Problema unui lăcat

Țifui Ioan Alexandru

Cerința 1)

Cerință:

Fișierele de input vor fi într-un folder a cărui cale va fi dată în linia de comanda. În linia de comandă se va da și calea pentru un folder de output în care programul va crea pentru fiecare fișier de input, fișierul sau fișierele cu rezultatele. Tot în linia de comandă se va da ca parametru și numărul de soluții de calculat (de exemplu, vrem primele NSOL=4 soluții returnate de fiecare algoritm). Ultimul parametru va fi timpul de timeout. Se va descrie în documentație forma în care se apelează programul, plus 1-2 exemple de apel.

Implementare:

Pentru apelul unui program se vor da in aceasta ordine parametrii: calea către fișierul de input, calea către fișierul de output, numărul de soluții căutat, timeout-ul în secunde și tipul de euristică folosită (1 pentru euristica banală, 2 pentru prima euristică admisibilă, 3 pentru a doua euristică admisibilă și 4 pentru euristica inadmisibilă). Ultimul parametru nu va fi prezent în cazul algoritmului ucs.

Toate fișierele de input se află în folderul 'inputs', iar cele de output în folderul 'outputs'

De exemplu, daca dorim sa aflam primele 3 soluții folosing algoritmul ucs, citind din fisierul 'input.in' și scriind în fișierul 'output.out' cu timeout de o secundă vom apela:

```
python3 ucs.py inputs/input.in outputs/output.out 3 1
```

Alte exemple de apelare:

```
python3 a_star.py inputs/input.in outputs/output.out 1 1 1
python3 ida_star.py inputs/input.in outputs/output.out 1 1 2
```

Cerința 2)

Cerință:

Citirea din fisier + memorarea starii. Parsarea fișierului de input care respectă formatul cerut în enunț

Implementare:

Folosim metoda 'read_data()' a clasei 'Alg' pentru a citi datele.

```
def read_data(self):
    # citim din fisierul dat ca parametru si instantiem clasa graf
    lista_chei = self.fin.read().strip().split()

    start = [('i', 1)] * len(lista_chei[0])
    scop = [('d', 0)] * len(lista_chei[0])

    self.gr = Graf(Lacat(start, []), Lacat(scop, []), lista_chei)
```

Cerința 3)

Cerință:

Functia de generare a succesorilor

Implementare:

Folosim metoda 'genereaza_succesori' a clasei 'Graf' care se bazeaza pe metoda 'aplica_cheie' a clasei 'Lacat'.

```
def aplica_cheie(self, cheie):
# returneaza un lacat nou
    if len(cheie) != len(self.incuietori):
        print("Numar diferit de zone si incuietori")
        exit(1)

    new_incuietori = []
    for i in range(len(cheie)):
        stare = self.incuietori[i][0]
        nr_incuieri = self.incuietori[i][1]
```

```
            if cheie[i] == 'd':
                if self.incuietori[i][0] == 'i':
                    if self.incuietori[i][1] == 1:
                        stare = 'd'
                        nr_incuieri = 0
                    else:
                        nr_incuieri -= 1
            elif cheie[i] == 'i':
                stare = 'i'
                nr_incuieri += 1

            new_incuietori.append((stare, nr_incuieri))

        return Lacat(new_incuietori, cheie)

        ...

    def genereaza_succesori(self, nod_curent):
        # aplicam cheile la nodul curent pentru a obtine noduri noi
        return [nod_curent.aplica_cheie(k) for k in self.lista_chei]
```

Cerința 4)

Cerință:
Calcularea costului pentru o mutare

Implementare:
Costul unei mutari este de fiecare dată 1 în cazul acestei probleme.

Cerința 5)

Cerință:
Testarea ajungerii în starea scop (indicat ar fi printr-o funcție de testare a scopului)

Implementare:
Deoarece există o singură stare de scop (cea în care lacătul este complet descuiat) testarea se reduce la o simple verificare.

Pentru aceasta am supraîncărcat operatorul '==' al clasei 'Lacat'

```
def __eq__(self, other):
    return self.incuietori == other.incuietori
```

Cerința 6)

Cerință:

4 euristici: - banala

- doua euristici admisibile posibile (se va justifica la prezentare si in documentație de ce sunt admisibile)

- o euristică neadmisibilă (se va da un exemplu prin care se demonstrează că nu e admisibilă). Atenție, euristica neadmisibilă trebuie să depindă de stare (să se calculeze în funcție de valori care descriu starea pentru care e calculată euristica).

