RAPORT TEHNIC

Universitatea din Craiova, Facultatea de Automatică, Calculatoare și Electronică

Artificial Intelligence Homework

Student : *Drăghici Andreea-Maria*

Grupa : *CR 2.1 B*

Anul de studiu : *II*

Specializarea : *Calculatoare Română*

◄ ***Iunie 2021*** ►

Rezumat

Acest raport este o introducere in obiectivele de dezvoltare a temei de casa la disciplina Inteligenta Artificiala.

Introducere cuprins

1 ENUNTUL PROBLEMEI

1.1 *Homework statement*

Let us assume that m vehicles are located in squares (1, 1) through (m, 1) (the bottom row) of an $m \times m$ 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.

1.2 *Requirements*

The task is to analyze the problems and fulfill the following requirements:

R1. Implement the appropriate code to solve the assigned problem as a search problem.

R2. Present and comment your experimental results and choices in a meaningful way.

2 EXPLICATIA METODEI ALESE

Căutare euristică

2.1 *Descrierea si intelegerea problemei*

Cautarea eurisitica foloseste informatii suplimentare pentru ghidarea procesului de descoperire a unei solutii. In cazul de fata, realizand un anumit pas, fiecare vehicul se poate deplasa in mai multe sensuri, sau poate ramane nemiscat. Dar, daca un vehicul nu este mutat, cel mult un alt vehicul poate sari peste el, deoarce doua vehicule nu pot ocupa aceeasi celula (acelasi loc de parcare).

Deducem ca masinile se misca simultan (astfel incat o masina se poate deplasa in locul altei masini daca cealalta masina se muta), iar doua masini nu au voie sa se deplaseze in aceeasi celula. Rezulta ca mutarea unei masini in orice directie (inclusiv saritura peste o alta masina) implica un cost de 1, in timp ce statul pe loc al unei masini nu are cost. Costul total in orice etapa este suma costului pentru fiecare masina.

Scopul este sa mutam toate masinile la randul de sus, dar in ordine inversa, astfel incat masina care porneste din celula (i, 1) sa ajunga in celula ($m - i + 1, m$).

Problema este adesea modelata ca un spatiu de stare, un set de stari in care problema poate fi configurata. Se formeaza un graf din ansamblul starilor in care exista o legatura intre doua stari daca exista o operatie care poate transforma starea actuala in cealalta stare. Deci trebuie sa gasim o cale cu cel mai mic cost dintr-un anumit nod initial catre un nod final sau obiectiv.
Cu cat acuratetea cu care aceasta functie descrie lungimea celei mai scurte cai pana la nodul obiectiv este mai mare, cu atat volumul de calcul necesar pentru evaluarea functiei euristice creste.

2.2 *Abordarea problemei*

Pentru fiecare celula alocam un numar de la 0 pana la dimensiunea parcarii ridicate la puterea 2. Locatia 0 reprezinta prima pozitie in parcare, adica pozitia (1,1) si asa mai departe.

Exemplu:
1 , 1 $\leftarrow$ 0
1 , 2 $\leftarrow$ 1
m , m $\leftarrow parking\_size^2$

Starea initiala : – reprezinta pozitia fiecarui vehicul m.
Deci, in cazul de fata starea initiala reprezinta pozitia de pornire pentru fiecare vehicul, care este prima coloana a unei matrice, dar fiecare pozitie i, j din matrice este notata cu numere incepand de la 0 până la dimensiunea parcarii astfel, daca i = 1 si j = 1, pozitia va fi 0; pentru i = 1 și j = 2, pozitia va fi 1 si asa mai departe pana la ultima pozitie, care va fi egal cu numarul variabilei parking_size, adica dimensiunea parcarii.

Exemplu: Daca dimensiunea parcarii este egala cu 4, rezulta ca starea initiala va fi (0, 4, 8, 12).

Starea finala (obiectivul) : – reprezinta pozitiile in care trebuie sa ajunga fiecare dintre vehiculele m la randul de sus, dar in ordine inversa.
Deci, in cazul de fata pentru fiecare (i, 1), destinatia finala va fi ($m - i + 1, m$). Conform abordarii mele, destinatia finala va fi ($parking\_size^2$ - initialState(i) - 1)

Exemplu: Daca dimensiunea parcarii este egala cu 4, rezulta ca starea finala va fi (15, 11, 7, 3).

Actiunile vehiculelor : – reprezinta actiunile pe care fiecare vehicul le poate executa in functie de locatia sa in parcare si de pozitia altor vehicule.
Conform specificatiilor cerintei din tema de casa, fiecare vehicul poate executa mai multe actiuni pentru a atinge starea finala intr-un mod admisibil; Astfel fiind dat, fiecare vehicul se poate deplsa la dreapta, stanga, sus, jos, poate sa stea pe loc, sau sa executate anumite salturi peste alte masini in functie de locatia sa in parcare sau de pozitia celorlalte vehicule.

Exemplu: Pentru un vehicul situat in locatia 0 din parcare, adica pozitia (1,1), prima pozitie, acesta nu poate urca si nu poate pleca, deoarece ar iesi din zona parcarii in afara, respectiv nu poate face o mutare in jos pentru ca un alt vehicul exista in locatia respectiva si cunosatem faptul ca doua masini nu au voie sa se deplaseze in aceeasi celula.

Observatie:

Figura de mai jos nu-mi apartine, este doar un exemplu pentru intelegerea problemei din tema de casa.
O imagine orientativa asupra actiunilor executate de fiecare masina.

