Aplicarea algoritmului A* in probleme

Dupa ce ati implementat algoritmul A* si v-ati asigurat ca functioneaza corect pe exemplul didactic cu graful din desen, pentru restul problemelor copiati intreg codul in alt fisier si modificati doar anumite detalii specifice de la caz la caz.

Structura in mare a claselor folosite va ramane aceeasi (dar pot aparea sau disparea anumite atribute, metode), la fel si functia principala „a_star” care implementeaza algoritmul (verificati daca necesita mici modificari in functie de ce ati schimbat in structura claselor). Pentru fiecare problema trebuie sa va ganditi cum implementati detaliile specifice acelui joc, anumite metode si atribute ale obiectelor din clasele folosite. Adaptati si functiile folosite pentru afisari daca e cazul.

Am folosit in tabel ca exemplu jocul de Sudoku, pentru a fi mai clar la ce se refera intrebarile. (Se da o matrice de 9x9 casute, unele contin deja cifre. Se cere sa se completeze casutele libere cu cifre de la 1 la 9 astfel incat pe fiecare linie, coloana, si in cele 9 cadrane de 3x3 casute sa apara o singura data fiecare dintre cifre.)

Intrebari:	**Exemplu raspunsuri pentru joc Sudoku:**
Q1) a) Ce informatii nu se schimba pe durata unui joc, dar pot varia pentru jocuri diferite? **b)** In ce consta o configuratie de joc? Ce informatii trebuie incluse in nodul curent (cele care se pot schimba atunci cand se executa o mutare a jocului)?	**A1) a)** Clasa Nod poate avea atributele L_MATRICE=9, L_CADRAN=3, NR_MAX=9 daca dorim o rezolvare mai generala care sa poata fi adaptata si pentru matrice de alte dimensiuni. **b)** Obiectele din clasa Nod au ca atribut „info” matricea de joc, de dimensiune L_MATRICE x L_MATRICE, care contine numere intre 1 si NR_MAX, sau 0 pentru casutele libere.
Q2) a) Care este nodul de start (configuratia initiala a jocului)? **b)** Cum aflam daca s-a terminat jocul (care sunt configuratiile / nodurile scop)?	**A2) a)** Obiectele din clasa Problema au ca atribut „nod_start” o matrice cu cateva numere deja completate si 0 in rest. **b)** In metoda „test_scop” din clasa NodParcurgere se verifica daca intreaga matrice este completata, adica nu are 0-uri (am avut grija anterior sa nu adaugam numere care sa creeze conflict pe linie, coloana sau in cadran).
Q3) a) In ce consta o mutare a jocului (prin ce difera configuratia „tata” fata de cea „fiu” in arborele de cautare)? **b)** Cum gasim toti succesorii posibili pentru configuratia actuala?	**A3) a)** Un 0 din matrice se inlocuieste cu un numar x (intre 1 si NR_MAX), daca x nu apare deja pe acea linie, coloana sau in acel cadran. **b)** Pentru fiecare casuta cu 0 din matricea nodului „tata”, pentru fiecare valoare posibla a lui x, daca nu apare conflict pentru linie, coloana sau cadran, obtinem cate o noua matrice pentru cate un nod „fiu”. Facem asta in metoda „expandeaza” din clasa NodParcurgere.

Q4) Ce valori folosim pentru functia g (ce cost au muchiile arborelui de cautare)?	**A4)** La Sudoku vom considera ca orice mutare a jocului are costul 1 (toate muchiile au g=1). In metoda „expandeaza” din clasa NodParcurgere se obtine o lista de tupluri cu 2 elemente. Primul element din tuplu va fi un obiect al clasei Nod („fiul” curent), iar al doilea element din tuplu va fi costul muchiei dintre „tata” si acest „fiu” (cost 1 de fiecare data pt Sudoku).

Q5) Ce valori folosim pentru functia euristica $\hat{h}$?
 	
Explicatie pt functia euristica:
● Euristica $\hat{h}$ reprezinta numarul minim de mutari necesare din configuratia curenta pana la finalul jocului (deci functia trebuie sa descreasca din nodul „tata” in nodul „fiu”, pe masura ce ne apropiem de un nod scop).
● Functia $\hat{h}$ aleasa trebuie sa sub-aproximeze numarul propriu-zis de pasi (functia ***h***) care vor fi facuti de joc, niciodata nu are voie sa-l depaseasca, sa supraestimeze.
	Obs: Pentru unele jocuri putem gandi diferite functii euristice $\hat{h}$, care sub-aproximeaza mai aproape sau mai departe fata de functia ***h***.
	
