Universitatea din Craiova, Facultatea de Automatică,
Calculatoare și Electronică

Titlu tema: M Vehicles

Nume si prenume:Ciontu Claudia-Elena

Grupa: CR2.1 B

Anul de studiu: Anul 2

Specialitatea: Calculatoare si Tehnologia Informatiei

Cuprins

1 Enuntul problemei

Let us assume that m vehicles are located in squares (1, 1) through (m, 1) (the bottom row) of an m × m squared parking. The vehicles must be moved to the top row, but arranged in reverse order; so vehicle i starting from (i, 1) must end up in (m i + 1, m). On each time step, each of the m vehicles is restricted to move only one square up, down, left, or right, or keep current position (i.e. does not move); but if a vehicle does not move, one other adjacent vehicle (but not more than one) can hop over it. Two vehicles cannot occupy the same square.

a. Write a detailed formulation for this search problem.

b. Identify a suitable search algorithm for this task and explain your choice.

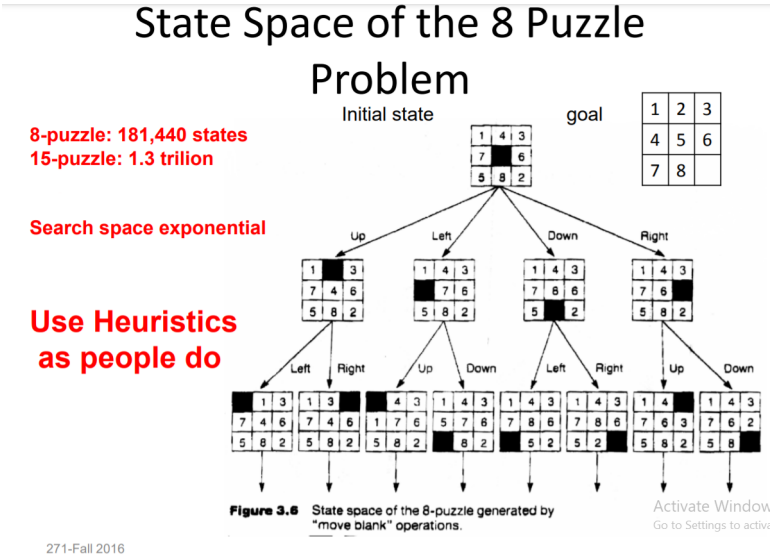
2 Pseudocode of Algoritms

Pentru problema primita am ales sa scriu codul in Python si sa ma folosesc de analogiile de la 8-puzzle si 15-puzzle

Am considerat ca cele doua metode de cautatre alese de mine ma vor ajuta in realizarea experiemntelor programului meu si ca vor fi potrivite pentru el. 8-puzzle-ul aparține familiei de puzzle-uri glisante, care sunt adesea folosite ca probleme de testare SLIDING-BLOCK PUZZLES pentru noi algoritmi de căutare în AI. Această familie este cunoscută ca fiind NP-completă

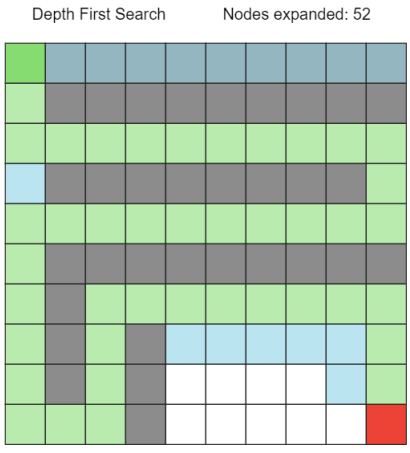
Deci nu ne așteptăm să găsim metode semnificativ mai bune în cel mai rău caz decât algoritmii de căutare descriși în acest capitol și în următorul. 8-puzzle are 9! / 2 = 181, 440 la care se poate ajunge afirmă și se rezolvă ușor.

Puzzle-ul 15 (pe o placă 4 × 4) are în jur de 1,3 trilioane de stări, iar instanțele aleatorii pot fi rezolvate optim în câteva milisecunde de către cei mai buni algoritmi de căutare. 24-puzzle (pe o tablă 5 × 5) are în jur de 1025 de stări, iar instanțele aleatorii durează câteva ore pentru a se rezolva optim



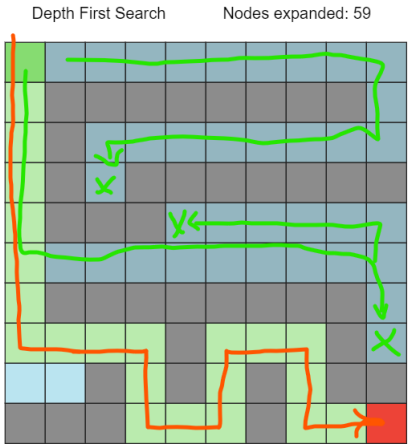
2.1 DEPTH-FIRST-GRAPH-SEARCH

DEPTH-FIRST-GRAPH-SEARCH este un algoritm pentru căutarea unei structuri grafice. Unul începe de la un vârf și explorează cât mai mult posibil într-o singură direcție înainte de a merge înapoi și de a alege o altă direcție. Putem vizualiza DFS, în cazul în care pătratele în gri reprezintă vertici neconectate care nu pot fi atinse în grafic. Urmăriți imaginea DFS de mai jos pentru a afla cum funcționează.



