Barbu Iulia Andreea

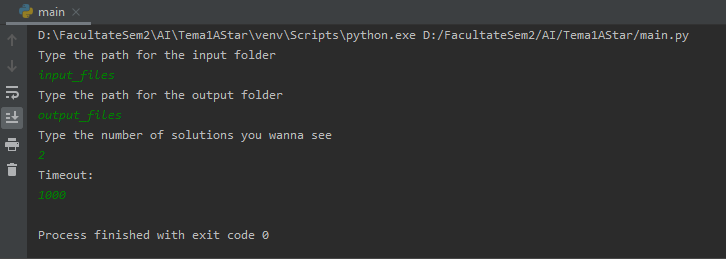
Grupa 231

Tema1

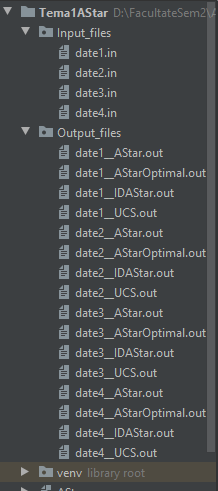
Documentatie

Se cer in linia de comanda calea catre folderul de input, cea catre folderul de output si numarul de solutii dorite.

Exemplu apelare program:



Programul va parcurge fiecare fisier din folderul dat pentru input si va creea in folderul de output cate 4 fisiere ( unul pentru fiecare algoritm: AStar, AStarOptimal, IDEAstar si UCS) cu rezultatele pentru fiecare input in parte. Exemplu cu 4 inputuri diferite:



Pentru AStar si IDAStar se afiseaza primele n solutii pentru fiecare euristica in parte(banala, admisibila1, admisibila2, neadmisibila).

Pentru AStar cu open si closed se afiseaza doar solutia cea mai buna pentru fiecare euristica in parte.

Pentru UCS se afiseaza primele n solutii.

**Despre euristici:**

* **Euristica admisibila1:** calculeaza costul minim catre o stare care se afla la fix un pas (un zid daramat) distanta. Este corecta deoarece, din moment ce stim ca ne aflam la cel putin o mutare distanta de starea finala ( mai este necesar cel putin un zid de distrus, ca altfel am fi fost deja in stare finala. ), atunci putem fi siguri ca si costul solutiei va fi cel putin cat am adunat pana acum + costul minim necesar urmatorului pas. h’ <= h
* **Euristica admisibila2:** gaseste costul minim catre urmatorul “o” aflat in matrice. Este corecta deoarece stim ca pozitia finala se afla neaparat intr o casuta “o”. Deci poate fi, fie plasata in “balta” de “o” uri cea mai apropiata, fie mai departe.

Fie k costul minim necesar pentru a ajunge la o balta. = h’

Caz 1: canalul se afla in balta in care ajungem cu costul k, deci h’ == h

Caz 2: canalul se afla in alta balta mai indepartata. Atunci stim sigur ca va trebui sa distrugem mai mult decat k ziduri sa ajungem la el, deci h’ < h

Din cazurile 1 si 2 => h’ <= h.

* **Euristica neadmisibila: = costul maxim posibil de unde a ajuns apa si pana la canal, pe distanta euclidiana.**

Adica, in momentul raspandirii apei, gasesc cel mai apropiat “o” de canal. Presupun ca de la acest “o” pana la canal mai am de distrus numai ziduri de cost maxim (adica de cost 5). Asadar calculez distanta euclidiana intre “o” si canal si o inmultesc cu 5. Gasirea distantei minime euclidiene se realizeaza oarecum in O(1), din moment ce oricum erau parcurse toate “o” urile baltii si marcate cu i la creearea starii ( vezi functia fill ) si atunci, in aceasta functia, doar am mai memorat pozitia “o”-ului cel mai apropiat de canal.