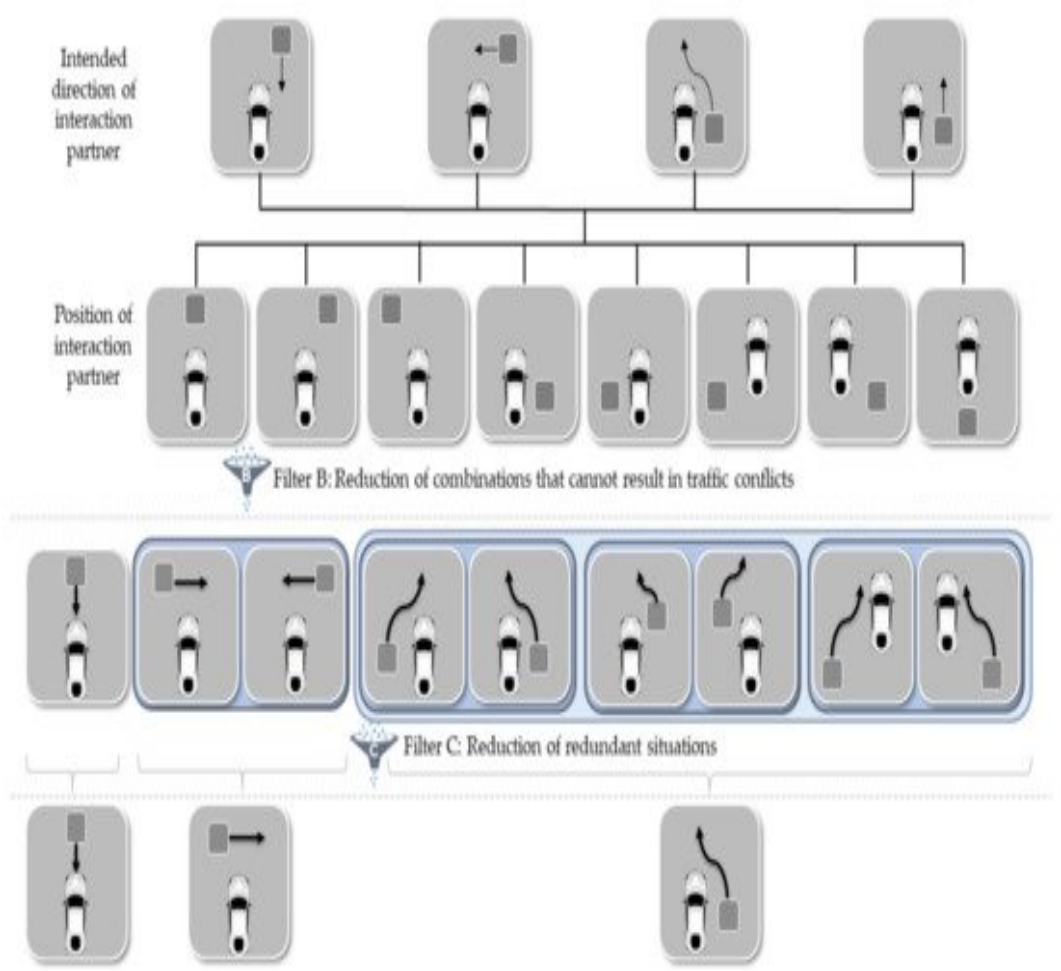


Figura 1: Intelegerea actiunilor care realizeaza salturile/mutarile masinilor

3 Pseudo-codul algoritmilor propusi

Algoritmii pe care i-am folosit pentru aceasta problema sunt divizati in mai multe functii in clasa numita m_Vehicle care mosteneste clasa din fisierul search.py, Problem. La baza rezolvarii acestei probleme am folosit ca algoritmi de cautare, best first search, A* search, folosindu-ma si de distanta Manhattan, pentru a determina costul exact al deplasarii masinilor din starea initiala catre starea finala sau obiectiv.

Pseudo-codul algoritmilor implementati va fi prezentat mai jos, atat pentru algoritmii de cautare pe care ii folosesc, cat si pseudo-codul pentru fiecare functie implementat in cadrul rezolvarii temei de casa:

function BEST_FIRST_SEARCH ($problem, f$) **returns** a solution node or none

1. $node \leftarrow$ a node with STATE = $problem$.INITIAL , PATH-COST = 0
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** $none$

end function

Denumirile variabileleor in Python, cat si implementarea difera din punct de vedere al sintaxei comparativ cu algoritmul in pseudo-cod, deoarece am incercat sa scriu algoritmii in pseudo-cod cat mai simplu si concins pentru a fi cititi si intelesi mai usor, implementarile din Python fiind adaptate in functie de fiecare algoritm in parte.
Specific faptul ca functiile de cautare fac parte din framework-ul aferent laboratorului 5.

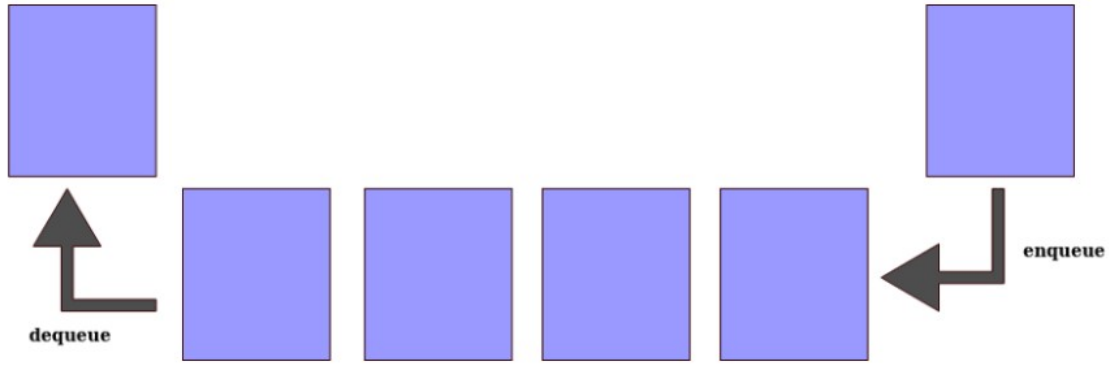


Figura 2: O coada care urmareste nodurile urmatoare pentru a fi vizitate in continuare

function ASTAR_SEARCH ($node$)

```
1.  a priority queue with priorities having as a function of cost on f (n)
2.  while the destination node has not been reached do
3.  consider the node with the lowest f score in the open list
4.      if this node is our destination node then
5.        return finished
6.      end if
7.      else
8.        put the current node in the closed list
9.        for each neighbor of the current node
10.           if neighbor has lower g value then current and is in the closed list then
11.             replace the neighbor with the new, lower g value
12.             current node is now the neighbor's parent
13.           end if
14.           else if current g value is lower and this neighbor is in the open list then
15.               replace the neighbor with the new, lower, g value
16.               change the neighbor's parent to our current node
17.           end else if
18.           else if this neighbor is not in both list then
19.               add it to the open list and set its g
20.           end else if
21.       end for
22. end while
```

end function

Am ales BEST-FIRST-SEARCH si A* search pentru a putea realiza un grafic compartiv pentru cei doi algoritmi in functie de costul caii, starea initiala, starea finala si in functie de actiunile executate de fiecare vehicul. Am observat ca best-first-search este mult mai optim din punct de vedere al timpului de executie decat A* search.
Dar ambii algoritmi fac parte din categoria algoritmilor "cautare intai-cel-mai-bun". Mai multe detalii vor fi precizate in sectiunea rezumatul rezultatelor.

Mai jos voi descrie pseudocod-ul pentru fiecare functie implementata in modulul implementation.py.

function MANHTDISTANCE($self, node$) **returns** the sum off all Manhattan distances

1. $node \leftarrow$ represents the state of a certain vehicle represented as a node
2. $manhattanDistance \leftarrow sum(abs(s - g) * 2\ for(s, g)\ in\ zip(node.state, self.goal))$
3. **return** $manhattanDistance$

end function

Presupunem ca exista o singura masina care porneste de la (*i*, 1) si ca problema ar fi mutarea masinii respective pana la locatia sa.

Pornind de la formula distantei Manhattan, adica h_i = $|x_1 - x_2| + |y_1 - y_2|$ pentru doua puncte din plan, $p_1 = (x_1, y_1)$ si $p_2 = (x_2, y_2)$, putem face afirmatia ca in cazul problemei noastre
h_i = | ($m - i + 1$) - x_i | +|$m - y_i$ | pentru a determina costul exact al deplasarii masinii catre obiectiv, iar scopul nostru este sa mutam toate masinile pe randul de sus, dar in ordine inversa astfel incat masina care porneste din celula (*i*, 1) sa ajunga in celula ($m - i + 1, m$).

Daca ar exista alte masini in celula respectiva, aceasta nu ar mai fi neaparat o euristica admisibila, deoarece orice masina ar putea sa sara peste alte masini permitandu-i sa ajunga la obiectiv in mai putini pasi decat daca folosim distanta Manhattan. Si cunoastem faptul ca o functie euristica este admisibila daca poate sa estimeze in mod eficient distanta reala dintre un nod *n* si nodul final, obiectiv.

Aceasta functie returneaza suma tuturor distantelor de la starea initiala la starea finala, fiind si functia care este folosita ca euristica pentru problema cautarii. Folosim distanta Manhattan deoarece ne ofera cel mai admisibil rezultat pentru aceasta problema, dar nu intocmai cel mai optim din punct de vedere al resurselor.

Putem lua in considerare o retea 2D care are mai multe obstacole si pornim de la o celula sursa (colorata rosu) pentru a ajunge la o celula obiectiv (colorata verde). Vrem sa atingem celula obiectiv (daca este posibil) pornind din celula de pornire cat mai repede posibil.

