, G

Uniform Cost Search 6



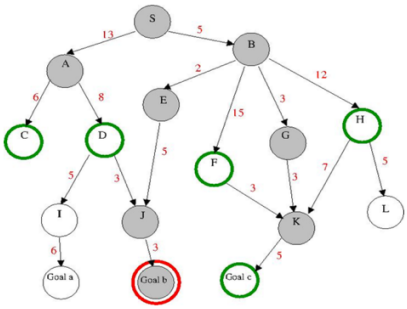
S	A	B	E	F	G	H	J	K	Goal C
0	13	5	7	20	8	17	12	11	16

S, B, E, G, K

Uniform Cost Search 9

Optimum Path:

S, B, E, J, Goal b



S	A	B	E	F	G	H	J	K	Goal c	Goal b	C	D
0	13	5	7	20	8	17	12	11	16	15	19	21

S, B, E, G, K, J, A, Goal b

Proprietati ale algoritmului uniform cost search

- Completitudine : Da, daca b costul pasului $\geq \partial > 0$
- Timp: Daca $g(n) \leq C^*$, $O(b^{\lceil C^*/\partial \rceil})$ unde C^* este costul solutiei optime si ∂ este costul pasului minimal
- Spatiu: La fel ca timpul
- Optim: Da, nodurile sunt extinse in ordine crescatoare a lui g(n)

C. Depth First Search (Cautarea in adancime)

Cautarea in adincime este o strategie care expandeaza starile cel mai recent generate, cu alte cuvinte nodurile cu adincimea cea mai mare din lista FRONTIERA. In consecinta, aceasta strategie parcurge o cale de la starea initiala pana la o stare ce poate fi stare finala sau care nu mai are nici un succesor. In acest ultim caz strategia revine pe nivelele anterioare si incearca explorarea altor cai posibile.

Strategia cautarii in adincime nu garanteaza obtinerea unei solutii a problemei, chiar in cazul in care solutia exista. O astfel de situatie poate apare, de exemplu, in cazul unui spatiu de cautare infinit in care ramura pe care s-a plecat in cautare nu contine solutia. Din acest motiv se introduce de obicei o limita a adancimii maxime de cautare, AdMax. Daca s-a atins aceasta limita fara a se gasi solutia, strategia revine si inspecteaza stari de

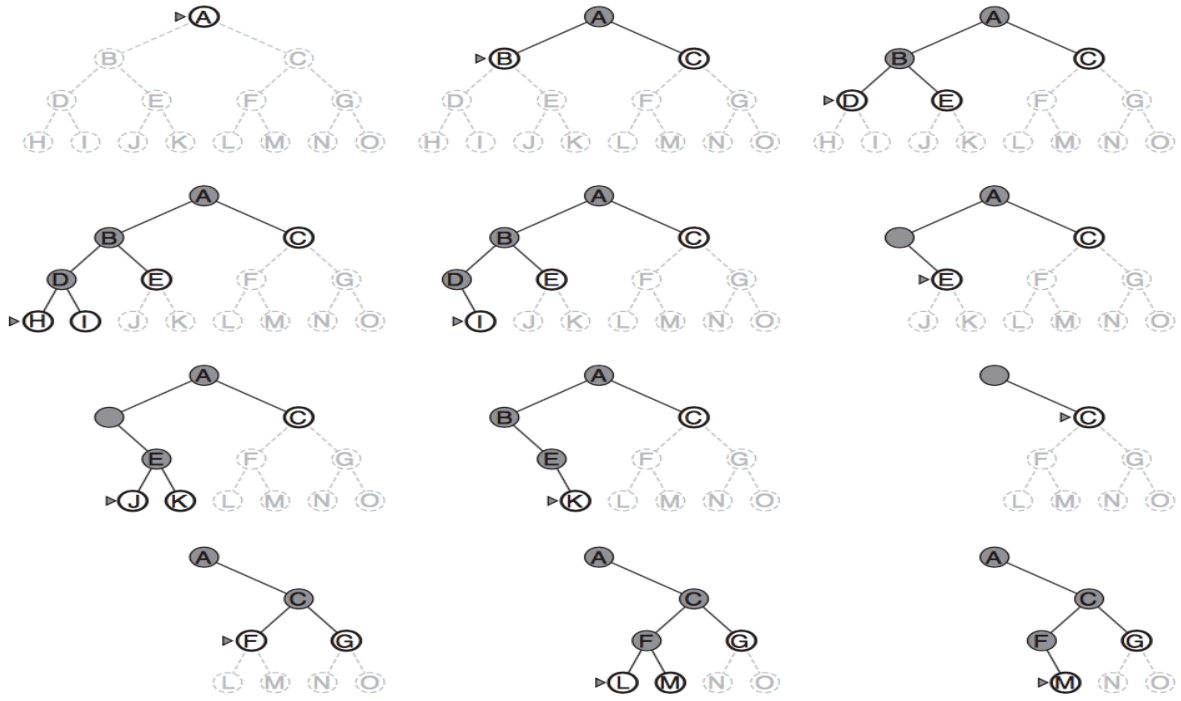
pe nivele inferioare lui AdMax dar aflate pe cai diferite. Solutia care s-ar gasi la o adincime de AdMax+p, de exemplu, ar fi pierduta. Daca strategia de cautare gaseste solutia, aceasta nu este neaparat calea cea mai scurta intre starea initiala si starea finala.

