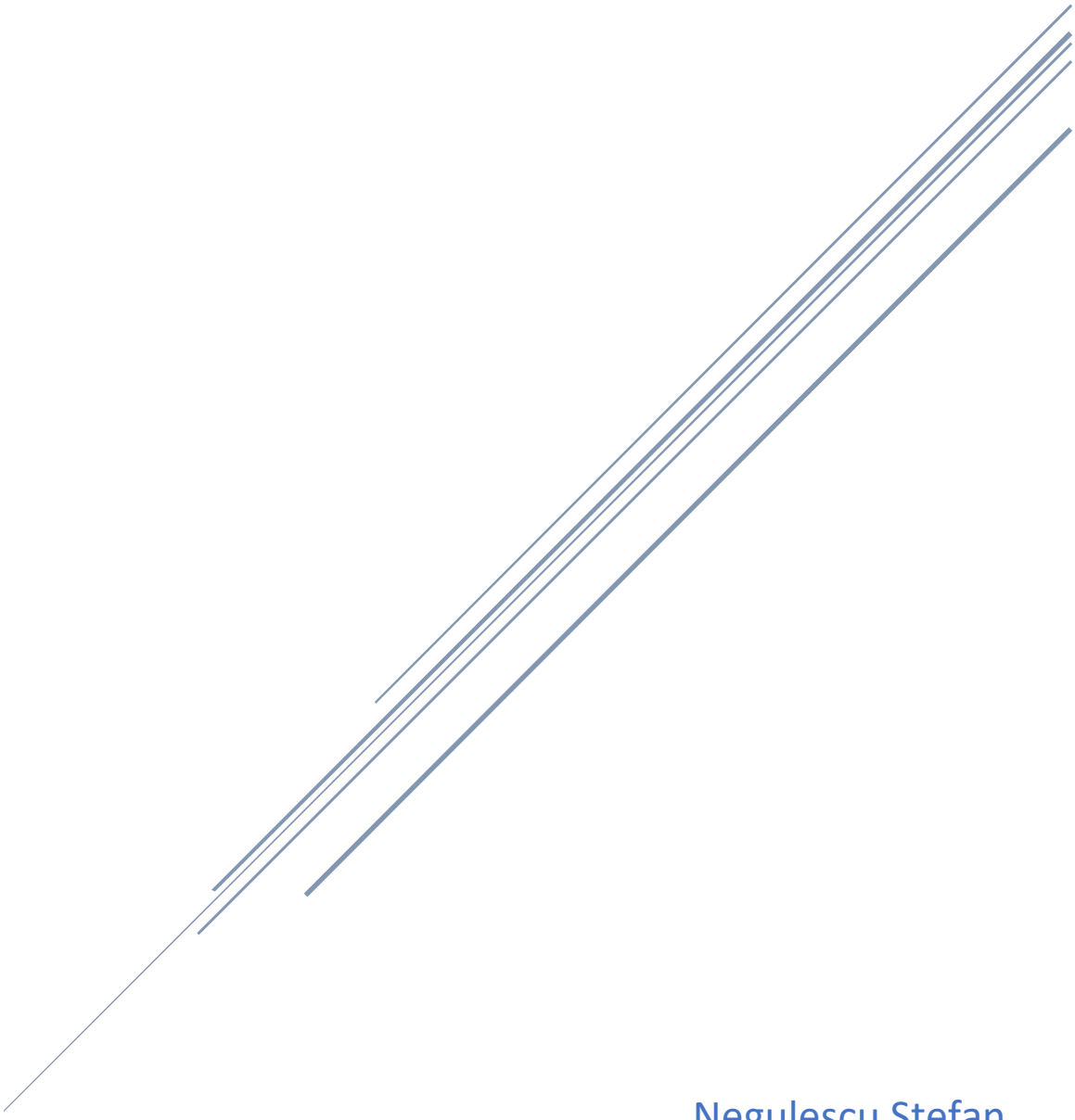
Documentatie

Problema decuparii



Negulescu Stefan
232

Cuprins

Problema

Context.

Se considera un grid (matrice) de dimensiuni N×M (N linii si M coloane). In grid avem litere de la a la z. Dorim prin tăieri succesive de linii și coloane să ajungem la o anumită stare scop. De exemplu, pentru starea:

Stări și tranziții. Cost.

Mutarile posibile sunt

- taierea a X (X≥1) coloane (cost: 1+(k/(nr coloane taiate)), unde k reprezinta numărul, din intreaga zona decupata, a perechilor de vecini care sunt diferiti intre ei; evident fara a lua si simetricele perechilor; de exemplu, daca am gasit perechea {(0,0), (0,1)} nu vom lua si perechea {(0,1), (0,0)}).
- taierea a X (X≥1) linii (cost: (nr coloane)/(nr linii taiate))

De exemplu, pentru starea:

```
bbx
aba
aab
ddd
```

decidem ca ar trebui eliminate primele doua coloane. Daca am elimina intai prima coloana, costul ar fi 1+2/1=3, deoarece avem vecini diferiti doar in perechile {(0,0),(1,0)} si {(2,0),(3,0)} si am taiat doar o coloana, iar costul celei de-a doua coloane este 1+2/1=3 (perechile {(1,1),(2,1)},{(2,1),(3,1)}), deci in total costul taierii separate ar fi 6. Dar daca am decupa ambele coloane in acelasi timp, costul ar fi 1+5/2=3.5 (perechile de vecini {(0,0),(1,0)}, {(1,0),(1,1)}, {(1,1),(2,1)}, {(2,0),(3,0)}, {(2,1),(3,1)}, deci in numar de 5, si am taiat doua coloane).