- **punctul rosu** = celula sursa
- **punctul verde** = celula tinta
- **patratele cu negru** = obstacole

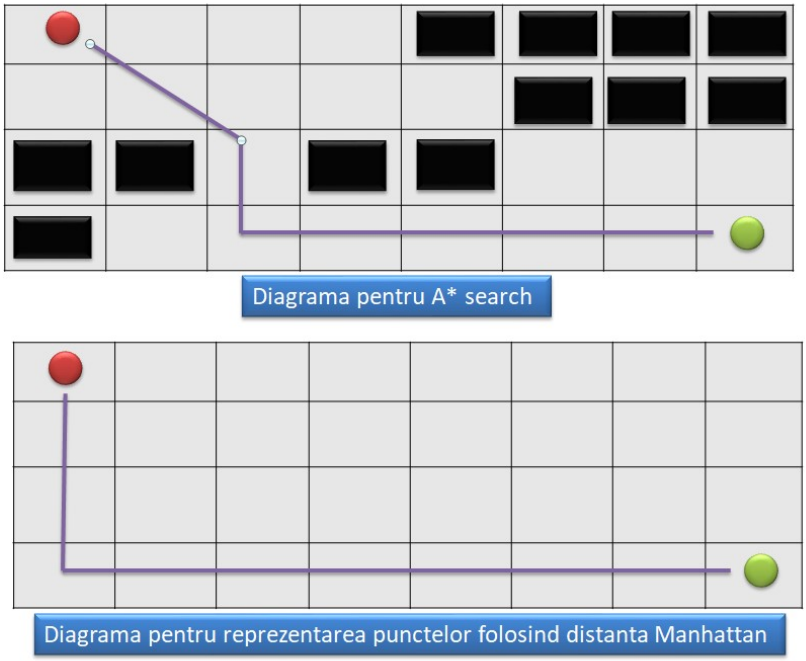


Figura 3: Diagrame pentru o mai usoara abordare a problemei

Observatie : Atasez mai jos link catre site-ul de unde m-am documentat, specific ca diagramele atasate mai sus au fost realizate in PowerPoint.
https://www.geeksforgeeks.org/a-search-algorithm/

Mai jos este prezentat pseudo-codul pentru crearea **starii initiale si scop**.
Initializam problema cu dimensiunea parcarii, starea initială si starea obiectivului.

function __INIT__($self, parking\_size, stare\_initiala, stare\_finala$)

1. $self.parking\_size \leftarrow parking\_size$
2. $self.initial \leftarrow stare\_initiala$
3. $self.goal \leftarrow stare\_finala$
4. $self.mutare\_masina \leftarrow -1$
5. **Problem.__init__**$self, initial, goal$

end function

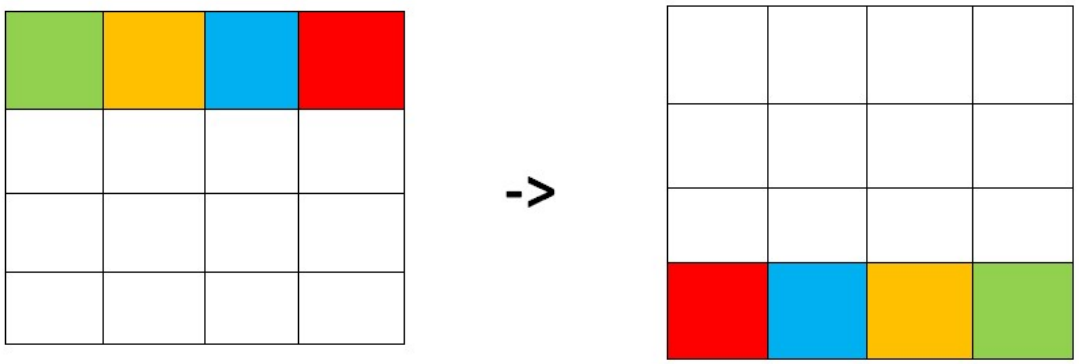
Masinile sunt reprezentate in figura 1 pentru a le putea diferentia si ordona in sens invers pe ultima linie.
Numarul de moduri de a plasa *m* masini distincte in *m* × *m* celule este :

$$m^2(m^2 - 1) \ldots (m^2 - m + 1) \in O(n^{2n})$$

Fiecare masina are cel mult 5 miscari posibile din orice pozitie (UP, DOWN, LEFT, RIGHT si STAY). Retinem ca daca fiecare masina are o celula 3 x 3 de spatiu gol in jurul ei rezulta ca toate cele cinci miscari sunt posibile pentru fiecare masina si acestea vor avea ca rezultat configuratii unice (deoarece masinile nu vor interactiona).
Cand m este suficient de mare avem suficiente celule de 3 x 3 pentru a sustine toate masinile posibile.

Exemplu:
Luam in considerare cand m = 12. Exista 16 celule 3 x 3 rezulta ca putem sustine toate cele 12 masini si chiar urmatoarele doua pe masura ce marim dimensiunea celulei.
De fiecare data cand crestem m cu 3, putem sustine in mod clar cel putin doua masini care depasesc dimensiunea parcarii. Daca masina *i* se afla la destinatie, atunci valoarea euristica este 0, care reprezinta si costul exact, rezulta, prin urmare, acesta este admisibil, altfel cand un vehicul efectueaza un salt peste alt vehicul, masina peste care sare nu poate fi la destinatie, ceea ce ar duce la o contradictie, astfel fiind dat masina care este sarita trebuie sa aiba cel putin un pas ramanand pana la destinatie.

Concluzia:
Eliminam anumite actiuni in functie de pozitia in parcare a masinilor, spre exemplu sa nu faca miscari in afara matricei(parcarii), sa nu faca salturi aiurea in afara matricei sau peste 2 masini. Fiecare actiune executata de ficare vehicul implica un cost = 1, iar factorul de ramificare in cel mai rau caz este = 5^n.



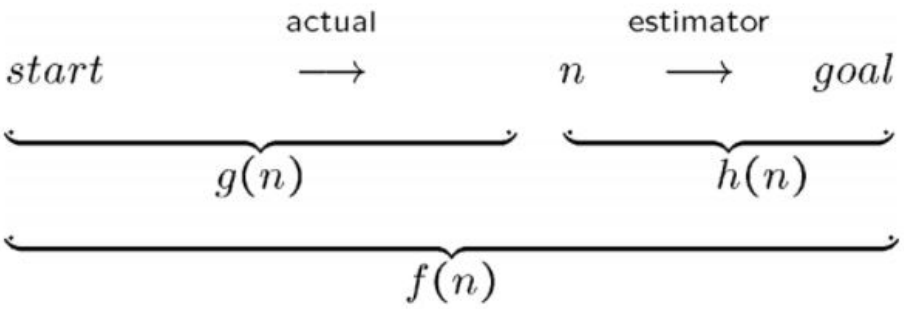


Figura 4: Starea initiala si obiectivul

initial state : configuratia initiala a masinilor in parcare.
goal state : configuratia finala a masinilor in parcare.

Scopul este sa mutam toate masinile pe randul de sus, dar in ordine inversa, astfel incat masina care porneste din celula $(i, 1)$ sa ajunga in celula $(m - i + 1, m)$.

Din cerinta problemei deducem faptul ca fiecare masina are cel mult 5 miscari posibile din orice pozitie (SUS, JOS, STANGA, DREAPTA si STAY) plus salturile/sariturile pe care le poate executa fiecare vehicul.
Adica fiecare vehicul poate executa mai multe actiuni pentru a atinge starea finala intr-un mod admisibil, in functie de locatia sa in parcare sau de pozitia celorlalte vehicule, acesta se poate deplasa inainte, inapoi, la stanga, la dreapta, respectiv poate executa salturi peste celelalte vehicule(daca este permis), sau poate sta pe loc in celula pe care deja o ocupa.
Calculam urmatoarea mutare a vehicului iar, daca vehiculul dinainte a executat o mutare atunci pozitia vehicului va deveni 0 si va incepe din nou cu primul vehicul, altfel trecem la pozitia urmatorului vehicul.

Verificam directiile pe care vehiculul nostru le poate realiza, conform parcarii, apo verificam actiunile pe care le poate face vehiculul nostru, conform celorlalte vehicule si eliminam anumite actiuni in functie de pozitia vehiculelor.
Returnam actiunile care pot fi executate in starea data, iar rezultatul este o lista, deoarece exista doar cinci actiuni posibile, plus salturile/sariturile pe care le poate executa fiecare vehicul.
Exista si anumite reguli, cum ar fi:

- 2 vehicule nu poate imparti acelasi loc de parcare, deci daca un vehicul este deasupra altui vehicul, acest vehicul nu poate sa coboare, deci actiunile descendente vor fi eliminate din lista cu actiuni posibile.
- cand un vehicul nu poate iesi in afara parcarii, asta inseamna ca atunci cand este plasat pe primul rand, nu poate merge in sus, astfel incat actiunile sus vor fi eliminate din lista actiunilor posibile.