	A5) Pentru Sudoku, daca o mutare a jocului inseamna sa inlocuim un 0 cu un numar nenul, atunci euristica reprezinta cate 0-uri mai sunt in configuratia curenta (*exact* atatea mutari vor mai fi necesare pana se termina jocul).
	
	Explicatii pt implementare:
	□ In constructorul („**__init__**”) clasei Nod trimiteti ca parametru doar „info” (fara „h”), iar in interior aveti
	`self.h = self.fct_h()`
	□ Si definiti in clasa Nod metoda **fct_h**
	`def fct_h(self):`
	`  M = self.info` # matricea de joc
	`  h = .....` # numar cate casute cu 0 sunt in matricea M
	`  return h`
	□ In metoda „expandeaza” din clasa NodParcurgere, dupa ce ati obtinut Mat = matricea „fiu”, adaugati in lista de succesori tuplul `(Nod(Mat),1)`.

Clarificări despre funcția euristică $\hat{h}$:

□ Pe caz general, euristica $\hat{h}$ reprezintă ***costul minim*** necesar pentru a ajunge din configurația curentă până în configurația finală a jocului. Deci calculul lui $\hat{h}$ depinde de funcția ***g*** (care sunt costurile muchiilor/mutărilor pentru acel joc.)

□ Mai sus, în tabel, am zis că “euristica $\hat{h}$ reprezintă ***numărul minim de mutări*** necesare pentru a ajunge din configurația curentă până în configurația finală a jocului”. Acest lucru este corect pentru acele probleme în care alegem toate mutările/muchiile de cost ***g = 1***.

□ Pentru ca algoritmul A* să funcționeze și să găsească *sigur* drumul de cost minim de la nodul de start la un nod scop, trebuie *obligatoriu* ca funcția euristică $\hat{h}$ să îndeplinească

"condiția de admisibilitate"**:* funcția $\hat{h}$ să nu supraestimeze niciodată valoarea efectivă dată de funcția ***h, adică $\hat{h}(n) \leq h(n)$ pentru toate nodurile *n* din arborele de căutare.

- În plus, trebuie verificat dacă euristica aleasă îndeplinește sau nu ***"condiția de consistență"***, adică trebuie verificat dacă $\hat{h}(tata) \leq cost\_muchie(tata, fiu) + \hat{h}(fiu)$, adică dacă $\hat{f}$ descrește monoton pe măsură ce ne îndepărtăm de nodul de start. Atenție, algoritmul funcționează chiar și dacă această condiție nu este îndeplinită.
- *Mai multe detalii și explicații despre aceste două condiții citiți în documentele de curs.*

Cateva observatii (despre implementarea problemelor):

(1) La exemplul didactic (graful desenat) din fisierul „Algoritmul A-star", se stia de la inceput *intregul graf* al jocului (dat in clasa Problema prin lista de noduri si lista de muchii). ***Atentie***, la celelalte probleme (Sudoku, pb blocurilor, pb 8-puzzle, pb canibali si misionari etc.) *NU stiti de la inceput intregul graf*.

- Stiti **nod_start** dat ca atribut in constructorul clasei Problema.
- Stiti cand se termina jocul (la Sudoku stiti ***conditia*** pe care o verificati in metoda „test_scop" din clasa NodParcurgere; iar la celelalte 3 probleme stiti chiar **nod_scop**, dat ca atribut in constructorul clasei Problema).
- Restul nodurilor le obtineti la fiecare pas in metoda „expandeaza" din clasa NodParcurgere.

(2) Clasa Problema: constructorul si initializare atribute obiect.
Dati ca parametri strictul necesar de informatie, iar valorile pentru celelalte atribute ale obiectului le ***deduceti in cod*** din informatia primita.

a) La **pb blocurilor**

```
class Problema:
   def __init__(self, start = [['a'], ['c', 'b'], ['d']],
                      scop = [['b', 'c'], [], ['d', 'a']]):

      self.nod_start = Nod(start, float("inf"))
      self.nod_scop = scop

      self.nr_stive = len(start) # N=3 in enunt
      self.nr_cuburi = sum([len(stiva) for stiva in start]) # M=4 in enunt
```

Apoi in main puteti folosi constructorul fara parametri (va folosi valorile implicite)

```
p = Problema()
```

sau varianta cu alti parametri

```
p = Problema([['c','b','a'],['d','e'],[]],  [[],[],['e','d','c','b','a']])
```