Pseudocod

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do

                S.push(w)
```

Exemplu



Proprietati ale algoritmului depth-first search

- Completitudine : Nu, in cazul in care solutia este pe un nivel foarte indepartat de radacina, algoritmul nu gaseste solutia(spatiu infinit)
- Timp: $O(b^m)$, algorimul functioneaza rau daca m este mai mare decat d, dar daca solutiile sunt dense poate fi mai rapid ca Breadth First Search
- Spatiu: La fel ca timpul
- Optim: Nu

Aplicatii

- Rezolva puzzle-uri cu o singură soluție, cum ar fi labirinturi. (DFS poate fi adaptat pentru a găsi toate soluțiile unui labirint incluzând numai noduri pe calea curentă din setul vizitat.)
- Găsirea componentelor conectate.
- Generarea labirintului poate folosi o căutare randomizată în funcție de adâncime.

D. Depth-limited search

Algoritmul depth limited search introduce o limită de adâncime pe ramuri care urmează să fie extinse. Regula de baza este de a nu extinde o ramură de mai jos de această adâncime. Algoritmul este util daca se cunoaste adâncimea maximă a soluției.

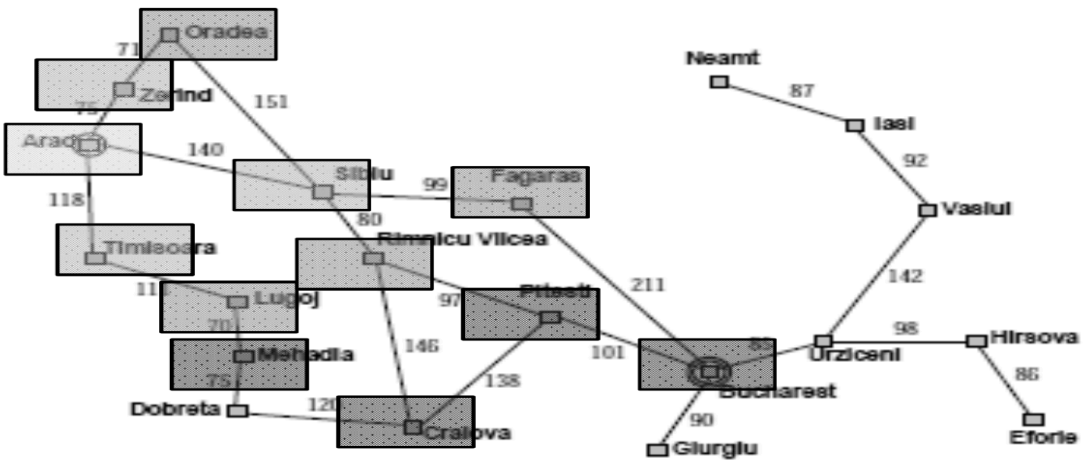
Pseudocod

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff
  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)
  else if *limit* = 0 **then return** *cutoff*
  else
    cutoff_occurred? ← false
    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**
      child ← CHILD-NODE(*problem*, *node*, *action*)
      result ← RECURSIVE-DLS(*child*, *problem*, *limit* − 1)
      if *result* = *cutoff* **then** *cutoff_occurred*? ← true
      else if *result* ≠ *failure* **then return** *result*
    if *cutoff_occurred*? **then return** *cutoff* **else return** *failure*

Exemplu

D =0 D =1 D =2 D =3



Proprietati ale algoritmului depth-limited search

- Completitudine : Da, daca $l \geq d$, l – limita de adancime, d - adâncimea soluție de ramificare cu cel mai mic cost
- Timp: $O(b^l)$, b – factorul de ramificatie, l – limita de adancime
- Spatiu: O(bl), b – factorul de ramificatie, l – limita de adancime
- Optim: Nu

E. Iterative deepening search

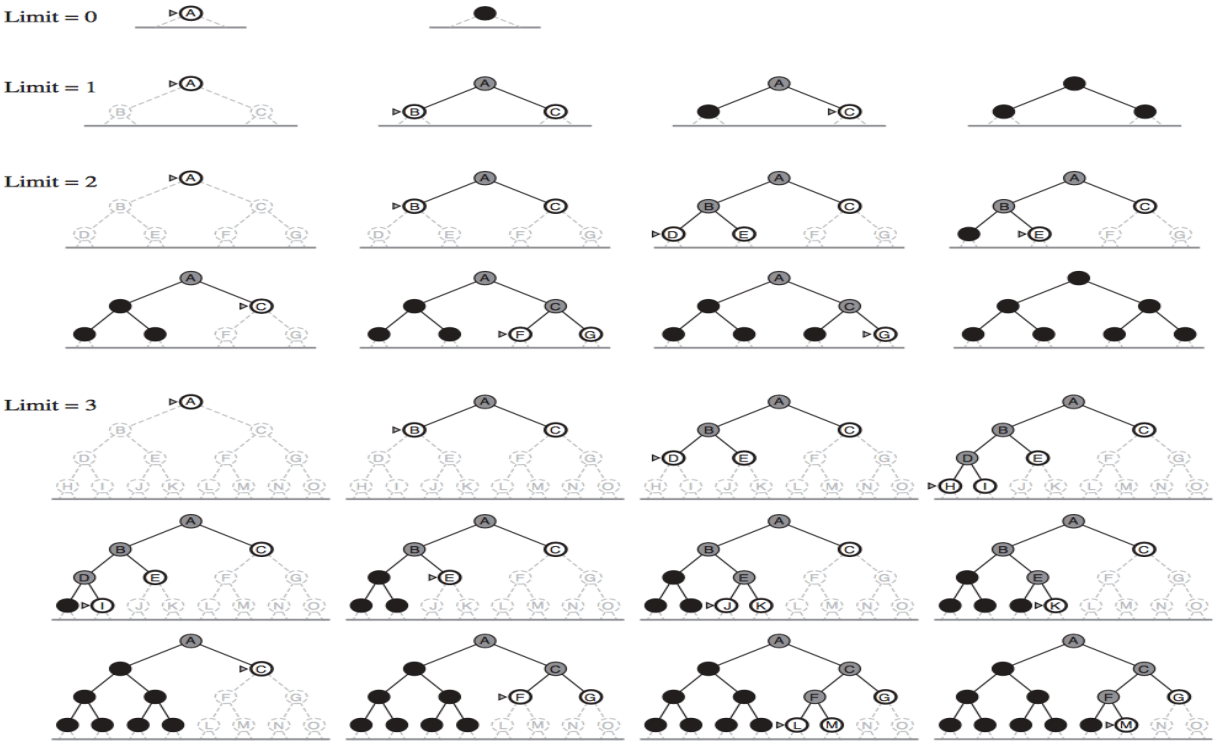
