

aprilie 2021

Problema decupării

- explicarea euristicilor -

Hazaparu Daria

Facultatea de Matematică și Informatică, Informatică

Inteligență artificială – Knowledge representation

Grupa 231

Enunț:

Se consideră un grid (matrice) de dimensiuni $N \times M$ (N linii și M coloane). În grid avem litere de la a la z . Dorim prin tăieri succesive de linii și coloane să ajungem la o anumită stare scop.

Mutările posibile sunt:

- tăierea a X ($X \geq 1$) coloane (cost: $1 + (k / (\text{nr coloane tăiate}))$), unde k reprezintă numărul, din întreaga zonă decupată, a perechilor de vecini care sunt diferiți între ei; evident fără a lua și simetricele perechilor; de exemplu, dacă am găsit perechea $\{(0,0), (0,1)\}$ nu vom lua și perechea $\{(0,1), (0,0)\}$.
- tăierea a X ($X \geq 1$) linii (cost: $(\text{nr coloane}) / (\text{nr linii tăiate})$)

Rezolvare

Am ales să reprezint fiecare stare ca un nod al unui graf de tip arbore cu rădăcina starea inițială.

Pentru rezolvarea acestei probleme, am folosit algoritmi UCS, A*, A* optimizat și IDA*.

Clasele rezolvării:

Clasa **NodParcure** reprezintă nodul din graf care are un id, o informație, care în cazul problemei este o matrice, un părinte (care pentru inițial este None), costul drumului până la nodul respectiv și euristica.

```
class NodParcure:  
    def __init__(self, id, info, parinte, cost=0, h=0):  
        self.id = id  
        self.info = info  
        self.parinte = parinte # parintele din arborele de parcurgere  
        self.g = cost # consider cost=1 pentru o mutare  
        self.h = h # euristica  
        self.f = self.g + self.h # suma
```

În această clasă sunt definite metodele `obțineDrum`, `afisDrum` și `contineInDrum`. Numele metodei indică și scopul acesteia.

Clasa **Graph** reprezintă modul în care este depozitat arborele (adică mai multe noduri conectate între ele pe care se aplică metode specifice). Atributele grafului sunt nodul de start/rădăcina, starea scop și câte un atribut pentru calcularea a 2 euristici.

```
class Graph:  
    def __init__(self, initial, finish):  
        self.start = NodParcure(1, initial, None)  
        self.scop = finish  
        self.eur1 = len(initial) - len(finish) if len(initial) - len(finish) > 0 else 1  
        self.eur2 = len(initial) - len(finish) + len(initial[0]) - len(finish[0]) \\\n            if len(initial) - len(finish) + len(\n                initial[0]) - len(finish[0]) > 0 else 1
```

În această clasă sunt definite metodele `testeazaScop`, `calculeaza_cost_linii`, `calculeaza_cost_coloane`, `calculeaza_euristica_1`, `calculeaza_euristica_2`, `calculeaza_euristica_3` și `genereazaSuccesori`.

genereazaSuccesori

Succesorii unei stări sunt de 2 feluri:

1) nodul curent din care tăiem prima linie, primele 2 linii, primele $n-1$ linii (unde n e nr de linii), a doua linie, a doua și a treia linie, ... ultima linie;

2) similar cu tăierea liniilor, dar pentru coloane.

Apoi se calculează costul tierii după formula din enunț și euristica.

- “eur” este 1 pentru euristica banală. Dacă avem altfel de euristică, se va apela metoda corespunzătoare.
- “ind” este numărul de ordine al nodului, declarat global, initializat în funcția de apel a algoritmului.

În lista de succesori se adaugă un `NodParcure` nou cu datele generate anterior. Apoi se returnează.

```
def genereazaSuccesori(self, nodCurent, tip_euristica="euristica banala"):
    global ind
    listaSuccesori = []
    matrice = nodCurent.info
    for i in range(len(matrice)):
        for j in range(i, len(matrice)):
            succ = matrice[i:j + 1]
            if succ == matrice:
                continue
            rest = matrice[:i] + matrice[j + 1:]
            cost = self.calculeaza_cost_linii(succ, matrice)
            eur = 1
            if tip_euristica == "euristica 1":
                eur = self.calculeaza_euristica_1(cost)
            if tip_euristica == "euristica 2":
                eur = self.calculeaza_euristica_2(cost)
            if tip_euristica == "euristica 3":
                eur = self.calculeaza_euristica_3(cost, succ)
            ind += 1
            if self.verifica_succesor(rest):
                listaSuccesori.append(NodParcursere(ind, rest, nodCurent, nodCurent.g + cost, eur))

    for i in range(len(matrice[0])):
        for j in range(i, len(matrice[0])):
            succ = []
            rest = []
            for linie in matrice:
                succ.append(linie[i:j + 1])
                rest.append(linie[:i] + linie[j + 1:])
            if succ == matrice:
                continue
            cost = self.calculeaza_cost_coloane(succ)
            eur = 1
            if tip_euristica == "euristica 1":
                eur = self.calculeaza_euristica_1(cost)
            if tip_euristica == "euristica 2":
                eur = self.calculeaza_euristica_2(cost)
            if tip_euristica == "euristica 3":
                eur = self.calculeaza_euristica_3(cost, succ)
            ind += 1
            if self.verifica_succesor(rest):
                listaSuccesori.append(NodParcursere(ind, rest, nodCurent, nodCurent.g + cost, eur))

    return listaSuccesori
```

```
def calculeaza_cost_linii(self, succ, matrice):  
    return len(matrice[0]) / len(succ)  
  
def calculeaza_cost_coloane(self, mat):  
    cost = 0  
    for i in range(len(mat)):  
        for j in range(len(mat[i])):  
            if i != len(mat) - 1:  
                if mat[i][j] != mat[i + 1][j]:  
                    cost += 1  
            if j != len(mat[len(mat) - 1]) - 1:  
                if mat[i][j] != mat[i][j + 1]:  
                    cost += 1  
    return 1 + cost / (len(mat[0]))
```