Într-un alt exemplu de DFS, vedem că încearcă să meargă într-

o direcție cât mai mult posibil și schimbând direcțiile numai când am ajuns la capătul (un perete) al acelei direcții. Dacă nu există nicio altă direcție de intrare și am ajuns la o fundătură completă, revenim la ultima intersecție despărțitoare și încercăm să mergem cât mai adânc posibil în altă direcție. Repetăm acest proces până găsim sfârșitul sau explorăm fiecare cale posibilă.



function **DEPTH-FIRST-GRAPH-SEARCH** (*problem*) **returns** a solution node or fail

1. $frontier \leftarrow$ a node with STATE = *problem*.INITIAL , PATH-COST = 0
2. *explored* $\leftarrow$ an empty set
3. **loop dot** IS-EMPTY? ($frontier$) **then return failure**
4. *node* $\leftarrow$POP($frontier$)
5. **if** *problem*.IS-GOAL-Test(*node*.STATE) **then**
6. **return** *Solution*(*node*)
7. **end if**
8. add node.State to explored
9. **for each** *action in problem*.ACTION(*node* , *State*) **do**
10. *child* $\leftarrow$ $CHILD\text{-}NODE(problem, node, action)$
11. **if** *child*.STATE **is not** in not *explored* of $frontier$
then SOLUTION(child)
12. **return** *none*

2.2 ASTAR-SEARCH

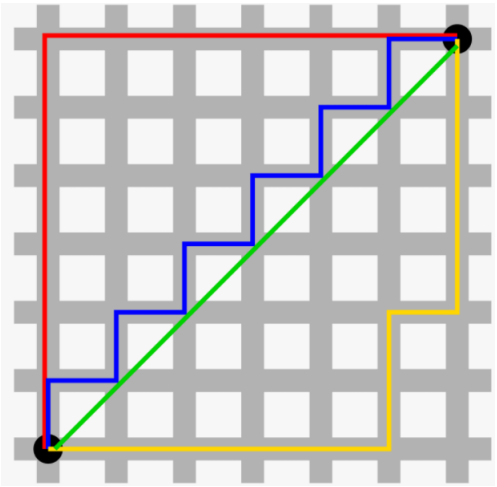
A * este un algoritm computerizat care este utilizat pe scară largă în căutare de trasee și traversarea graficelor. Algoritmul trasează în mod eficient o cale parcursibilă între mai multe noduri sau puncte, pe grafic.

Pe o hartă cu multe obstacole, identificarea drumurilor de la punctele A la B poate fi dificilă. Un robot, de exemplu, fără a obține multă altă direcție, va continua până când întâlnește un obstacol, ca în exemplul de găsire a căilor din stânga de mai jos.

Cu toate acestea, algoritmul A * introduce o euristică într-un algoritm obișnuit de căutare a graficelor, planificând în esență la fiecare pas, astfel încât să se ia o decizie mai optimă. Cu A *, un robot ar găsi în schimb o cale într-un mod similar cu diagrama din dreapta de mai jos.

A * este o extensie a algoritmului lui Dijkstra cu unele caracteristici ale căutării prin lățime (BFS).

La fel ca Dijkstra, A * funcționează realizând un arbore de cale cu cel mai mic cost de la nodul de pornire la nodul țintă. Ceea ce face A * diferit și mai bun pentru multe căutări este că pentru fiecare nod, A * folosește o funcție f (n) f (n) care oferă o estimare a costului total al unei căi care utilizează acel nod. Prin urmare, A * este o funcție euristică, care diferă de un algoritm prin aceea că o euristică este mai mult o estimare și nu este neapărat corectă.
A * extinde căile care sunt deja mai puțin costisitoare utilizând această funcție:
f (n) = g (n) + h (n),Unde
f (n) = costul total estimat al traseului prin nod n
g (n) = cost până acum pentru a ajunge la nod n
h (n) = costul estimat de la nn la obiectiv. Aceasta este partea euristică a funcției de cost, deci este ca o presupunere.



În grila de mai sus, algoritmul A * începe la început (nod roșu) și ia în considerare toate celulele adiacente. Odată ce lista celulelor adiacente a fost completată, aceasta le filtrează pe cele care sunt inaccesibile (pereți, obstacole, în afara limitelor). Apoi alege celula cu cel mai mic cost, care este f (n) estimat. Acest proces se repetă recursiv până când a fost găsită cea mai scurtă cale către țintă (nod albastru). Calculul lui f (n) f (n) se face printr-o euristică care oferă de obicei rezultate bune.

function **ASTAR-SEARCH** $(problem, h = NONE)$ **returns** a solution node or failure

1. $h \leftarrow$ memoize(h or problem.h, 'h')
2. **return** $best\_first\_graph\_search(problem, lambdan : n.path\_cost + h(n))$