Un exemplu in care euristica neadmisibila **nu** obtine solutia optima:

date de intrare:

0 0  
5 3  
ooo##ooo#  
#o#o#o##o  
####ooo#o  
#o####o##  
#oo#ooo#o  
o##oo##oo

date de iesire:

neadmisibila  
  
Timp de executie = 1 ms  
nr ziduri distruse (lungimea drumului) = 3  
  
  
Sparg zidul de pe pozitia 0 0  
costul acestei mutari =0  
cost total = 0  
nr ordine =0  
['i', 'i', 'i', '#', '#', 'o', 'o', 'o', '#']  
['#', 'i', '#', 'o', '#', 'o', '#', '#', 'o']  
['#', '#', '#', '#', 'o', 'o', 'o', '#', 'o']  
['#', 'o', '#', '#', '#', '#', 'o', '#', '#']  
['#', 'o', 'o', '#', 'o', 'o', 'o', '#', 'o']  
['o', '#', '#', 'o', 'o', '#', '#', 'o', 'o']  
  
Sparg zidul de pe pozitia 2 1  
costul acestei mutari =7  
cost total = 7  
nr ordine =4  
['i', 'i', 'i', '#', '#', 'o', 'o', 'o', '#']  
['#', 'i', '#', 'o', '#', 'o', '#', '#', 'o']  
['#', 'i', '#', '#', 'o', 'o', 'o', '#', 'o']  
['#', 'i', '#', '#', '#', '#', 'o', '#', '#']  
['#', 'i', 'i', '#', 'o', 'o', 'o', '#', 'o']  
['o', '#', '#', 'o', 'o', '#', '#', 'o', 'o']  
  
Sparg zidul de pe pozitia 5 2  
costul acestei mutari =3  
cost total = 10  
nr ordine =14  
['i', 'i', 'i', '#', '#', 'i', 'i', 'i', '#']  
['#', 'i', '#', 'o', '#', 'i', '#', '#', 'o']  
['#', 'i', '#', '#', 'i', 'i', 'i', '#', 'o']  
['#', 'i', '#', '#', '#', '#', 'i', '#', '#']  
['#', 'i', 'i', '#', 'i', 'i', 'i', '#', 'o']  
['o', '#', 'i', 'i', 'i', '#', '#', 'o', 'o']

Outputul optim ar fi fost:

admisibila1  
  
Timp de executie = 3 ms  
nr ziduri distruse (lungimea drumului) = 3  
  
  
Sparg zidul de pe pozitia 0 0  
costul acestei mutari =0  
cost total = 0  
nr ordine =0  
['i', 'i', 'i', '#', '#', 'o', 'o', 'o', '#']  
['#', 'i', '#', 'o', '#', 'o', '#', '#', 'o']  
['#', '#', '#', '#', 'o', 'o', 'o', '#', 'o']  
['#', 'o', '#', '#', '#', '#', 'o', '#', '#']  
['#', 'o', 'o', '#', 'o', 'o', 'o', '#', 'o']  
['o', '#', '#', 'o', 'o', '#', '#', 'o', 'o']  
  
Sparg zidul de pe pozitia 0 3  
costul acestei mutari =4  
cost total = 4  
nr ordine =1  
['i', 'i', 'i', 'i', '#', 'o', 'o', 'o', '#']  
['#', 'i', '#', 'i', '#', 'o', '#', '#', 'o']  
['#', '#', '#', '#', 'o', 'o', 'o', '#', 'o']  
['#', 'o', '#', '#', '#', '#', 'o', '#', '#']  
['#', 'o', 'o', '#', 'o', 'o', 'o', '#', 'o']  
['o', '#', '#', 'o', 'o', '#', '#', 'o', 'o']  
  
Sparg zidul de pe pozitia 0 4  
costul acestei mutari =2  
cost total = 6  
nr ordine =8  
['i', 'i', 'i', 'i', 'i', 'i', 'i', 'i', '#']  
['#', 'i', '#', 'i', '#', 'i', '#', '#', 'o']  
['#', '#', '#', '#', 'i', 'i', 'i', '#', 'o']  
['#', 'o', '#', '#', '#', '#', 'i', '#', '#']  
['#', 'o', 'o', '#', 'i', 'i', 'i', '#', 'o']  
['o', '#', '#', 'i', 'i', '#', '#', 'o', 'o']

**Functii suplimentare:**

**- sunt definite in State (clasa Nod ):**

* **Functia fill:** este o functie recursivacare are rolul de a raspandi apa prin toate casutele vecine “o” dupa ce un zid a fost daramat, aplicand chiar algoritmul fill. Cu ocazia asta determina si casutele vecine ale starii (neighbours), adica toate posibilitatile de avansare cu un pas ( toate zidurile vecine care vor reprezenta coordonatele starilor copii). Si de asemenea, determina si cel mai aporpiat “o” (in care a ajuns apa pana acum) de canal (ma folosesc de distanta dintre cele doua in euristica neadmisibila).

def fill(self, x, y, neighbours):  
  