Pentru a reduce numărul de succesori generați, am ales să creez o funcție care verifică dacă prima linie din scop se mai găsește undeva în succesori, nu neapărat o literă lângă cealaltă. Astfel, am redus cu mult numărul de succesori generați.

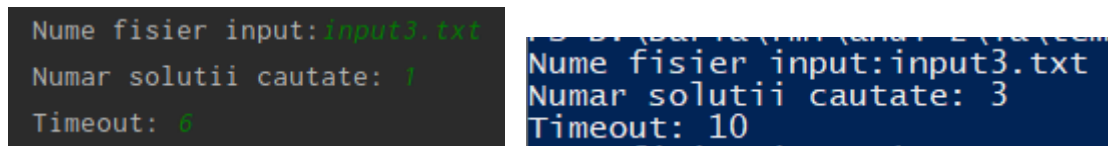
```
def verifica_succesor(self, matrice):  
    linie_scop = self.scop[0]  
    # print(linie_scop)  
    for i in range(len(matrice)):  
        for j in range(len(matrice[i])):  
            if matrice[i][j] == linie_scop[0] and len(linie_scop) - j >= 0:  
                index = 0  
                for k in range(j, len(matrice[i])):  
                    if index < len(linie_scop):  
                        if matrice[i][k] == linie_scop[index]:  
                            index += 1  
                    else:  
                        break  
                if index == len(linie_scop) - 1:  
                    return True  
    return False
```

Rezolvarea propriu-zisă reprezintă găsirea drumului efectiv prin arbore, unde la fiecare stare se generează fiii, de la rădăcină până la starea finală/ scop. Descrierea algoritmilor nu face parte din subiectul documentației.

Rulare

Proiectul este scris în PyCharm.

Pentru rulare din linia de comandă, se scrie doar “python main.py”, iar restul datelor (fișierul de input, NSOL și timeout-ul) se vor introduce în cele ce urmează:



```
Nume fisier input: input3.txt
Numar solutii cautate: 1
Timeout: 6

Nume fisier input: input3.txt
Numar solutii cautate: 3
Timeout: 10
```

Atenție: pentru fișierul input2.txt, NSOL trebuie să fie 1, altfel se termină programul.

Fișiere

- Input1.txt conține date de intrare pentru care nu există soluție.
- Input2.txt conține date de intrare pentru care starea inițială este egală cu starea finală.
- Input3.txt conține date de intrare pentru care există soluții și nu blochează niciun algoritm.
- Input4.txt conține date de intrare pentru care există soluții, dar durează foarte mult.

Există câte un fișier de output pentru fiecare algoritm și fiecare euristică (13 fișiere) cu nume sugestive. În aceste fișiere sunt scrise soluția/soluțiile și, la final, durata execuției, numărul de noduri existente la un moment dat și numărul total de noduri generate.

Euristici

Euristica banală

Euristica o să fie egală cu 1 și va avea un comportament asemănător cu UCS.

Euristica 1

```
def calculeaza_euristica_1(self, cost):  
    return cost * self.eur1
```

Am ales această euristică pentru că scurtează foarte mult din durata execuției algoritmului de UCS.

Față de UCS, se adaugă și parametrul “h” care ia valoare costului înmulțit cu “eur1”, adică câte linii trebuie tăiate din starea inițială ca să se ajungă la starea finală. La prima vedere, înmulțirea nu pare să fie alegerea corectă, ci mai degrabă împărțirea, însă voi explica mai jos de ce nu s-a potrivit.

Euristica 2

```
def calculeaza_euristica_2(self, cost):  
    return cost * self.eur2
```

“eur2” reprezintă numărul de linii adunat cu numărul de coloane care trebuie tăiate din starea inițială ca să se ajungă la starea finală.