Implementare:

- euristica banală este $\widehat{h}(nod) = 0$ pentru fiecare nod (simiar ucs)

- prima euristică admisibilă considerată este $\widehat{h}(nod) = min(i(x))$ unde x reprezintă o încuietoare a lacătului reprezentat de nod, iar $i(x)$ reprezintă de câte ori aceasta a fost încuiată; pentru ca din nodul curent sa ajungem în nodul scop trebuie să descuiem încuietoarea cu $i(x)$ minim, deci vom face cel puțin $\widehat{h}(nod)$ pași, de unde rezultă că euristica este într-adevăr admisibilă

- a doua euristică admisibilă considerată este $\widehat{h}(nod) = max(i(x))$, cu notațiile anterior discutate; aceasta este admisibilă din același motiv pentru care cea dinainte este admisibilă

- euristica inadmisibilă discutată este $\widehat{h}(nod) = \sum i(x)$, cu notațiile anterior discutate; să presupunem că avem lacătul $[(i,1),(i,1),(i,1)]$ și cheia $[d,d,d]$, atunci $\widehat{h}(nod) = 3$, dar $h(nod) = 1 < \widehat{h}(nod))$, deci euristica este într-adevăr inadmisibilă

Cerința 7)

Cerință:

crearea a 4 fisiere de input cu urmatoarele proprietati:

a. un fisier de input care nu are solutii

b. un fisier de input care da o stare initiala care este si finala (daca acest lucru nu e realizabil pentru problema, aleasa, veti mentiona acest lucru, explicand si motivul).

c. un fisier de input care nu blochează pe niciun algoritm și să aibă ca soluții drumuri lungime micuță (ca să fie ușor de urmărit), să zicem de lungime maxim 20.

d. un fisier de input care să blocheze un algoritm la timeout, dar minim un alt algoritm să dea soluție (de exemplu se blochează DF-ul dacă soluțiile sunt cât mai ”în dreapta” în arborele de parcurgere) dintre ultimele doua fisiere, cel putin un fisier sa dea drumul de cost minim pentru euristicile admisibile si un drum care nu e de cost minim pentru cea euristica neadmisibila

Implementare:

a. fișierul 'no_sol.in'

b. nu este posibil pentru problema noastră

c. fișierul 'simplu.in'

d. fișierul 'tricky.in'

Cerința 8)

Cerință:

Pentru cele NSOL drumuri(soluții) returnate de fiecare algoritm (unde NSOL e numarul de soluții dat în linia de comandă) se va afișa:

- lungimea drumului

- costului drumului

- timpul de găsire a unei soluții (atenție, pentru soluțiile de la a doua încolo timpul se consideră tot de la începutul execuției algoritmului și nu de la ultima soluție)

- numărul maxim de noduri existente la un moment dat în memorie

- numărul total de noduri calculate (totalul de succesori generati; atenție la DFI și IDA* se adună pentru fiecare iteratie chiar dacă se repetă generarea arborelui, nodurile se vor contoriza de fiecare dată afișându-se totalul pe toate iterațiile)

Implementare:

Fișierele de output corespund cerinței, cu mențiunea că în cazul problemei noastre, lungimea și costrul drumului se confundă.

Cerința 9)

Cerință:

Afisarea in fisierele de output in formatul cerut

Implementare:

Fișierele de output corespund cerinței, cu mențiunea că în cazul problemei noastre, lungimea și costrul drumului se confundă.

Cerința 10)

Cerință:

Validări și optimizari. Veți implementa elementele de mai jos care se potrivesc cu varianta de temă alocată vouă:

- găsirea unui mod de reprezentare a stării, cât mai eficient

- verificarea corectitudinii datelor de intrare

- găsirea unor conditii din care sa reiasă că o stare nu are cum sa contina in subarborele de succesori o stare finala deci nu mai merita expandata (nu are cum să se ajungă prin starea respectivă la o stare scop)

- găsirea unui mod de a realiza din starea initială că problema nu are soluții. Validările și optimizările se vor descrie pe scurt în documentație.