 *"""  
 Functie care are rolul de a raspandi apa prin toate pozitiile vecine de tip 'o'* ***:param*** *x,* ***:param*** *y: coordonatele pozitiei unde tocmai a ajuns apa* ***:param*** *neighbours: coordonatele vecine valide ale intregii balti, dupa ce e umpluta  
 """* self.matrix[x][y] = 'i'  
  
 if x == self.final\_x and y == self.final\_y:  
 self.isFinalState = True  
  
 d = self.getDistance(x, y)  
  
 if d < self.best\_o\_dist:  
 self.best\_o\_dist = d  
  
 for i in range(0, 4):  
  
 neighbour\_x = x + vx[i]  
 neighbour\_y = y + vy[i]  
  
 if isInRange(self.matrix, neighbour\_x, neighbour\_y):  
 if self.matrix[neighbour\_x][neighbour\_y] == 'o':  
 self.fill(neighbour\_x, neighbour\_y, neighbours)  
  
 elif self.matrix[neighbour\_x][neighbour\_y] == '#':  
 if (neighbour\_x, neighbour\_y) not in neighbours:  
 neighbours.append((neighbour\_x, neighbour\_y))  
  
 self.neighbours\_coord = neighbours

**(Functii pe care le-am folosit pentru euristici):**

* **Pentru admisibila1:** parcurg vecinii nodului si il gasesc pe cel de cost minim

def one\_step\_forward\_cost(self):  
 best = None  
  
 for neighbour\_x, neighbour\_y in self.neighbours\_coord:  
  
 cost = calculate\_cost(self.matrix, neighbour\_x, neighbour\_y)  
  
 if best is None:  
 best = cost  
  
 if cost < best:  
 best = cost  
  
 return best

* **Pentru admisibila2:** caut in toata matricea urmatorul “o” catre care costul e minim. Pun intr un priority queue costul pana intr un punct si coordonatele punctului si extrag mereu costul minim, pana intalnesc un “o”.

def next\_o\_cost(self):  
 *"""  
 O funcie foarte asemanatoare cu AStar care ca imi gaseste costul minim pana la o casuta "o",  
 """* neighbours = deepcopy(self.neighbours\_coord)  
 matrix = deepcopy(self.matrix)  
  
 queue = PriorityQueue()  
  
 for x, y in neighbours:  
 queue.put((0, x, y))  
  
 while True and not queue.empty():  
 cost, x, y = queue.get()  
  
 if matrix[x][y] == 'o':  
 return cost  
 else:  
 # nu are cum sa fie decat "#", nu ajunge niciodata cu "i" aici  
 for i in range(4):  
 neighbour\_x = x + vx[i]  
 neighbour\_y = y + vy[i]  
  
 if isInRange(matrix, neighbour\_x, neighbour\_y):  
 if matrix[neighbour\_x][neighbour\_y] != 'i':  
 current\_cost = calculate\_cost(matrix, neighbour\_x, neighbour\_y)  
 new\_cost = cost + current\_cost  
 queue.put((new\_cost, neighbour\_x, neighbour\_y))  
  
 matrix[x][y] = 'i'  
  
 return 0

**Optimizari:**

O optimizare ar fi faptul ca nu construiesc o lista de stari finale si verificarea pentu o stare finala este realizata in O(1) (Daca am ajuns la coordonatele canalului, pot fi sigura ca drumul realizat pana aici a fost corect, deoarece ma deplasez din aproape in aproape, deci nu e nevoie sa verific cum arata intreaga matrice).

Am folosit un PriorityQueue in loc de coada normala, oriunde am avut nevoie.

**Validari:**

Verific daca coordonatele punctelor se incadreaza in matrice.

if not isInRange(matrix, robinet\_x, robinet\_y) or not isInRange(matrix, canal\_x, canal\_y):  
 g.write('Datele de input sunt gresite. Problema nu are solutii')

**Observatii:**

AStarOptimizat pare sa dea rezultate mai lente decat AStar pe aceasta problema, deoarece el face operatii in plus pentru anticiparea duplicatelor. In mod normal, cand AStar ar ajunge la duplicate, l ar depasi ca timp (s-ar purta mai lent), dar in aceasta problema se intampla foarte rar sa se obtina doua stari la fel, deoarece e vorba de deplasari din aproape in aproape in matrice de la un punct de plecare la un punct destinatie, si atunci doar foarte rar s-ar intampla sa aiba motive sa faca intoarceri (aproape niciodata). De aceea aici coada closed nu ar fi necesara.