Aceasta functie va fi utilizata pentru fiecare vehicul, tinandu-se cont de reguli, respectiv cazurile in care se afla fiecare vehicul.

Mai jos este prezentat pseudo-codul pentru implementarea functiei care verifica directiile pe care vehiculul nostru le poate realiza, conform parcarii.

function ACTIONS $(self, stare)$ **returns** an action

1. **if** $self.mutare\_masina == self.parking\_size - 1$ **then**
2. $self.mutare\_masina \leftarrow 0$
3. **else**
4. $self.mutare\_masina \leftarrow self.mutare\_masina + 1$
5. **end if**
6. **ListaActiunilorPosibile** $\leftarrow list['SUS' + str(self.mutare\_masina), 'JOS' + str(self.mutare\_masina), 'STANGA' + str(self.mutare\_masina), 'DREAPTA' + str(self.mutare\_masina), 'STAY' + str(self.mutare\_masina), 'SARITURA\ STANGA' + str(self.mutare\_masina), 'SARITURA\ DREAPTA' + str(self.mutare\_masina), 'SARITURA\ INAINTE' + str(self.mutare\_masina), 'SARITURA\ INAPOI' + str(self.mutare\_masina)]$
7. **locatia1** $\leftarrow state[self.mutare\_masina]$
8. **if** $locatia1 < self.parking\_size$ **then**

```
9.      if 'SUS' + str(self.mutare_masina) in ListaActiunilorPosibile then
10.       ListaActiunilorPosibile.remove('SUS' + str(self.mutare_masina))
11.     end if
12.   end if
13.   if locatia1 >= self.parking_size * (self.parking_size - 1) then
14.     if 'JOS' + str(self.mutare_masina) in ListaActiunilorPosibile then
15.       ListaActiunilorPosibile.remove('JOS' + str(self.mutare_masina))
16.     end if
17.   end if
18.   if locatia1 modulo self.parking_size == 0 then
19.     if 'STANGA' + str(self.mutare_masina) in ListaActiunilorPosibile then
20.       ListaActiunilorPosibile.remove('STANGA' + str(self.mutare_masina))
21.     end if
22.   end if
23.   if locatia1 modulo self.parking_size == self.parking_size - 1 then
24.     if 'DREAPTA' + str(self.mutare_masina) in ListaActiunilorPosibile then
25.       ListaActiunilorPosibile.remove('DREAPTA' + str(self.mutare_masina))
26.     end if
27.   end if
28.   if locatia1 modulo self.parking_size <= 1 then
29.     if 'SARITURA STANGA' + str(self.mutare_masina) in ListaActiunilorPosibile then
30.       ListaActiunilorPosibile.remove('SARITURA STANGA' + str(self.mutare_masina))
31.     end if
32.   end if
33.   if locatia1 modulo self.parking_size >= self.parking_size - 2 then
34.     if 'SARITURA DREAPTA' + str(self.mutare_masina) in ListaActiunilorPosibile then
35.       ListaActiunilorPosibile.remove('SARITURA DREAPTA' + str(self.mutare_masina))
36.     end if
37.   end if
38.   if locatia1 < self.parking_size * 2 then
39.     if 'SARITURA INAINTE' + str(self.mutare_masina) in ListaActiunilorPosibile then
40.       ListaActiunilorPosibile.remove('SARITURA INAINTE' + str(self.mutare_masina))
41.     end if
42.   end if
43.   if locatia1 > self.parking_size^2 - 2 * self.parking_size - 1 then
44.     if 'SARITURA INAPOI' + str(self.mutare_masina) in ListaActiunilorPosibile then
45.       ListaActiunilorPosibile.remove('SARITURA INAPOI' + str(self.mutare_masina))
46.     end if
47.   end if
48.   idx ← 0
49.   while idx < self.parking_size do
50.    locatia2 == stare[idx]
51.   if idx ≠ self.mutare_masina then
52.           if locatia1 - self.parking_size == locatia2 then
53.             if 'SUS' + str(self.mutare_masina) in ListaActiunilorPosibile then
54.               ListaActiunilorPosibile.remove('SUS' + str(self.mutare_masina))
55.             end if
56.           end if
57.           if locatia1 + self.parking_size == locatia2 then
```

```
58.              if 'JOS' + str(self.mutare_masina) in ListaActiunilorPosibile then
59.                ListaActiunilorPosibile.remove('JOS' + str(self.mutare_masina))
60.              end if
61.            end if
62.            if locatia1 - 1 == locatia2 then
63.              if 'STANGA' + str(self.mutare_masina) in ListaActiunilorPosibile then
64.                ListaActiunilorPosibile.remove('STANGA' + str(self.mutare_masina))
65.              end if
66.            end if
67.            if locatia1 + 1 == locatia2 then
68.              if 'DREAPTA' + str(self.mutare_masina) in ListaActiunilorPosibile then
69.                ListaActiunilorPosibile.remove('DREAPTA' + str(self.mutare_masina))
70.              end if
71.            end if
72.            if locatia1 - 2 == locatia2 then
73.              if 'SARITURA STANGA' + str(self.mutare_masina) in ListaActiunilorPosibile then
74.                ListaActiunilorPosibile.remove('SARITURA STANGA' + str(self.mutare_masina))
75.              end if
76.            end if
77.            if locatia1 + 2 == locatia2 then
78.              if 'SARITURA DREAPTA' + str(self.mutare_masina) in ListaActiunilorPosibile then
79.                ListaActiunilorPosibile.remove('SARITURA DREAPTA' + str(self.mutare_masina))
80.              end if
81.            end if
82.            if locatia1 - 2 * self.parking_size == locatia2 then
83.              if 'SARITURA INAINTE' + str(self.mutare_masina) in ListaActiunilorPosibile then
84.                ListaActiunilorPosibile.remove('SARITURA INAINTE' + str(self.mutare_masina))
85.              end if
86.            end if
87.            if locatia1 + 2 * self.parking_size == locatia2 then
88.              if 'SARITURA INAPOI' + str(self.mutare_masina) in ListaActiunilorPosibile then
89.                ListaActiunilorPosibile.remove('SARITURA INAPOI' + str(self.mutare_masina))
90.              end if
91.            end if
92.        end if
93.        idx ← idx + 1
94.    end while
95.    return ListaActiunilorPosibile
end function
```

Observatie

Din punct de vedere al sintaxei si pentru a optimiza algoritmul, am modificat liniile 52-87 de mai sus in implementarea din Python, punand cele 2 instructiuni if intr-o singura instructiune if, folosind operatorul logic 'and', atat sintaxa de mai sus, cat si sintaxa modificata in Python returneaza acelasi rezultat, in cadrul pseudo-codului de mai sus nu am mai modificat sintaxa, m-am gandit la o imbunatatire a algoritmului in Python dupa ce scrisesem deja pseudo-codul aici.

Avand in vedere starea si actiunea, revenim la o noua stare care reprezinta rezultatul actiunii finale. Actiunea se presupune ca este o actiune valida.
Mai jos este prezentat pseudo-codul pentru implementarea functiei care returneaza noua stare calculata in functie de actiunile posibile. Starea noua reprezinta pozitia vehicului *i* si actiunea pe care o executa vehiculul.

function RESULT ($self, stare, actiune$) **returns** the new state calculated according to the possible actions

1. $locatia1 \leftarrow stare[self.mutare\_masina]$
2. $noua\_stare \leftarrow list[stare]$
3. **delta** $\leftarrow list['SUS' + str(self.mutare\_masina) : -self.parking\_size,$
   $'JOS' + str(self.mutare\_masina) : +self.parking\_size,$
   $'STANGA' + str(self.mutare\_masina) : -1,$
   $'DREAPTA' + str(self.mutare\_masina) : +1,$
   $'STAY' + str(self.mutare\_masina) : 0,' SARITURA\ STANGA' + str(self.mutare\_masina) : -2,$
   $'SARITURA\ DREAPTA' + str(self.mutare\_masina) : +2,$
   $'SARITURA\ INAINTE' + str(self.mutare\_masina) : -2 * self.parking\_size,$
   $'SARITURA\ INAPOI' + str(self.mutare\_masina) : +2 * self.parking\_size\ ]$
4. $noua\_stare[self.mutare\_masina] \leftarrow locatia1$+**delta**(actiune)
5. **return** $tuple(noua\_stare)$

end function

Am exprimat delta ca actiune pe care o poate executa un vehicul, adica retine rezultatul pentru fiecare actiune. Apoi returnam rezultatul actiunii ca fiind un tuplu.
Ca idee, avem in vedere o stare si returnam true daca starea respectiva este o stare a obiectivului nostru sau false, altfel.
Pentru calcularea modificarii pe care o actiune o face asupra starii avem mai multe cazuri:

– daca actiunea va fi „SUS", aceasta va scadea valoarea variabilei parking_size din starea actuala a vehiculului;
– daca actiunea va fi „JOS", adauga aceasta valoare la starea actuala;
– daca actiunea este la „DREAPTA" adauga 1 la starea actuala;
– daca actiunea este la „STANGA" scade 1;
– iar daca actiunea este „STAY" nu face nimic, deoarece vehiculul nu se misca, sta pe loc.