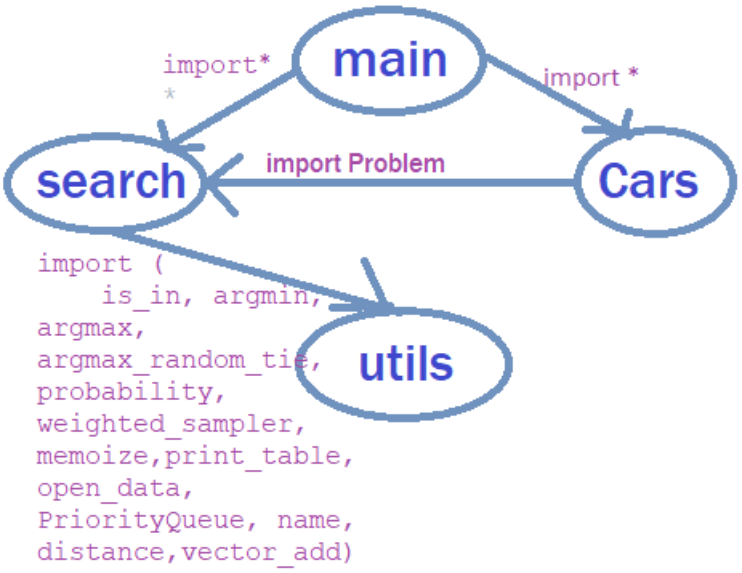
function **BEST-FIRST-SEARCH** $(problem, f)$ **returns** a solution node or none

1. $node \leftarrow$ a node with STATE = $problem$.INITIAL
2. $frontier \leftarrow$ a priority queue ordered by f
3. $explored \leftarrow$ a lookup table, with one entry with key $problem$.INITIAL and value $node$
4. **while** not IS-EMPTY $(frontier)$ **do**
5. $node \leftarrow$POP$(frontier)$
6. **if** $problem$.IS-GOAL-Test($node$.STATE) **then**
7. **return** $node$
8. **end** if

9. **for each** *child* **in** EXPAND(*problem* , *node*)
10. **if** *child*.STATE **is not** in *explored* **and** *child* **is not** in *frontier*
or *child*.PATH-COST $<$ *explored* $[child.STATE].PATH - COST$ **then**
11. *explored* $[child.STATE] \leftarrow child$
12. **add** *child* **to** *frontier*
13. **end** if
14. **end** for
15. **return** *none*

3 Schita Aplicației.

3.1 Ansamblul arhitectural al aplicatiei



Programul este organizat pe module,fiecare modul continand functii particulare. Functiile sunt apelate in main. Structura programului arata ca in figura de mai sus,iar modulele sunt urmatoarele:

-main(modulul principal ce ruleaza programul)

-Cars(modulul unde sunt implememtate functiile ce ajuta la rezolvare problemei)

-search(modulul unde sunt implementate diverse functii de cautare)

-utils(modulul ce contine tool-uri ce ajuta la rezolvarea problemei

3.2 Specificatiile formatului datelor de intrare

Generarea datelor de intrare se realizeaza cu ajutorul variabilei *marime_parcare*
Variabila *marime_parcare* este generata aleatoriu cu ajutorul functiei din Python numita *random.randint(a,b)* . Valorile acestei variabile se afla intre 1 si 10.

In functie de valoarea generata pentru variabila *marime_parcare* initiam *INITIAL_STATE* .
Acest lucru se realizeaza cu ajutorul metodei *append()* care adaugă un element la sfârșitul listei.

```
for i in range(marime_parcare):
    date_intrare.append(i + 1)
for i in range(marime_parcare * (marime_parcare - 1)):
    date_intrare.append(0)
```

3.3 Specificatiile formatului datelor de iesire

Pentru generarea datelor de iesire initializam *GOAL_STATE* cu ajutorul metodei *append()* care adaugă un element la sfârșitul listei.
De mentionat ca variabila *marime_parcare_aux* este o copie a variabilei *marime_parcare*

```
for i in range(marime_parcare * (marime_parcare - 1)):
    date_iesire.append(0)
for i in range(marime_parcare):
    date_iesire.append(marime_parcare_aux)
    marime_parcare_aux -= 1
```

Datele de iesire se genereaza in fisiere de tip *date_X.txt* cat si in consola.
In fisierul de mai jos putem observa ca datele noastre de iesire consta in:
-tipul de cautare folosit;
-dimensiunea folosita a parcarii noatre de masini;
-starea initiala ;
-obiectivul nostru unde vrem sa ajungem;
-actiunile executate si cefunctie de a calcula distanta;
-timpul de executie in secunde;
-costul cai;

```
Folosind A* search
Dimensiunea parcarii este : 2x2
Starea initiala este: (1, 2, 0, 0)
Obiectivul este: (0, 0, 2, 1)
Actiunile executate cu ajutorul Distantei Manhattan sunt: ['DOWN', 'DOWN', 'UP'
Timpul executat este de 0.0010008811950683594s
Costul caii (Manhatten distance ) este: 6
Folosind Depth_first_graph_search
Dimensiunea parcarii este : 2x2
Starea initiala este: (1, 2, 0, 0)
Obiectivul este: (0, 0, 2, 1)
Actiunile executate cu ajutorul Distantei Manhattan  sunt: ['DOWN', 'RIGHT', 'L
Timpul executat este de 0.0s
Costul caii este: 4
```

