Laborator 5

IDA*

Deoarece algoritmul A* folosește o coadă, în care salvează întreaga frontieră a arborelui de căutare, se poate ajunge ușor la o cantitate mare de memorie ocupată, iar pentru unele probleme, se depășesc resursele sistemului ducând la eșecul algoritmului.
Prin urmare s-a încercat limitarea memoriei utilizate combinând A* cu modul de lucru al strategiei DepthFirst.

Pseudocod IDA* (varianta recursivă)
Considerăm nodStart nodul de la care începe căutarea. Considerăm că trebuie să afișăm primele N soluții.

```
Se inițializează variabila Limita=f^(nodStart).
Setăm nodul curent ca fiind nodStart.

Creăm o funcție expandeaza(nodCurent, Limita, eventual alti parametri), care are rolul de a expanda nodul curent și descendenții acestuia numai dacă măsura f^ a acestora nu depășește limita. Funcția va returna și noua limită de cost pentru expandare care va fi cel mai mic f^ întâlnit pentru nodurile care nu au putut fi expandate.
Considerăm variabila nodCurent ca fiind nodul curent de expandat.
Dacă nodul curent are măsura f^ > Limita atunci:
      Nu expandăm nodul, ci doar returnăm măsura f^ a acestuia
Altfel (dacă nodul curent are măsura f^ <= Limita):
      Dacă este nod scop, și drumul până la el nu a mai fost afișat anterior (pentru o Limita mai mică), atunci afișăm drumul soluție (de la nodStart până la el) și decrementăm numărul de soluții de afișat.
      Expandăm nodul curent, obținând lista de succesori LS.
      Dacă există succesori:
            Pentru fiecare succesor din LS:
                  Apelăm Lim_succesor = expandeaza(succesor, Limita)
            Calculăm minimul pentru limitele returnate pentru toți succesorii, returnăm acel minim
      Dacă nu există succesori, returnăm N_MAX, un număr foarte mare (mai mare decât orice f^ al oricărui nod).

Repetitiv apelăm funcția Limita=expandeaza(nodStart, Limita), (setând mereu noua limită la valoarea returnată de funcție).
```

```
Ne oprim fie în cazul în care Limita ajunge să fie N_MAX (înseamnă că numărul
de soluții din graf e mai mic decât numărul de soluții care a fost cerut) fie
când s-a afișat numărul de soluții cerute.
```

Problema 8-puzzle

Într-o cutie pătratică (văzută ca o matrice de dimensiune 3x3) sunt 8 plăcuțe, numerotate de la 1 la 8 și un loc liber. Plăcuțele sunt inițial amestecate, ca în imagine. Plăcuțele pot fi mutate pe linie sau coloană doar în spațiul liber (dacă e vecin cu ele). Scopul este să ajungem cu plăcuțele ordonate crescător ca în a doua imagine

Exerciții.
1) Folosiți clasele Graf și Nod implementate într-o cerință anterioară pentru algoritmul A*. Realizați următoarele cerințe pentru problema 8-puzzle.

a) Se va citi dintr-un fișier starea inițială (nu e necesar să citim și starea finală, deoarece este mereu aceeași). Starea inițială va fi reprezentată astfel: fiecare linie de plăcuțe va fi reprezentată de o linie în fișier cu numerele plăcuțelor. Locul liber va fi simbolizat prin cifra 0. Numerele vor fi separate prin câte un spațiu. O stare va fi memorată ca matrice (o listă de liste cu numerele plăcuțelor și 0 pentru locul gol).
b) Scrieți o metodă valideaza() în clasa Graf care verifică dacă fișierul dat e valid:
    - S-a citit o matrice corectă (este o matrice pătratică de dimensiune 3x3)
    - Numerele din matrice sunt toate cele de la 0 la 8 inclusiv.
    - Daca din starea inițială se poate ajunge într-o stare scop. Pentru asta, numărul de inversiuni din matrice trebuie să fie par. Pentru a defini numărul de inversiuni, scriem matricea desfășurată ca un vector, concatenând liniile între ele. Numărul de inversiuni este numărul de perechi de numere (ni,nj) nenule (diferite de locul gol) aflate pe poziția i, respectiv j în vector, cu ni>nj și i<j
c) Modificați funcția scop() astfel încât să verifice că informația nodului curent este egală cu scopul cerut.