Tot aici, au fost luate in calcul si cazurile cand un vehicul poate efectua o saritura la dreapta, la stanga, inainte sau inapoi peste un alt vehicul, astfel incat obiectivul sa fie indeplinit intr-un mod admisibil, tinandu-se cont de locatia vehiculului respectiv in parcare sau de pozitia celorlalte vehicule, indeplinind cazurile enumerate mai sus.

4 DATE EXPERIMENTALE

Pseudo-codul algoritmiilor care ne genereaza formatul datelor de intrare este prezentat mai jos :

function GENERARE_DIMENSIUNE_PARCARE

1. **return** $random.randint(1, 7)$

end function

function GENERARE_STARE_INITIALA ($dimensiune$)

1. stare_initiala $\leftarrow$ tuple()
2. idx $\leftarrow$ 0
3. **while** idx $<$ $dimensiune$ **do**
4. stare_initiala $\leftarrow$stare_initiala$+$ $tuple([dimensiune * idx])$
5. idx $\leftarrow$ idx $+$ 1
6. **end while**
7. **return** stare_initiala

end function

function GENERARE_STARE_FINALA ($dimensiune$, $stare\_initiala$)

1. generare_stare_finala $\leftarrow$ tuple()
2. idx $\leftarrow$ 0
3. **while** idx $<$ $dimensiune$ **do**
4. stare_finala $\leftarrow$ stare_finala $+$ $tuple([(dimensiune^2) - stare\_initiala[idx] - 1])$
5. idx $\leftarrow$ idx $+$ 1
6. **end while**
7. **return** stare_finala

end function

Folosim o metoda pentru generarea automata a datelor de intrare, astfel incat generam aleatoriu dimensiunea parcarii, starea initiala si cea finala in functie de dimensiunea parcarii. Mai jos descriu care este metoda pentru generare, de ce folosesc o metoda de generare random a datelor de intrare si care este intervalul de generare pentru datele de intrare.

Folosim functia *r*andint pentru a returna un numar intreg aleatoru situat in intervalul (0, *r*andint_max), unde randint_max este valoarea maxima pe care o poate lua intervalul nostru generand aleatoriu toate valorile din intervalul respectiv. Cu ajutorul buclei *w*hile putem specifica limita unei secvente de valori, astfel functiile pentru generarea starii initiale, cat si pentru generarea starii finale o sa returneze un numar aleatoru selectat din intervalul nostru, dar nu mai mare de valoarea limita pentru dimensiunea parcarii.

Generam random dimensiunea parcarii *m* x *m*. Returnam aleatoriu un numar cuprins intre 1 si 7. Cum limbajul Python este mai lent din punct de vedere a timpului de executie, folosim functiile deja scrise din modulele limbajului.

Generam starea initiala cu pozitia de pornire pentru fiecare vehicul, care este prima coloana a unei matrice, dar fiecare pozitie i, j din matrice este notata cu valori incepand de la 0 pana la dimensiunea parcarii astfel, daca i = 1 si j = 1, pozitia va fi 0; pentru i = 1 și j = 2, pozitia va fi 1 si asa mai departe pana la ultima pozitie, care va fi egal cu numarul variabilei parking_size, adica dimensiunea parcarii.

Generam starea obiectivului, urmand aceeasi idee ca mai sus, dar folosind o o alta formula, deoarece fiecare vehicul trebuie sa ajunga la pozitia cu coordonatele (*m* - *i* + 1 , *m*), asta inseamna ca trebuie sa ajunga pe randul de sus, dar în ordine inversa. In conformitate cu cele enumerate mai sus, putem argumenta faptul ca datele generate de algoritmul nostru prezentat sunt semnificative pentru testarea problemei din tema de casa.

Acesti algoritmi genereaza aleatoriu dimensiunea parcarii *m* x *m* , starea initiala si starea finala; dimensiunea parcarii fiind data aleatoriu intre 1 si 7, iar starea initiala, cat si cea finala au valorile generate in functie de dimensiunea parcarii. Cum intervalul de generare a datelor este cuprins intre (1,7), rezulta ca datele de intrare sunt admisibile, dar, doar pana la o dimensiunea de *7* x *7*, deoarece doar intre aceste valori algoritmul ofera un raspuns.
Mai multe detalii despre rezultatele obtinute vor fi abordate in sectiunea rezumatul rezultatelor.

5 PROIECTAREA APLICATIEI

5.1 *Structura de nivel inalt a aplicatiei*

Aplicatia creata este organizata in module, fiecare modul continand functii particulare. Specific faptul ca pentru rezolvarea problemei din tema de casa am folosit framework-ul de la 15-puzzle problem din platforma de laborator 5, adaptand, testand si gasind o euristica aferent potrivita implementarii acestui assigment.

Astfel pentru rezolvarea problemei in limbajul Python, programul este divizat in mai multe fisiere. Fiecare modul contine functii ce ajuta la rezolvarea problemei, importandu-le direct din fisierele dorite. Functiile programului sunt apelate in fisierul main.py, ce reprezinta fisierul principal ce ruleaza programul.

Structura acestuia arata in felul urmator :

- main.py – fisierul principal ce ruleaza programul;
- implementation.py – fisierul unde sunt definite actiunile in functie de miscariile masinilor;
- generator_date_intrare.py – fisier ce contine implementate functii pentru generare random a datelor de intrare;
- search.py – fisier unde sunt apelate diverse functii de cautare;
- utils.py – fisier ce ofera anumite utilitati pentru rezolvarea problemei;

In fisierul generator_date_intrare gasim trei funtii implementate cu scopul de a genera random datele de intrare pentru dimensiunea parcarii, starea initiala si starea finala.

Am folosit aceasta structura modulara pentru ca la final sa putem observa mai bine diferenta dintre rezultatul final, cat si timpul necesar rezolvari problemei pentru anumite date de intrare random. Pe urmatoarea pagina intalnim o diagrama realizata in PowerPoint pentru structura metodei implementate in limbajul Python.