In consola,dupa cum putem vedea in imaginea de mai jos, avem generat toate mutarile pe care le fac masinile din *INITIAL_STATE* pana in *GOAL_STATE*

De asemenea avem generat cu ajutorul functiei *compare_searcher* din modulul *search* o comparatie intre cele 2 metode de cautare,respectiv

metode de a calcula distanta (de metionat ca aceasta functie am preluat-o in lucrarea de laborator)

```
"C:\Users\ciont\Desktop\Facultate\Inteligenta Artificiala\venv\Scripts\python.exe" "C:/Users/cio
[<Pas: (1, 2, 0, 0)>
, <Pas: (0, 2, 1, 0)>
, <Pas: (0, 0, 1, 2)>
, <Pas: (1, 0, 0, 2)>
, <Pas: (0, 1, 0, 2)>
, <Pas: (0, 1, 2, 0)>
, <Pas: (0, 0, 2, 1)>
]

[<Pas: (1, 2, 0, 0)>
, <Pas: (1, 0, 0, 2)>
, <Pas: (0, 1, 0, 2)>
, <Pas: (0, 1, 2, 0)>
, <Pas: (0, 0, 2, 1)>
]

Searcher                A* h1(n)                      A* h2(n)
depth_first_graph_search  <   5/   6/  11/(0, 0, 2, 1)>  <   5/   6/  11/(0, 0, 2, 1)>
astar_search              <   7/   9/  17/(0, 0, 2, 1)>  <   7/   9/  17/(0, 0, 2, 1)>
```

3.4 Lista Modulelor Aplicatiei

Penttru realizarea temei de laborator ,m-am folosit de *code_skeleton*-ul de la laboratorul 5 al disciplinei *Inteligenta Artificiala*. De mentionat ca am preluat modulul *search.py* si *utils.py* din laboartor si am updatat modulul *main.py* si *Cars.py* din laborator pentru Problema *fifteen_puzzle.*
Mai jos in aceasta sectiune voi face o prezentare a tuturor modulelor aplicatieii cat si o descriere a acestora.

3.4.1 main.py

Modulul **main** este modulul unde apelam toate functiile si declaram variabilele necesare. Incepem prin a crea fisierul unde se vor afisa rezultatele , declaram variabila marime_parcare care reprezinta dimensiunea parcarii noastre so se calculeaza folosint o functie de calculare random a unui numar .
Acestei variabole ii creem o copie ce ne va ajuta la initializarea *GOAL_STATE*-ului nostru.
Declaram listele noastre de *INITIAL_STATE* si *GOAL_STATE* cu ajutorul a doua for-uri:

Primele 2 for -uri creeaza *INITIAL_STATE* (de exemplu pentru o parcare de marime egala cu 3 avem *INITIAL_STATE* = 1,2,3,0,0,0,0,0,0

```
    for i in range  (marime_parcare):
date_intrare.append(i+1)
    for i in range  (marime_parcare *(marime_parcare − 1) ):
date_intrare.append(0)
```

Urmatoarele 2 for -uri creeaza *GOAL_STATE* (de exemplu pentru o parcare de marime egala cu 3 avem *GOAL_STATE* = 0,0,0,0,0,0,3,2,1

```
    for i in range  (marime_parcare):
date_intrare.append(i+1)
    for i in range  (marime_parcare *(marime_parcare − 1) ):
date_intrare.append(0)
```

Am folosim tuple(x) pentru a transforma sirul nostru intr-un tuplu ca sa se poata da ca parametru

```
carsMiss=CarsMiss(tuple(date_intrare), marime_parcare, tuple(date_iesire))
```

```
carsMht=CarsMht(tuple(date_intrare), marime_parcare, tuple(date_iesire))
```

Mai jos se poate vedea un exemplu de cod unde calculez timpul de execcutie pe care il are programul
t1=reprezinta timpul de inceput ,iar t2=reprezinta timpul de sfarsit al programului
t=este diferenta dintre timpul final si cel initial al programlui si aceasta variabila se va afisa intr-un fisier

```
t1 = time.time()
path = depth_first_graph_search(carsMiss)
t2 = time.time()
t = t2 - t1
```

Costul caii il afisez de asemenea tot in fisier apeland str(path.path_cost)
De asemena la finalul programului apelez functia compare_searchers pentru a face o comparatie intre 2 metode de cuatare diferite sau calcul de distanta a aceluasi searcher

3.4.2 Cars.py

In modului *Cars.py* am declarate functiile
-definire a starii noatre initiale si finale
-returnare aa indexului unui pătratului gol într-o stare dată
-definirea actiunilor cum ar fi Up,Down,Left,etc.
-revenim la o nouă stare avand în vedere starea și acțiunea
-definim o unctie de returnare a unei valori euristice pentru o stare data

3.4.3 search.py

Acest modul a fost preluat din code_skeleton-ul laboratorului 5.