Este o strategie de căutare care rezultă atunci când combinați BFS și DFS, combinând astfel avantajele fiecărei strategii, luând cerințele de memorie modeste ale DFS-ului și caracterul complet și optimalitatea BFS-ului.

IDS funcționează prin căutarea celor mai bune adâncimi de căutare d, astfel, începând cu limita de adâncime 0 aplica un BFS și în cazul în care căutarea nu a reușit se v-a mări limita de adâncime la 1 și încercați din nou un BFS cu adâncime de 1 și așa mai departe până când se ajunge la o adâncime d în cazul în care o tinta este găsita.

Pseudocod

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

Exemplu

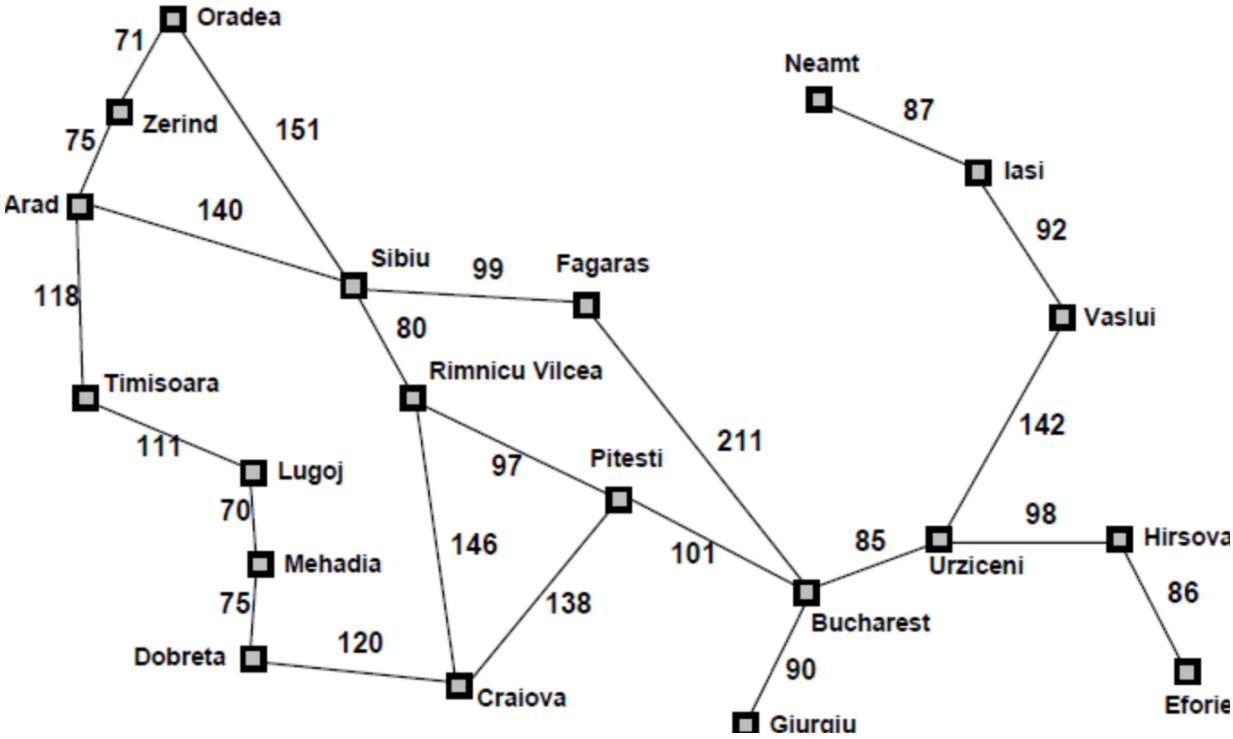


Proprietati ale algoritmului breadth-first search

- Completitudine : Da
- Timp: $(d+1)b^0+db^1+(d-1)b^2+...+b^d = O(b^d)$
- Spatiu: O(bd)
- Optim: Da, daca costul pasului este egal cu 1

Tema:

1. Aplicati fiecare algoritm studiat pe un exemplu.
2. Aplicati UNIFORM COST SEARCH pe graful reprezentat de Harta Romaniei.



3. Implementati 3 din cei 5 algoritmi in JAVA/C#/Python cu interfata grafica.