Prin urmare, in acest caz e mai eficient sa taiem impreuna coloanele

Dar pentru cazul de mai jos in care vrem sa taiem tot primele 2 coloane:

```
abx
aba
abb
abd
acd
```

Costul taierii separate a primeia este 1+0/1=1. Costul taierii celei de-a doua este 1+1/1=2, deci costul total al celor doua taieri este 2. Daca taiem ambele coloane, costul va fi 1+6/2=4 deci in acest caz este mai eficient sa taiem separat coloanele.

Fisierul de intrare

Fisierul de intrare va contine gridul initial si starea scop, cu o linie vida intre ele:

```
ddaabbb
aaaffxc
aabccdc
adddaab
```

abb
axc

Fișier output.

Pentru starea:

K=3
ddaabbb
aaaffxc
aabccdc
adddaab

Consideram urmatoarea stare finala:

abb
axc

Un drum posibil este cel de mai jos (nu neaparat cel de cost minim):

```
1)
daabbb
aaffxc
abccdc
addaab

Am eliminat coloanele 1,2,3

2)
dabb
aaxc
abdc
adab

Am eliminat coloana 0

3)
abb
axc
bdc
dab

Am eliminat linile 2,3

4)
abb
axc
```

Euristica

1. **Euristica banala** – Returneaza 0 daca starile sunt egale, 0 altfel.
2. **Euristica e1** – Returneaza 0 daca starile sunt egale, altfel lui nr I se adauga 1 daca mai trebuie eliminate coloane si/sau I se adauga raportul dintre numarul de coloane din scop (nr min de coloane) inmultit cu numarul de coloane care trebuie eliminate / nr de coloane care trebuie eliminate. Aceasta este admisibila deoarece costul va fi cel putin 1 in cazul in care mai trebuie eliminate coloane, iar in cazul liniilor, costul minim care poate exista este cel prezentat, raportul fiind minim deoarece numaratorul este minimul de coloane (nr-ul de coloane din scop), iar numitorul va fi numarul total de linii care trebuie eliminate.
3. **Euristica e2** – Aceasta, in plus de e1 trateaza cazurile in care numarul de linii sau numarul de coloane este acelasi.
    a. Cand numarul de linii este acelasi, returneaza raportul descries mai sus. Am demonstart la (2.) ca este admisibila aceasta ramura.
    b. Cand numarul de coloane este acelasi returneaza 1 + raportul dintre numarul minim de elemente vecine diferite pe fiecare coloana inmultit cu diferenta de coloane fata de scop / aceeasi diferenta. Aceasta este admisibila deoarece 1 va fi adaugat obligatoriu in cazul in care mai trebuie eliminate coloane, iar numaratorul este valoarea minima a elementelor vecine inegale pe coloane * numarul de coloane care trebuie eliminate / nr coloane. Astfel, acesasta valoare nu va depasi costul real.
4. **Euristica neadmisibila** – Returneaza numarul de linii + numarul de coloane care mai trebuie eliminate pentru a avea m si n ale starii scop. Este neadmisibila deoarece pot exista mutari cu costul 1 in

care sunt eliminate toate liniile/coloanele iar aceasta euristica va returna numarul total al liniilor/coloanelor care trebuie eliminate, rezultand un numar mai mare decat cel real.

Exemplu euristica neadmisibila:

Costul estimat este 7, desi, in realitate, costul va fi doar 3.

Fisier input:

```
aaaaaaaa
aaaaaaaa
aaaaaaaa

aa
aa
```

Euristica neadmisibila:

```
1:
['a' 'a' 'a' 'a' 'a' 'a' 'a' 'a']
['a' 'a' 'a' 'a' 'a' 'a' 'a' 'a']
['a' 'a' 'a' 'a' 'a' 'a' 'a' 'a']
g: 0  h: 7
--------------
```

Euristica admisibila:

```
1:
['a' 'a' 'a' 'a' 'a' 'a' 'a' 'a']
['a' 'a' 'a' 'a' 'a' 'a' 'a' 'a']
['a' 'a' 'a' 'a' 'a' 'a' 'a' 'a']
g: 0  h: 3.0
--------------
2:
['a' 'a']
['a' 'a']
['a' 'a']
g: 1.0  h: 2.0
--------------
3:
['a' 'a']
['a' 'a']
g: 3.0  h: 0
--------------
Cost: 3.0  Length: 3
```