3.4.4 utils.py

Acest modul a fost preluat din code_skeleton-ul laboratorului 5.

3.5 Lista Functiilor Aplicatiei

3.5.1 main.py ← Functii

Modulul **main** este modulul unde apelam toate functiile si declaram variabilele necesare. Incepem prin a crea fisierul unde se vor afisa rezultatele

3.5.2 Cars.py ← Functii

class Cars(Problem):

def __init__ (self, initial: object, marime_parcare: object, goal = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,0)):
Definim starea obiectivului și inițializam o problemă
self.goal = goal
self.numar_curent = 0
self.marime_parcare = marime_parcare
Problem.__init__(self, initial, goal)

---

def find__blankt__square (self, state):

```
        Returnează indexul pătratului gol într-o stare dată
        numar_curent calculeaza pentru fiecare state in care masina sa fie mutata,
astfel muta doar o masina la un pas
    if self.numar_curent ≤ self.marime_parcare :
      self.numar_curent += 1
    if  self.numar_curent == self.marime_parcare + 1:
      self.numar_curent += 1
    return state.index ( self.numar_curent )
```

```
    def action  (self, state):
        Definim actiunile in functie de miscarile masinilor
        posible_action = [ 'UP', 'DOWN', 'LEFT', 'RIGHT', 'STAY',
'JUMPLEFT', 'JUMPRIGHT', 'JUMPUP', 'JUMPDOWN']
        daca ajungem pe goal state sa eliminam miscarile
    if  state [ self.index ] == self.goal [self.index ]:
      posible_action.remove = ( 'LEFT')
      posible_action.remove = ( 'UP')
      posible_action.remove = ( 'RIGHT')
      posible_action.remove = ( 'DOWN')
      posible_action.remove = ( 'JUMPLEFT')
      posible_action.remove = ( 'JUMPUP')
      posible_action.remove = ( 'JUMPRIGHT')
      posible_action.remove = ( 'JUMPDOWN')
      eliminam anumite actiuni in functie de plasarea masinilor
      spre exemplu sa nu faca miscari in afara matricei,
sa nu faca salturi aiurea in afara matricei sau peste 2 masini
    if  self.index % self.marime_parcare== 0 or  state [ self.index -1 ] ! = 0:
      posible_action.remove = ( 'LEFT')
    if  self.index % self.marime_parcare<= 1
 or  state [ self.index -2 ] ! = 0 or  state [ self.index -1 ] = = 0:
      posible_action.remove = ( 'JUMPLEFT')
    if  self.index ≤  self.marime_parcare
or  state [ self.index −  self.marime_parcare ] ! = 0:
      posible_action.remove = ( 'UP')
    if  self.index ≤ 2 * self.marime_parcare
or  state [ self.index −  2 * self.marime_parcare ] ! = 0:
or  state [ self.index −  self.marime_parcare ] = = 0:
      posible_action.remove = (
```

```
'JUMPUP')
    if self.index % self.marime_parcare== self.marime_parcare- 1
 or  state [ self.index +1 ] ! = 0:
      posible_action.remove = ( 'RIGHT')
    if  self.index % self.marime_parcare≥ =self.marime_parcare- 2
or  state [ self.index + 2 ] ! = 0:
or  state [ self.index +1 ] = = 0:
      posible_action.remove = ( 'JUMPRIGHT')
    if  self.index ≥( pow(self.marime_parcare,2)-self.marime_parcare-1)
or  state [ self.index +  self.marime_parcare ] ! = 0:
      posible_action.remove = ( 'DOWN')
    if  self.index ≥( pow(self.marime_parcare,2)-2*self.marime_parcare-1)
or  state [ self.index +  self.marime_parcare ] = = 0:
or  state [ self.index +  2*self.marime_parcare ] ! = 0:
      posible_action.remove = ( 'JUMPDOWN')
```

```
    def result  (self, state,action):
        Avand în vedere starea si actiunea,revenim la o noua stare care este rezultatul actiunii.
        Actiunea se presupune ca este o actiune valida in stat.
        blank is the index of the blank square.
        new_state = list ( state)
        exprimam delta ca actiunea pe care o poate executa o masina,
clasicele sus, jos, stanga, dreapta, si salturile peste o casuta, plus stay.
        delta = { 'UP':− self.marime_parcare ,'DOWN' :self.marime_parcare ,
'Left' :−1,'RIGHT' :1,'STAY' :0,'JUMPLEFT' : −2,'JUMPRIGHT' :2
'JUMPDOWN' :+2* self.marime_parcare,'JUMPUP' :−2*self.marime_parcare}
        neighbor = self.index +delta[action]
        new_state[self.index], new_state[neighbor] = new_state[neighbor], new_state[self.index]
        return tuple ( new_state)
```

```
    def h  (self, node):
        return sum ( s ! = g for (s,g) in zip(node.state,self.goal))
```

```
    classCarsMht(Cars):

        Manhattan Distance
```

```
    def h  (self, node):

        dim=self.marime_parcare
        return sum ( ( abs(int(s/dim) - int(g/dim))+ abs(int(s%dim) - int(g%dim))
for (s,g) in zip(node.state,self.goal))
```

```
        Return the heuristic value for a given state.
Default heuristic function used is
 h(n) = number of misplaced tiles
    classCarsMiss(Cars):

    def h  (self, node):
        return sum ( s ! = g for (s,g) in zip(node.state,self.goal))
```