Implementare:

- nodul din graful problemei este reprezentat de clasa 'Lacat' în care menținem stările încuietorilor și ultima cheie aplicată lactătului; se va mai folosi clasa 'NodParcurgere' cu semnificația de la laborator

- singura problemă era ca lungimile cheilor să fie de lungimi diferite, caz în care nu putem ști care este cea corectă, astfel am considerat că datele de intrare sunt corecte

- știm că problema nu are soluții dacă există o încuietoare care să nu fie deschisă de nicio cheie (adică dacă există o poziție pentru care nicio cheie din cele date nu conține caracterul 'd' pe acea poziție); în afară de această observație care poate fi aplicată stării inițiale, nu am observat nicio condiție care să ne garanteze că niciunul dintre succesorii unei stări nu este stare scop

Cerința 11)

Cerință:

Comentarii pentru clasele și funcțiile adăugate de voi în program (dacă folosiți scheletul de cod dat la laborator, nu e nevoie sa comentați și clasele existente). Comentariile pentru funcții trebuie să respecte un stil consacrat prin care se precizează tipul și rolurile parametrilor, cât și valoarea returnată (de exemplu, reStructured text sau Google python docstrings).

Implementare:

Am folosit scheletul de la laborator puțin adaptat, dar am adăugat comentarii pentru a face codul mai ușor de citit.

Cerința 12)

Cerință:

Documentație cuprinzând explicarea euristicilor folosite. În cazul euristicilor admisibile, se va dovedi că sunt admisibile. În cazul euristicii neadmisibile, se va găsi un exemplu de stare dintr-un drum dat, pentru care h-ul estimat este mai mare decât h-ul real. Se va crea un tabel în documentație cuprinzând informațiile afișate pentru fiecare algoritm (lungimea și costul drumului, numărul maxim de noduri existente la un moment dat în memorie, numărul total de noduri). Pentru variantele de A* vor fi mai multe coloane în tabelul din documentație: câte o coloană pentru fiecare euristică. Tabelul va conține datele pentru minim 2 fișiere de input, printre care și fișierul de input care dă drum diferit pentru euristica neadmisibilă. În caz că nu se găsește cu euristica neadmisibilă un prim drum care să nu fie de cost minim, se acceptă și cazul în care cu euristica neadmisibilă se obțin drumurile în altă ordine decât crescătoare după cost, adică diferența să se vadă abia la drumul cu numărul K, K ¿ 1). Se va realiza sub tabel o comparație între algoritmi și soluțiile returnate, pe baza datelor din tabel, precizând și care algoritm e mai eficient în funcție de situație. Se vor indica pe baza tabelului ce dezavantaje are fiecare algoritm.

Implementare:

Pentru următorul input avem următorul tabel de rezultate:

```
iid
did
gdg
```

Algoritm	Euristică	Cost	Timp de execuție	Total noduri	Maxim noduri
UCS	N/A	3	0.0023	42	26
A*	banală	3	0.0028	42	26
A*	admisibilă 1	3	0.0031	42	26
A*	admisibilă 2	3	0.00007	9	5
A*	inadmisibilă	3	0.00004	9	5
A* optimizat	banală	3	0.0001	19	19
A* optimizat	admisibilă 1	3	0.0002	19	19
A* optimizat	admisibilă 2	3	0.00006	8	8
A* optimizat	inadmisibilă	3	0.00006	8	8
IDA*	banală	3	0.001	44	4
IDA*	admisibilă 1	3	0.002	40	4
IDA*	admisibilă 2	3	0.001	23	4
IDA*	inadmisibilă	3	0.00006	9	4

Pe baza acestui tabel putem trage concluzia ca euristica admisibila 1 nu imbunatateste cu mult performanta algoritmilor IDA* și A* față de UCS. Performanțele cresc abia de la a doua euristică admisibilă. Euristica inadmisibilă a dat răspunsuri valide pentru acest input. De asemenea remarcam ca IDA* a avut performantele cele mai mari.

Pentru următorul input avem următorul tabel de rezultate:

```
iiiddddgggg
dddggggddgg
dddgggggggdd
```

Algoritm	Euristică	Cost	Timp de execuție	Total noduri	Maxim noduri
UCS	N/A	3	0.0003	29	16
A*	banală	3	0.0003	29	16
A*	admisibilă 1	3	0.0001	29	16
A*	admisibilă 2	3	0.0008	12	6
A*	inadmisibilă	4	0.0009	10	5
A* optimizat	banală	3	0.0003	16	16
A* optimizat	admisibilă 1	3	0.0003	16	16
A* optimizat	admisibilă 2	3	0.0009	10	10
A* optimizat	inadmisibilă	4	0.0008	9	9
IDA*	banală	3	0.0001	27	4
IDA*	admisibilă 1	3	0.0001	23	4
IDA*	admisibilă 2	3	0.0001	21	4
IDA*	inadmisibilă	10	0.0001	19	11

Pe baza tabelului putem vedea ca toți algorimtii dau un răspuns greșit pe cazul când folosesc euristica inadmisibilă. Totuși IDA* este de departe cel mai eronat.