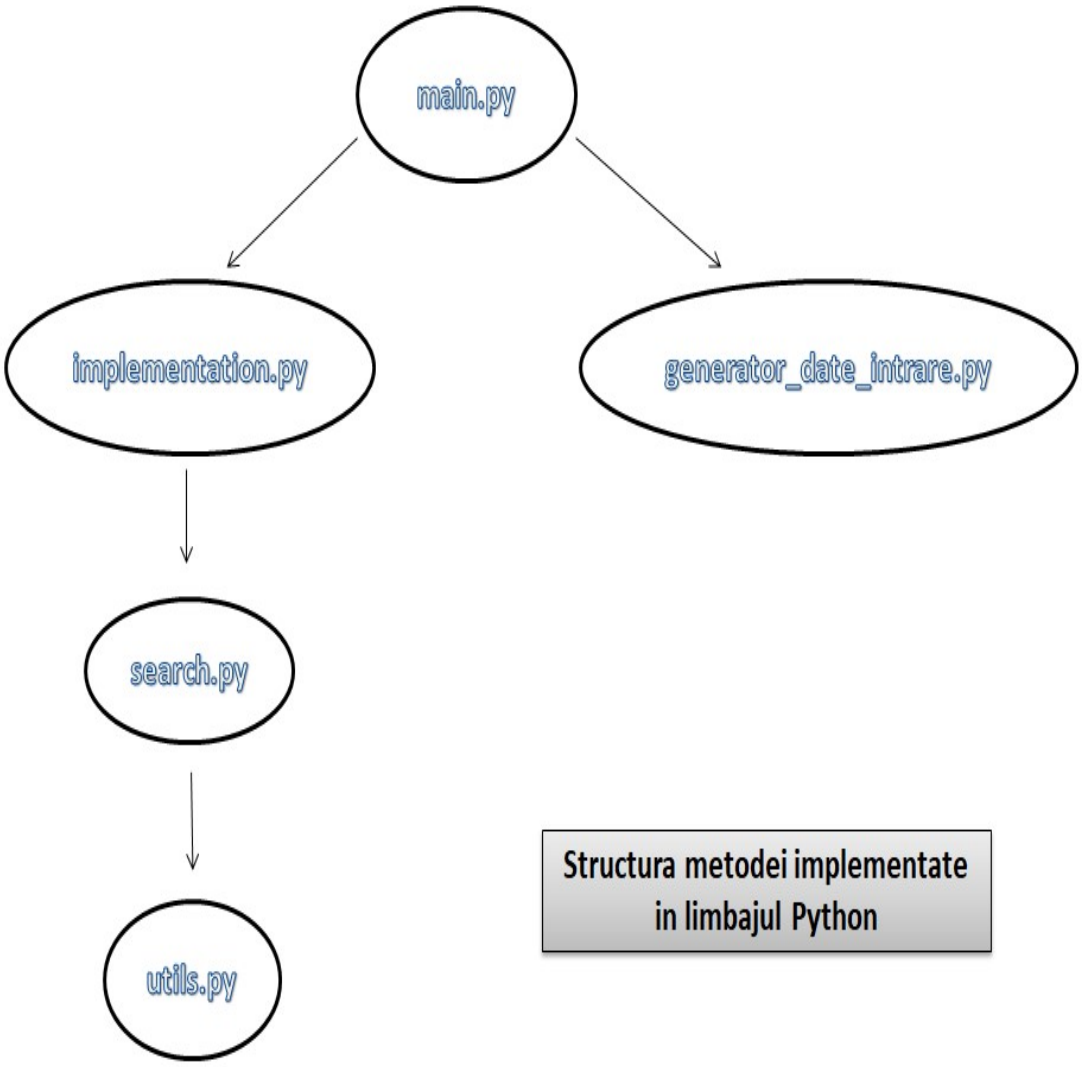


Figura 5: Structura de nivel inalt a aplicatiei

5.2 *Specificatia formatului datelor de intrare*

Pentru generarea datelor de intrare, am creat un nou fisier numit generator_date_intrare.py care contine implementate 3 functii care calculeaza dimensiunea parcarii in mod aleatoriu si, de asemenea, calculeaza starea initiala si starea obiectivului conform unor reguli matematice. Acestea genereaza aleatoriu date de intrare relevante si netriviale pentru testarea algorimului.

Datele de intrare reprezinta dimensiunea unei parcari sau a unei matrice $m \times m$, dimensiune ce este generata aleatoriu, iar m reprezinta dimenisiunea matricei; in functie de dimensiunea matricei generam si numarul de masini.

Fiecare element din matrice reprezinta o locatie in parcare si este reprezentat printr-un numar pozitiv si intreg semnificand cota locatiei respective in parcare.

Definim actiunile in functie de miscarile masinilor, adica calculam pentru fiecare stare care masina sa fie mutata pentru a putea muta doar o masina la un pas anume, tinand cont de specificatiile cerintei din tema de casa.

5.3 *Specificatia formatului datelor de iesire*

Datele de iesire sunt generate de functia astar search din fisierul search.py. Aceasta functie apeleaza o alta functie numita $best\_first\_graph\_search$ care calculeaza calea generata pentru problema noastra, adica datele de iesire.

Datele de iesire reprezinta o lista de actiuni:
$[SUS_i\ ,\ JOS_i\ ,\ DREAPTA_i\ ,\ STANGA_i\ ,\ STAY_i, SARITURA\ STANGA_i, SARITURA\ DREAPTA_i$
$SARITURA\ INAINTE_i, SARITURA\ INAPOI_i]$, unde *i* reprezinta defapt indicele starii vehiculului *i* care este folosit cu scopul de a ne ajuta sa vedem care vehicul a realizat actiunea.

Datele de iesire vor fi livrate nontrivial in fisiere, generand aleatoriu 7 valori, automat vor exista si 7 fisiere in concordanta cu fiecare dimensiune/valoare a parcarii, costul caii, timpul de executie, actiunile efectuate, starea initiala, cat si stare finala.
Actiunile efectuate vor fi afisate in fisiere sub forma de lista, un exemplu se afla putin mai sus specificat.

In consola vor fi afisate mutarile efectuate incepand cu starea initiala, pana la starea finala efectuand toate mutarile, combinatiile posibile, deoarce nu am gasit o modalitate de a afisa aceste cazuri utilizand operatiile cu fisiere.

5.4 *Lista tuturor modulelor aplicatiei si descrierea lor*

Aici voi realiza o mica prezentare a tuturor modulelor aplicatiei si o descriere a acestora, cu ce scop au fost folosite, de ce si la ce ne ajuta.
Am precizat la subparagraful 5.1 de la pagina 18 care este structura de nivel inalt a aplicatiei, urmand acum sa specific lista cu toate modulele cat si descierea acestora.
Incep mai intai prin a enumera toate functiile ce se gasesc in fisierul generator_date_intrare.py:

- **def** $generare\_dimensiune\_parcare()$**;**
- **def** $generare\_stare\_initiala(dimensiune)$**;**
- **def** $generare\_stare\_finala(dimensiune, stare\_initiala)$**;**

Functia **def generare_dimensiune_parcare()** este o functie fara argumente si este folosita cu scopul de a returna aleatoriu un numar intreg cuprins intre 1 si 7, dand o valoare mai mare a numarului, algoritmul nostru necesita un timp mai lung de calcul si genereaza mai greu un rezultat admisibil, de aceea numarul nostru este cuprins intre 1 si 7. Mai multe exemple vor fi prezentate in sectiunea rezultate si concluzii.

Functia **def generare_stare_initiala(dimensiune)** este o functie ce contine un argument care reprezinta dimensiunea parcarii noastre; acest argument este dat aleatoriu in prima functie prezentata mai sus. Aceasta functie returneaza o variabila numita stare_initiala in functie de cum a fost calculata, mai multe detalii se gasesc in paragraful 4 Date experimentale de la pagina 14.

Functia **def generare_stare_finala(dimensiune, stare_initiala)** este o functie ce contine doua argumente dimensiunea parcarii si stare_initiala, fiind utilizata pentru a calcula starea finala sau obiectivul. Aceasta functie ne returneaza o variabila stare_finala in functie de cum a fost calculata, mai multe detalii se gasesc in paragraful 4 Date experimentale de la pagina 14.

Acum urmeaza sa enumar functiile ce se gasesc in fisierul implementation.py:

- **def** $\_\_init\_\_(self, parking\_size, stare\_initiala, stare\_finala)$**;**
- **def** $actions(self, stare)$**;**
- **def** $result(self, stare, actiune)$**;**
- **def** $goal\_state(self, state)$**;**
- **def** $manhtDistance(self, node)$**;**

Functia **def __$init$__(self, parking_size,stare_initiala,stare_finala)** contine trei argumente cu scopul de a initializa dimensiunea parcarii, starea initiala si stare finala sau obiectivul cu valorile de care avem nevoie, argumentul self fiind utilizat cu scopul de accesa varibilele din interiorul clasei.