3.5.3 search.py ← Functii

```
    def depth_first_graph_search  (problem):
 Căutați mai întâi cele mai adânci noduri din arborele de căutare..
Cauta printre succesorii unei probleme pentru a găsi un scop.
 Frontiera argumentului ar trebui să fie o coadă goală.Nu este prins in bucle.
 Dacă două căi ajung la aceeasi stare, o va utiliza doar pe prima..
frontier = [(Node(problem.initial))]
 explored = set()
while frontier:
 node = frontier.pop()
if problem.goal_test(node.state):
 return node
explored.add(node.state)
frontier.extend(child for child in node.expand(problem)
if child.state not in explored and
child not in frontier)
 return None
```

```
  def astar_search
Căutarea A * este cea mai bună căutare grafică cu f (n) = g (n) + h (n).
Trebuie să specificați funcția h când apelați astar_searchsau
altceva în subclasa voastra
(problem, h=None):
h = memoize(h or problem.h, 'h')
return best_first_graph_search(problem,lambda n:n.path_cost+h(n)
```

3.5.4 utils.py ← Functii

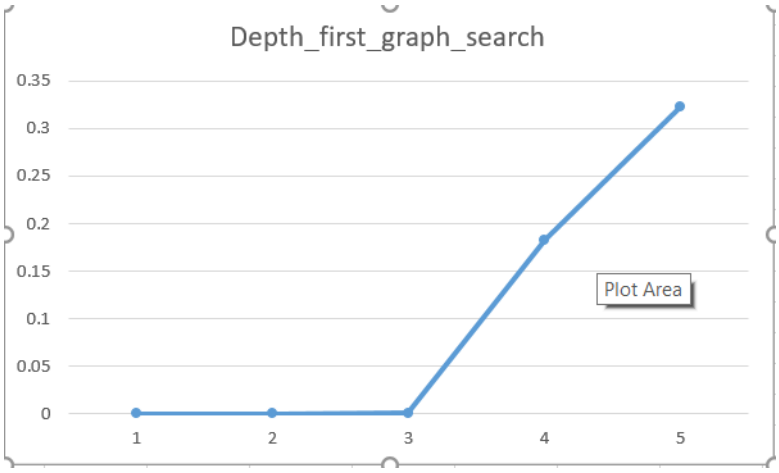
Functiile din modulul util.py sunt niste tool-uri ce sunt apelate de functiile din modulul search.py

4 Experimente si resultate

4.1 Timp de executie→Depth-first-graph-search

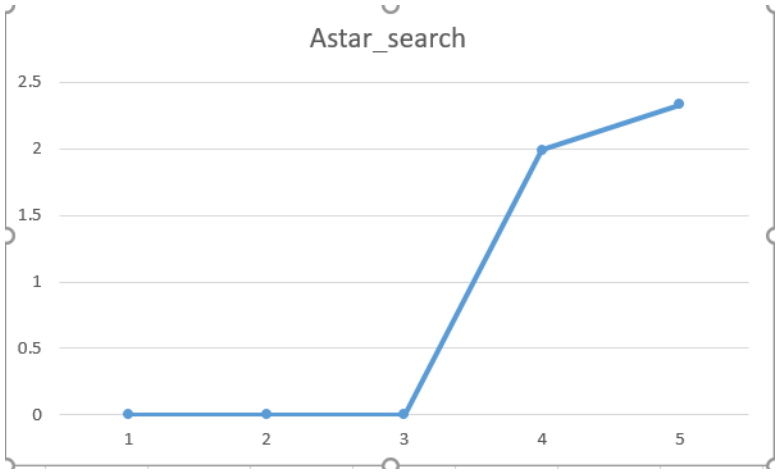
In sectiunea urmatoare am ales pentru experimente doua tipuri diferite de search si am comparat costul caii,respectiv timpul de executie pentru a face o comparatie.

In primul grafic am calculat timpul de executie al metodei de cautare *DEPTH-FIRST-GRAPH-SEARCH* pentru o parcare de marimea 1 , 2, 3, 4 si 5 si am trecut datele experimetale calculate cu distanta Manhattan. Se paote vedea o crestere destul de mare de la marime_parcare >3