Am ales înmulțirea în locul împărțirii pentru că la împărțirea cu un număr mai mare, algoritmul ar deveni mai ineficient decât înainte.

Exemplu:

Pentru A* cu euristica 1 cu împărțire

```
Timp de executie: 0.4691944122314453 secunde
```

Și cu euristica 2 cu împărțire

```
Timp de executie: 0.7419741153717041 secunde
```


În schimb, A* cu euristica 1 cu înmulțire

```
Timp de executie: 0.06304717063903809 secunde
```

Și cu euristica 2 cu înmulțire

```
Timp de executie: 0.02203226089477539 secunde
```

- analog si pentru A* optimizat.

Exemplu rulat pe fisierul de input cu datele:

```
ddaab  
aaaff  
aabcc  
bbbff  
dadad  
  
aa  
af  
bc
```

NSOL = 3

Euristica 3

```
def calculeaza_euristica_3(self, cost, rest):  
    return cost / len(rest)
```

Acesta este un exemplu de euristică neadmisibilă. Ea depinde de starea în care se află nodul și nu de o formulă dată valabilă pentru toate nodurile (cum ar fi înmulțirea cu același număr ca mai sus). De asemenea, la un NSOL > 1, se observa ca drumurile se schimba si capata o lungime si un cost mai mare.

În tabelele de mai jos, se notează:

L = lungimea drumului

C = costul drumului

T = timpul execuției algoritmului

NMAX = numărul maxim de noduri existente la un moment dat în memorie

NT = numărul total de noduri

EB = euristica banală

E1 = euristica 1

E2 = euristica 2

E3 = euristica 3

```

ddaab
aaaff
aabcc
bbbff
dadad
|
aa
af
bc

```

NSOL = 1

	L	C	NMAX	NT	T
UCS	4	7.5	2471	4114	0.418052
A* EB	4	7.5	2471	4114	0.407069
A* E1	4	7.5	930	2900	0.070995
A* E2	4	7.5	528	4332	0.053001
A* E3	4	7.5	2638	9284	0.383034
A* opt EB	4	7.5	750 / 1176	2862	0.246903
A* opt E1	4	7.5	250 / 557	1519	0.035103
A* opt E2	4	7.5	214 / 623	1942	0.032995
A* opt E3	4	7.5	584 / 1368	3943	0.184996
IDA* EB	4	7.5		62611	0.462211
IDA* E1	4	7.5		17907	0.155004
IDA* E2	4	7.5		68636	0.535090
IDA* E3	4	7.5		112180	0.879723

```

ddaaba
aaafff
aabccd
bbaafa
dadada
|
aa
af
bd

```

NSOL = 1

	L	C	NMAX	NT	T
UCS	4	8.5	6534	9493	2.535795
A* EB	4	8.5	6534	9493	3.335638
A* E1	4	8.5	5352	13255	2.660620
A* E2	5	11	3191	28797	1.216617
A* E3	4	8.5	9114	43174	9.107143
A* opt EB	4	8.5	2296	5330	0.473468
A* opt E1	4	8.5	1543	3961	0.341630
A* opt E2	5	11	1896	5984	0.349994
A* opt E3	4	8.5	4757	13279	1.216399
IDA* EB	4	8.5		235730	2.246963
IDA* E1	5	10.5		428978	3.358908
IDA* E2	5	11		576046	4.710487
IDA* E3	4	8.5		2595724	22.556680

Concluzii

În urma studierii tabelelor, se pot afirma următoarele:

- UCS și A* cu euristica banală sunt foarte asemănătoare, produc același număr de noduri și dau același rezultat într-un timp foarte apropiat.
- Pe exemple mai mari, se observă că euristica 2 este mult mai bună decât euristica 1 pentru A* și A* optimizat.
- Chiar dacă euristica 2 tinde să producă mai multe noduri și drumuri de cost mai mare și mai lungi, timpul este mai scurt.
- Se observă ușor că euristica 3 nu numai că produce mai multe noduri, dar și durează foarte mult, dar numărul de noduri generate este mult mai mic.
- În funcție de input, există cazuri în care A* optimizat cu euristica 2 durează foarte puțin peste euristica 1.
- Față de A* și A* optimizat, IDA* nu pare să sufere nicio modificare pozitivă pentru euristicile mai bune, ba chiar durează dublu față de euristica banală și numărul nodurilor este dublat.

Alte observații:

- Cerința problemei nu este de a găsi un drum minim, ci de a găsi un drum oricare ar fi el, de aceea nu consider că lungimea mai mare pentru euristica 2 este o problemă atât timp cât rulează mai repede.
- Cei mai ineficienți algoritmi sunt UCS, A* EB, A* E3 și toate variantele de IDA*.