Exemplu de fișier de input:

```
2 6 4
7 8 1
5 0 3
```

Un fișier input cu drum mai scurt:

```
1 0 3
4 2 5
7 8 6
```

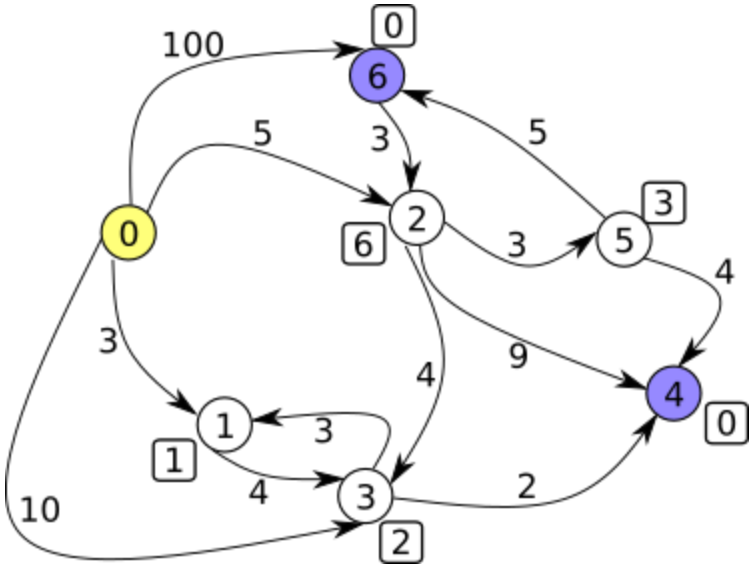
2) Modificați funcția succesori(nod) astfel încât să genereze toți succesorii valizi (a căror informație nu se repetă în istoricul lor), pentru această problemă. Creați două obiecte de tip Nod cu informații diferite și afișati succesorii pentru ele, verificând corectitudinea informațiilor.
Rulați cu BF și DF problema și verificați soluțiile din output.
Pentru rularea cu A* cerută mai jos se consideră costul pe o mutare egal cu 1.

3) Rezolvați problema 8-puzzle folosind A* urmând cerințele:

- Considerați costul pe orice mutare egal cu numărul plăcuței care se mută.
- Implementați funcția estimeaza_h() corespunzătoare problemei, funcția trebuind să returneze o estimație admisibilă pentru fiecare nod. Funcția va avea și un parametru numit euristică prin care se va alege tipul de euristică returnat. Funcția trebuie să admită 4 valori pentru euristică (primele 3 sunt admisibile):
    a. "banala" - caz în care va returna costul minim pe o mutare daca starea nu e scop și 0 dacă e scop
    b. "euristica mutari" - caz în care va returna un număr de mutări (mai mic sau egal decât numărul real de mutări astfel încât să ajungem de la starea curentă la cea mai apropiată stare scop) și 0 dacă starea curentă e scop. Un astfel de număr de mutări e dat de np= "câte plăcuțe nu sunt la locul lor față de pozițiile din stare finală"
    c. "euristica mutari cost" - caz în care va returna suma costurilor pentru mutarea plăcuțelor care nu sunt la locul lor.
    d. "euristica manhattan" - considerăm dm[i] distanța Manhattan pentru plăcuța i de la poziția curentă la poziția în starea finală. Funcția va returna suma acestor distanțe.
    e. "euristica manhattan costuri" - Ca și mai sus, considerăm dm[i] distanța Manhattan pentru plăcuța i de la poziția curentă la poziția în starea finală. Funcția va returna costul total al mutării plăcuțelor pe aceste distanțe, adunând pentru fiecare plăcuță i valoarea i*dm[i]
    f. "euristica neadmisibila" - caz in care se vor returna valori astfel încât estimația pentru nod să fie neadmisibilă.

  Exemplu de apelare: graf.estimeaza_h(nod, euristica="banala")

4) Implementați IDA* pentru graful:


0
100
6
5
3
5
0
2
3
6
5
3
4
3
9
4
4
0
1
3
1
4
2
3
10
2