4.2 Timp de executie→AStar-search

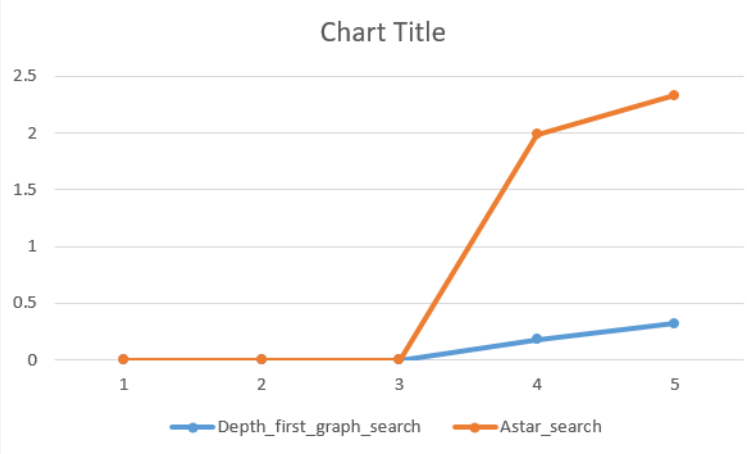
In cel de-al doilea grafic am calculat timpul de executie al metodei de cautare *ASTAR-SEARCH* pentru o parcare de marimea 1 , 2, 3, 4 si 5 si am trecut datele experimetale calculate cu distanta Manhattan. Se paote vedea o crestere destul de mare de la marime_parcare >3 insa o incetinire a cresterii pentru marime_parcare egal cu 4



4.3 Timp de executie→Comparatie

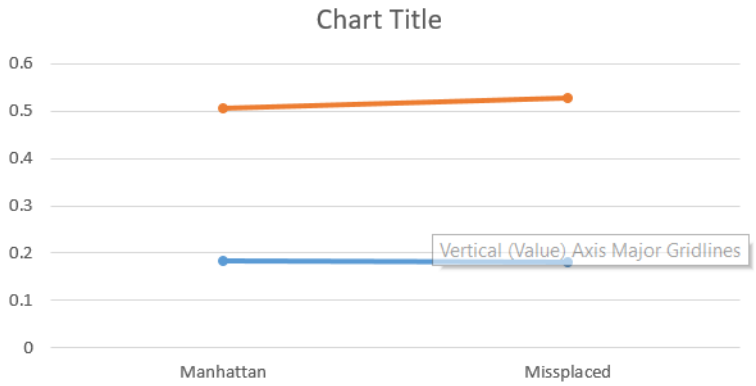
In cel de-al treilea grafic am facut o comparatie a metodei de cautare *DEPTH-FIRST-GRAPH-SEARCH* si a metodei de cautare *ASTAR-SEARCH* pentru o parcare de marimea 1 , 2, 3, 4 si 5 Se poate ob-

serva din graficul timpului de executie ca metoda *DEPTH-FIRST-GRAPH-SEARCH* este mai eficienta din punctul de vedere al timpului fata de metoda de cautare *ASTAR-SEARCH* . De asemenea o parcare mai mare sau egala cu 6 timpul de executie deja devine de nemasurat sau un timp ce tinde spre infinit din perspectiva metodei de cautare *ASTAR-SEARCH*



4.4 Distanta Manhatan si Missaplaced Tiles→Comparatie

In cel de-al patrulea grafic prezentat se poate observa o diferenta in timpul de executie metoda Distantei Manhattan si Missplaced Tiles



In cazul prezentat (marime_parcare este egal cu 3 respectiv 4) se vede o usoara diferenta intre cele doua metode siastfel reiese Manhattan este mai bun in comparatie cu Missplaced Tiles.

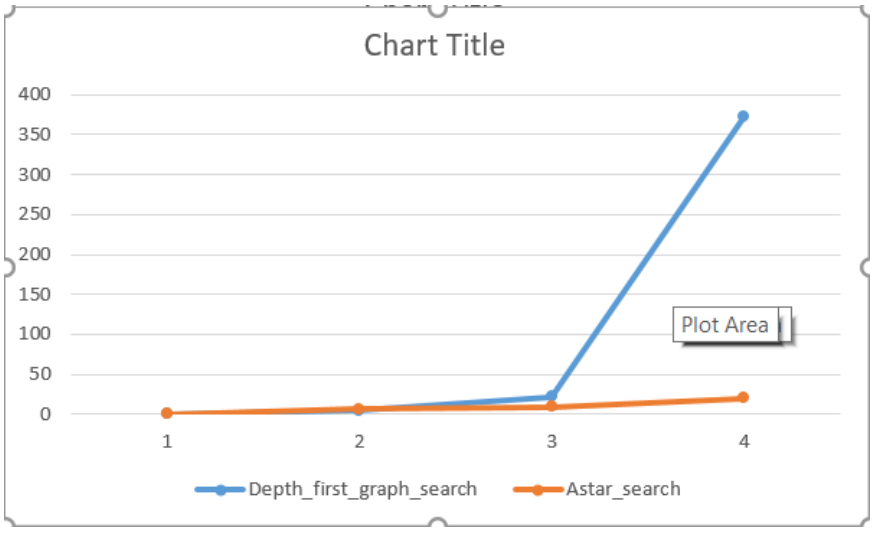
4.5 Costl caii→Comparatie

In ultimul grafic prezentat in acesta sectiune , se poate observa diferenta dintre costul caii metodei de cautare *DEPTH-FIRST-GRAPH-SEARCH* si costui caii metodei de cautare *ASTAR-SEARCH* .

Dupa cum se observa costul caii se prezinta in antiteza graficului de calculare a timpului de executie pentru cele 2 metode de cautare .