Apoi

```
NodParcurgere.problema = p
a_star()
```

b) La **pb 8-puzzle**

```
class Problema:
   def __init__(self, start = [2,4,3, 8,7,5, 1,0,6],
                      scop = [1,2,3, 4,5,6, 7,8,9]):

      self.nod_start = Nod(start, float("inf"))
      self.nod_scop = scop

      # daca considerati necesar,
      # calculati lungimea listei sau latura matricei (pt lista de liste)
      self.N = len(start) # sau radical din len(start)
                          #   daca avem 9 elem si vrem latura 3
```

Apoi puteti folosi constructrul in diverse moduri:

```
p = Problema()
p = Problema(start = [...])
p = Problema(start = [...], scop = [...])
```

c) La **pb canibali si misionari**

```
class Problema:
   def __init__(self, N=3, M=2, mal="est"):
      nod_E = (0,0,N,N,"est")
      nod_V = (N,N,0,0,"vest")

      start = nod_E if mal=="est" else nod_V
      scop = nod_V if mal=="est" else nod_E

      self.nod_start = Nod(start, float("inf"))
      self.nod_scop = scop
```

Atentie, obiectele din clasa de mai sus vor avea atributele (*cele initializate in parametri si cele initializate in interiorul constructorului*):

`N` (misionari = canibali = N)

`M` (nr locuri in barca)

`mal` ("est" sau "vest", malul de unde pleaca barca la inceputul problemei)

`nod_start` (obiect de tip Nod, cu atributele info si h)

`nod_scop` (contine doar informatia nodului scop)

Apoi puteti folosi constructrul in diverse moduri, printre care:

```
p = Problema()
```

```
p = Problema(N=5)
p = Problema(N=5, M=3, mal="vest")
```

(3) □ Dacă pentru calculul funcției euristice $\hat{h}$ este nevoie doar de configurația curentă, atunci calculul se face în clasa Nod, cum am explicat în tabel pentru problema Sudoku. □ Dacă în schimb avem nevoie să comparăm configurația curentă cu configurația scop sau avem nevoie de alte informații din clasa Problemă, atunci calculul euristicii $\hat{h}$ se va face în clasa NodParcurgere (detaliata mai jos). Adica in metoda "expandeaza" din clasa NodParcurgere, pentru "`self.nod_graf_info`" (pentru configuratia curenta de joc, "tatal") generam toate mutarile valide posibile, adica gasim toti succesorii. Pentru fiecare succesor (configuratie "fiu"), apelam "`h_succesor = self.euristica_h(succesor)`", calculam "`g_muchie = …`" (costul muchiei de la configuratia "tata" la configuratia "fiu" curenta), apoi "`l_succesori.append((Nod(succesor, h_succesor), g_muchie))`".

```
def euristica_h(self, info):
    # avem "info" (configuratia de joc a nodului
    #               caruia vrem sa-i calculam euristica)
    # si "self.problema.nod_scop" (configuratia scop)
    # sau "self.problema.x", unde "x" erau alte atribute utile
    #                        ale obiectelor din clasa Problema
    h = ... # calculam euristica
    return h
```

(4) La final, cand afisati concluzia, in loc sa folositi functia „**str_info_noduri**” (care afisa toate informatiile din obiectul de tip NodParcurgere, adica inclusiv h, parinte, g, f), creati-va o **alta functie „afisare_simpla”**, care primeste ca parametru (`L`) tot o lista cu obiecte de tip NodParcurgere, dar care sa afiseze doar informatia despre cum arata fiecare configuratie de joc (`for x in L: config = x.nod_graf.info; afisare(config)`), intr-un mod usor de citit si de urmarit care au fost miscarile facute. Intre doua configuratii lasati 1-2 randuri libere. De exemplu:

a) La **pb blocurilor**, aveti `config` o lista de liste, deci in loc de `afisare(config)` scrieti detaliat ca sa afisati fiecare element la locul lui. Adica fie (preferabil) calculati inaltimea maxima a unei stive si afisati stivele „in picioare” aliniate in partea de jos, fie (varianta mai usoara) „culcati” stivele pe dreapta si le aliniati la stanga ecranului elementul cel mai de jos, iar in dreapta varful stivei, cate o stiva pe cate un rand.

b) La **pb 8-puzzle**, aveti `config` o lista de liste, deci in loc de `afisare(config)` scrieti cum afisati frumos matricea de joc, cate o lista pe cate un rand, iar in cadrul fiecarui rand elementele cu spatiu intre ele.

c) La **pb canibali si misionari**, aveti `config` tuplul cu cele 5 informatii, deci in loc de `afisare(config)` scrieti clar sub forma de propozitii:

```
Pe malul de est sunt … misionari si … canibali.
Pe malul de vest sunt … misionari si … canibali.
Barca se afla pe malul de … [est / vest].
```