Statistici

Input 1: o solutie

```
abcdefg
abcdefg
abcdefg
abcdefg
abcdefg

af
af
```

Algoritm\Date	Lungime	Cost	Nod.tot	Nod.max	Timp (ms)
UCS	5	3.(6)	41689	38645	1718
A*/ e ban	5	3.(6)	13825	13117	619
A*/ e1	5	3.(6)	3245	3182	211
A*/ e2	5	3.(6)	3245	2968	149
A*/ neadm	4	4.(3)	217	194	11
A*opt/ e ban	5	3.(6)	1893	1668	2709
A*opt/ e1	5	3.(6)	1896	1550	2700
A*opt/ e2	5	3.(6)	1896	1550	2424
IDA*/ e ban	5	3.(6)	2385793	1194694	2000
IDA*/ e1	5	3.(6)	25969	13810	403
IDA*/ e2	5	3.(6)	355169	177410	553

Input 2: o solutie

```
amasfaqwan
dadssafeme
aodaveucwv
oofavrkpvm

ssaeme
avecwv
```

Algoritm\Date	Lungime	Cost	Nod.tot	Nod.max	Timp (ms)
UCS	3	7.75	39849	35863	2326
A*/ e ban	3	7.75	20093	18426	1150
A*/ e1	3	7.75	7621	7592	430
A*/ e2	3	7.75	8773	7936	509
A*/ neadm	3	8.25	8737	7590	492
A*opt/ e ban	3	7.75	9859	5946	66115
A*opt/ e1	3	7.75	4991	3970	20912
A*opt/ e2	3	7.75	5592	3703	23819
IDA*/ e ban	3	7.75	-	-	43677
IDA*/ e1	3	7.75	1765827	8830008	2384
IDA*/ e2	3	7.75	-	-	5649

Concluzii

1. Cel mai efficient algoritm bazandu-ne pe cele doua tabele este A*simplu, cele mai eficiente euristici fiind e1 si e1 in functie de input. Euristica e2 este da rezultate mai bune cand exista Acest algoritm este mai rapid decat IDA* si decat A*opt deoarece acesta are implementata PirorityQueue din python, in timp ce IDA*, cat si A*opt nu au.
2. IDA* este mai rapid decat A*opt, insa acesta necesita mai multa memorie.
3. UCS este destul de rapid datorita faptului ca a fost implementat cu PriorityQueue-ul din python.

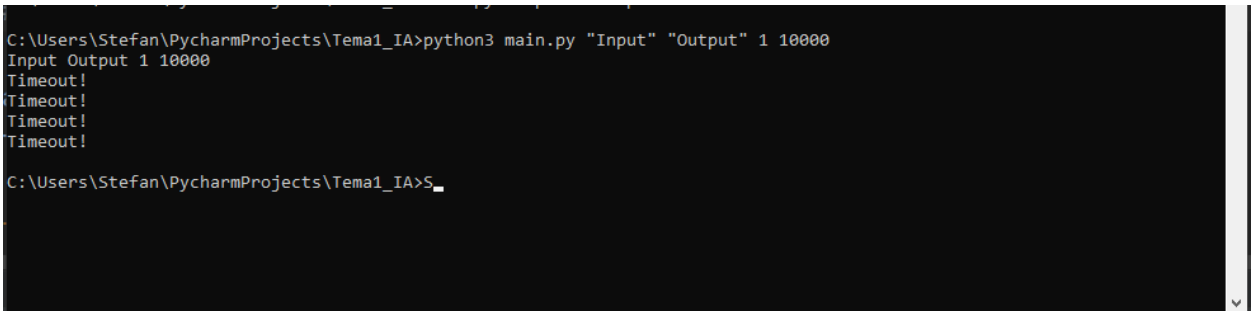
Exemple de apel

```
:\Users\Stefan\PycharmProjects\Tema1_IA>python3 main.py "Input" "Output" 1 10000
nput Output 1 10000
an't get this scope from this start!
raceback (most recent call last):
 File "C:\Users\Stefan\PycharmProjects\Tema1_IA\main.py", line 563, in <module>
   gr = Graph(folder_input + "\\" + file)
 File "C:\Users\Stefan\PycharmProjects\Tema1_IA\main.py", line 97, in __init__
   raise Exception("Invalid input!")
xception: Invalid input!

:\Users\Stefan\PycharmProjects\Tema1_IA>
```

In acest caz apare exceptia “Invalid input!”, deoarece in fisierul input1.txt inputul este gresit.

```
C:\Users\Stefan\PycharmProjects\Tema1_IA>python3 main.py "Input" "Output" 1 10000
Input Output 1 10000
Traceback (most recent call last):
  File "C:\Users\Stefan\PycharmProjects\Tema1_IA\main.py", line 560, in <module>
    gr = Graph(folder_input + "\\" + file)
  File "C:\Users\Stefan\PycharmProjects\Tema1_IA\main.py", line 103, in __init__
    raise Exception("States start and scope are equal!")
Exception: States start and scope are equal!

C:\Users\Stefan\PycharmProjects\Tema1_IA>
```

Starea scop este egala cu starea initiala, deci nu are rost ca programul sa ruleze in continuare.



Programul a functionat normal, dar unele functii au deposit limita de timeout, astfel acestea s-au oprit, lasand fisierul in care scriau incomplete.