In timp ce timpul de executie pentru *DEPTH-FIRST-GRAPH-SEARCH* este mai eficient fata de *ASTAR-SEARCH*, calea mai eficienta este calcuata de ASTAR-SEARCH si cea mai putin eficienta este calculata de *DEPTH-FIRST-GRAPH-SEARCH*.

Se poate observa ca pentru o parcare mai mare sau egala cu 6 timpul costul caii deja devine de nemasurat sau tinde spre infinit in cazul metodei *DEPTH-FIRST-GRAPH-SEARCH*



4.6 Compare_searcher

```
compare_searchers(problems=[carsMiss, carsMht],
                  header=['Searcher', 'A* h1(n)',
                          'A* h2(n)'], searchers=[
        depth_first_graph_search,
        depth_first_graph_search])
```

Aceasta functie am preluat-o din lucrarea de la laborator 5 din fisierul *fifteen_puzzle* si am apelat-o pentru a vedea diferentele dintre cele doua metode de cautare pe care le-am ales sa le compar .

In imaginea de mai sus functia este folosita pentru comparatia a aelesasi metode de cautare insa diferenta consta in calcularea distantei

```
"C:\Users\ciont\Desktop\Facultate\Inteligenta Artificiala\venv\Scripts\python.exe"
[<Pas: (1,)>
]

[<Pas: (1,)>
]

Searcher                  A* h1(n)                A* h2(n)
depth_first_graph_search  <   0/   1/   0/(1,)>  <   0/   1/   0/(1,)>
astar_search              <   0/   1/   0/(1,)>  <   0/   1/   0/(1,)>

Process finished with exit code 0
```

Mai sus este prezentata functia pentru o parcare de 1 x 1 si dferenta dintre cele 2 metode alese

```
"C:\Users\ciont\Desktop\Facultate\Inteligenta Artificiala\venv\Scripts\python.exe" "C:/Users/cio
[<Pas: (1, 2, 0, 0)>
, <Pas: (0, 2, 1, 0)>
, <Pas: (0, 0, 1, 2)>
, <Pas: (1, 0, 0, 2)>
, <Pas: (0, 1, 0, 2)>
, <Pas: (0, 1, 2, 0)>
, <Pas: (0, 0, 2, 1)>
]

[<Pas: (1, 2, 0, 0)>
, <Pas: (1, 0, 0, 2)>
, <Pas: (0, 1, 0, 2)>
, <Pas: (0, 1, 2, 0)>
, <Pas: (0, 0, 2, 1)>
]

Searcher                  A* h1(n)                        A* h2(n)
depth_first_graph_search  <   5/   6/  11/(0, 0, 2, 1)>   <   5/   6/  11/(0, 0, 2, 1)>
astar_search              <   7/   9/  17/(0, 0, 2, 1)>   <   7/   9/  17/(0, 0, 2, 1)>
```

Mai sus este prezentata functia pentru o parcare de 2 x 2 si pasii de executie

```
earcher                   A* h1(n)                                    A* h2(n)
epth_first_graph_search   < 434/ 434/1346/None>                       < 444/ 444/1353/None>
star_search               <  56/  58/ 176/(0, 0, 0, 0, 0, 0, 3, 2, 1)>   < 189/ 191/ 576/(0, 0, 0,

rocess finished with exit code 0
```

Mai sus este prezentata functia pentru o parcare de 3 x 3 si dferenta dintre cele 2 metode alese

5 Concluzii

In urma acestui Assigment mi-am dezvoltat abilitatile de codare in limbajul Python, mi-am intarit cunostintele despre Search Problem si mi-am inbunatatit abiliattiel de a scrie in LATEX.

Acest Assigment a fost o adevarata provocare , am intampinat cateva greutati la adaptatrea problemei folosind frameworkul de la *fifteen_puzzle* si *eight_puzzle*, cum sa fac sa mut fiecare masina cate un pas pe tura si mai ales testarea si gasirea unei euristici potrivite.

Am incercat sa respect fiecare cerinta din metodologie, astfel incat sa pot descrie fiecare parametru corespunzator acesteia in functie de implementarile, abordarea si rezultatelepe care le-am justificat mai sus

6 Referinte

https://classroom.google.com/u/1/c/Mjc5NTAwMTYxNjky/m/MzI0MjgOMjYxMzkx/details (frameworkul de la laboratorul 5)

https://cs.calvin.edu/courses/cs/344/kvlinden/resources/AIMA-3rd-edition.pdf

https://github.com/aimacode/aima-pseudocode/blob/master/aima3e-algorithms.pdf

https://www.w3schools.com/python/ref_list_append.asp

http://aima.cs.berkeley.edu/

https://www.overleaf.com/learn/latex/Tutorials

https://www.w3schools.com/python/

https://en.wikibooks.org/wiki/LaTeX/Colors

https://stackoverflow.com/questions/227459/how-to-get-the-ascii-value-of-a-ch

http://openbookproject.net/thinkcs/python/english3e/tuples.html

https://www.geeksforgeeks.org/a-search-algorithm/

https://cse442-17f.github.io/A-Star-Search/

https://www.ics.uci.edu/~kkask/Fall-2016%20CS271/slides/03-InformedHeuristicSearch.pdf