Functia **def actions(self,stare)** contine un singur argument, deoarece cum am specificat si mai sus, argumentul self este folosit pentru a accesa variabila din interiorul clasei, astfel argumentul stare reprezinta starea problemei noastre. Aceasta functie este folosita cu scopul de a vedea care vehicule realizeaza actiunile necesare si de a accesa pozitia unui vehicul *i*. Functia returneaza o lista numita ListaActiunilorPosibile care contine actiunile pe care un vehicul le poate face, tinand cont de specificatiile cerintei din tema, fiind utilizata pentru fiecare vehicul.

Functia **def result(self,stare,actiune)** este o functie ce contine doua argumente, stare care reprezinta starea si actiune ce reprezinta actiunea curenta efectuata de vehicul. Aceasta functie este folosita cu scopul de a returna modificarea pe care o actiune o poate face asupra starii unui vehicul *i*.

Functia **def manhtDistance(self,node)** este o functie ce contine un singur argument si reprezinta chiar implementarea functiei noastre euristice pe care o folosim pentru rezolvarea acestei probleme. Argumentul node poate fi privit ca starea unui vehicul doar ca reprezentata ca un nod. Am folosit distanta Manhattan deoarece consider ca imi ofera cel mai admisibil rezultat pentru aceasta problema.

In modulul main.py doar apelam functiile din modulele implementation.py si search.py si realizam cateva operatii cu fisiere, respectiv tratatrea unor posibile exceptii care ar aparea cand rulam programul. Acesta modul reprezinta fisierul principal ce ruleaza programul.

Observatie

Celelalte module, cum ar fi search.py si utils.py cuprind anumite functii de cautare, respectiv ofera anumite utilitati pentru rezolvarea problemei, acestea fac parte din framework-ul de la laboratorul 5, deci nu sunt implementate de mine, sunt doar utilizate, importate si folosite doar pentru a ajuta la rezolvarea temei de casa.

Totusi, o sa mentionez in raport si de aceste module, fiindca consider ca ar trebui sa fie si ele detaliate in cadrul acestei sectiuni, mai ales ca au fost utlizate si acestea cu un scop in cadrul rezolvarii temei de casa.

In modulul search.py intalnim o clasa abstracta pentru o problema formala, ce cuprinde implementate mai multe metode, cat si instante rezolvand diferite funtii de cautare.
In modulul utils.py sunt intalnite mai multe utilitati care ajuta la rezolvarea problemei noastre. Aceste utilitati sunt utilizate, importate pe scara larga de alte module. Aceste module ma ajuta in cadrul celorlalte functii din modulul implementation.py, acesta este scopul pentru care am apelat la framework-ul de la laboratorul 5.

6 REZUMATUL REZULTATELOR

In aceasta sectiune voi descrie printr-un mic rezumat rezultatele obtinute in urma rularii programului. Rezultatele au fost obtinute alateoriu in functie de generarea datelor de intrare. Astfel fiind dat, exista 7 cazuri pentru datele de iesire, deoarece am generat aleatoriu un numar intreg cuprins intre 1 si 7, fiindca dand o valoare mai mare, algoritmul necesita un timp mai lung de calcul a rezultatului si nu stim cand va fi generat, de aceea exista 7 cazuri.
Astfel cum datele de intrare au o valoare din ce in ce mai mare, timpul de executie va creste si el.

O alta observatie ar fi faptul ca si numarul de pasi executati pana la atingerea obiectivului creste. Cu cat dimensiunea parcarii este mai mare, cu atat numarul de pasi sau numarul de mutari executate de vehicule creste.

O a doua observatie este data de faptul ca si numarul actiunilor executate de vehicule creste in functie de dimensiunea parcarii, respectiv in functie de numarul de vehicule din parcare.

Observam ca best-first-search este mult mai optim din punct de vedere al timpului de executie decat A* search, dar ambii algoritmi fac parte din categoria algoritmilor "cautare intai-cel-mai-bun".

Mai jos am realizat doua tabele cat si doua grafice in PowerPoint si Excel pentru a vedea rezultatele obtinute in functie de dimensiunea parcarii, timpul executat de algoritm, cat si costul caii. Atat valorile din tabel,cat si cele din grafice/diagrame reprezinta valorile obtinute in urma rularii programului nostru.

Primul tabel, respectiv si primul grafic cuprind rezultatele obtinute in urma rularii programului pentru algoritmul A* search, iar al doilea tabel, respectiv cel de-al doilea grafic cuprind rezultatele obtinute in urma rularii programului pentru algoritmul Best-First-Search.
Am ales aceasta abordare de a folosi doi algoritmi pentru a putea compara mai usor rezultatele obtinute in urma rularii programului.

Atat rezultatele obtinute utilizand Astar search, cat si best first search sunt generate in fisiere pentru a se putea observa mai bine comparatiile intre timpul de excutie, costul caii cat si actiunile execuate de fiecare vehicul pentru cei doi algoritmi de cautare.

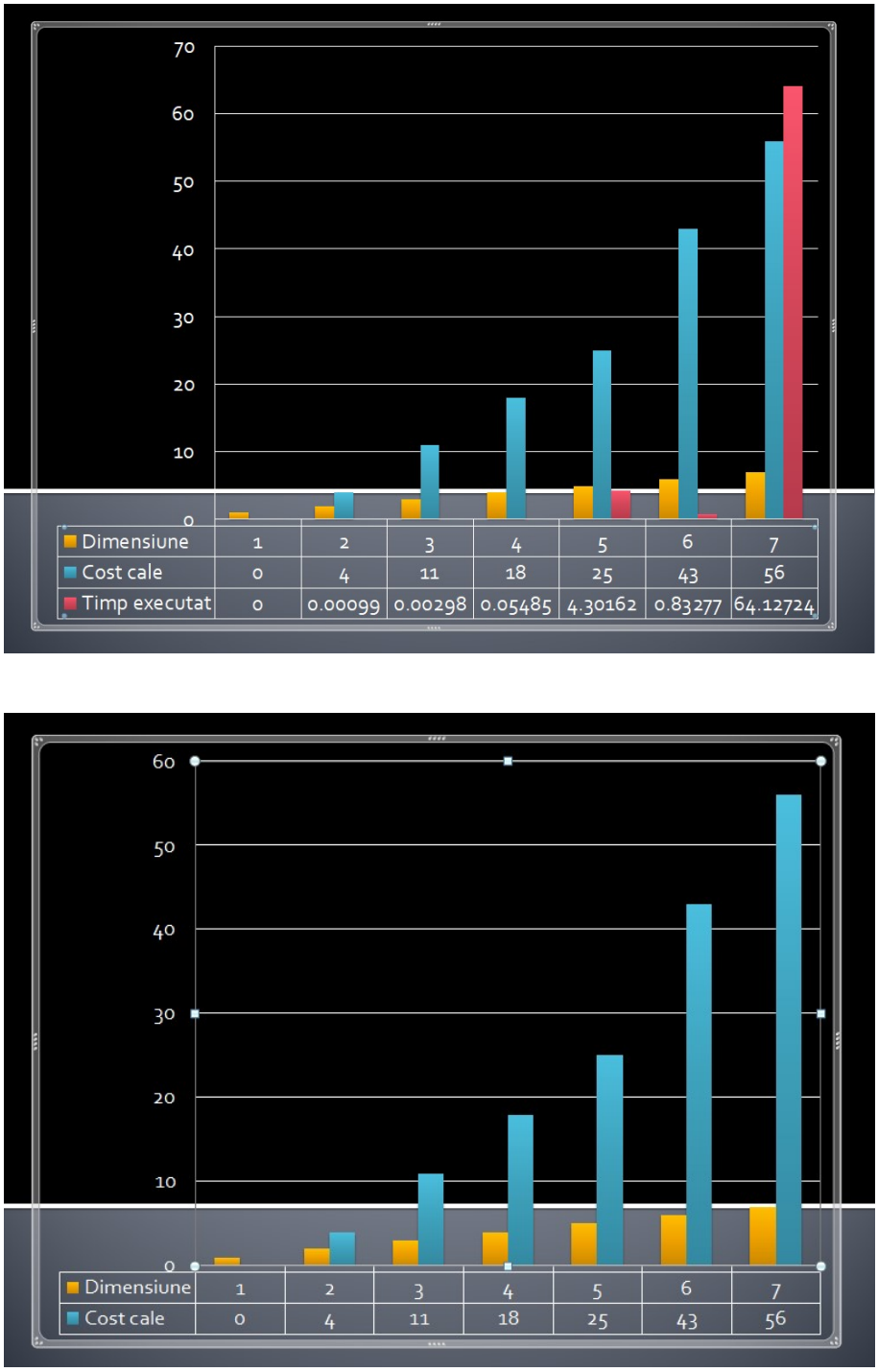
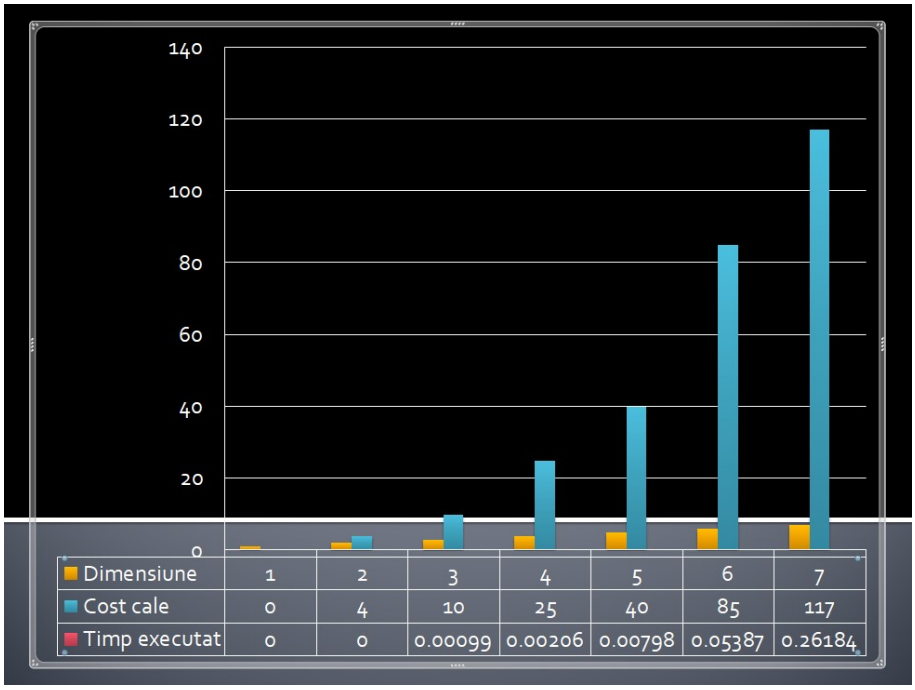


