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)`.

Cateva observatii:

(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) 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].
```

(3) La **pb 8-puzzle** incercati cu inputul **nod_start = [[2, 4, 3], [8, 7, 5], [1, 0, 6]]**, ajunge mai repede la raspuns decat ce va dadusem in poza din fisier (acela dureaza foarte mult).