Figura 6: **Grafice-Rezultate comparative pentru Astar-search**



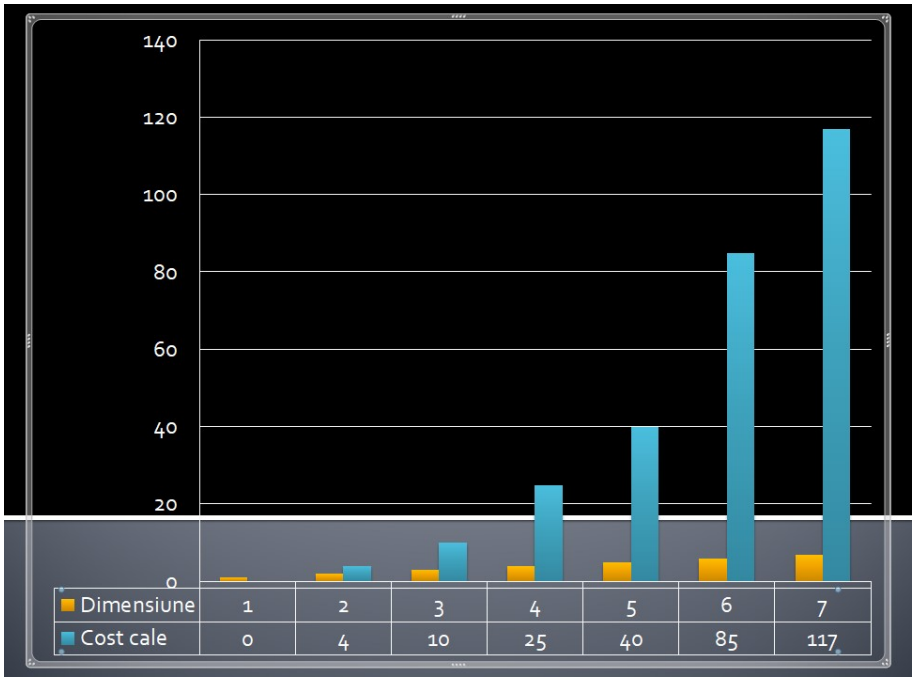


Figura 7: **Grafice-Rezultate comparative pentru Best-First-Search**

STAREA INITIALA	DIMENSIUNE PARCARE	NUMAR PASI EFECTUATI	STARE FINALA
(0,2)	2x2	4	(3,1)
(0,4,8,12)	4x4	18	(15,11,7,3)
(0,6,12,18,24,30)	6x6	43	(35,29,23,17,11,5)
(0,3,6)	3x3	11	(8,5,2)
(0,5,10,15,20)	5x5	25	(24,19,14,9,4)
(0,0)	1x1	0	(0,0)
(0,7,14,21,28,35,42)	7x7	56	(48,41,34,27,20,13,6)

Date obtinute apeland functia pentru Astar search.

STAREA INITIALA	DIMENSIUNE PARCARE	NUMAR PASI EFECTUATI	STARE FINALA
(0,2)	2x2	4	(3,1)
(0,4,8,12)	4x4	25	(15,11,7,3)
(0,6,12,18,24,30)	6x6	85	(35,29,23,17,11,5)
(0,3,6)	3x3	10	(8,5,2)
(0,5,10,15,20)	5x5	40	(24,19,14,9,4)
(0,0)	1x1	0	(0,0)
(0,7,14,21,28,35,42)	7x7	117	(48,41,34,27,20,13,6)

Date obtinute apeland functia pentru BEST-FIRST-SEARCH.

6.1 *Complexitatea computationala*

Pentru fiecare iteratie a buclei, A* trebuie sa determine calea pe care sa o extinda, astfel o face tinand cont de costul caii si a unei estimari a costului necesar pentru a extinde calea pana la obiectiv. Astfel costul total in orice etapa este suma costului pentru fiecare masina.

Mutarea unei masini in orice directie implica un cost de 1, in timp ce statul pe loc al unei masini nu are cost. In cazul de fata pentru fiecare test, costul caii va fi altul, acesta reprezentand suma costului pentru fiecare masina, numarul de actiuni executate de fiecare vehicul. Cu cat dimensiunea parcarii, respectiv numarul de actiuni executate de fiecare vehicul creste, cu atat si costul caii creste.

A* calculeaza calea in functie de urmatoarea formula, adica selecteaza calea care minimizeaza:

$f(n) = g(n) + h(n)$, unde:

- **n = nodul urmator de pe cale;**
- **g(n) = costul caii de la nodul initial pana la un nod final/obiectiv n;**
- **h(n) = functia euristica care estimeaza de fapt un cost mai ieftin pentru calea n pana la obiectiv, in cauzul nostru aceasta functie este chiar distanta Manhattan.**

In tabelul prezentat mai jos, putem observa performanta, cat si complexitatea spatiului in cel mai rau caz.

A* search	
Performance	Space complexity
$O(\|E\|) = O(b^d)$	$O(\|V\|) = O(b^d)$

Factorul de ramificare poate fi determinat prin masurarea numarului de noduri N si adancimea solutiei, astfel: $N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$, unde b^* = 1, iar d = numarul de actiuni executate in cazul problemei noastre, lungimea solutiei adica.

Best first search se implementeaza tratand multimea frontiera ca o coada cu prioritati avand drept functie de cost pe h(n). Va selecta la fiecare pas nodul din frontiera care "pare" a fi cel mai aproape de un nod obiectiv. Cu alte cuvinte se selecteaza nodul n care minimizeaza h(n). Best first search nu garanteaza intotdeauna gasirea unei solutii de cost minim.

In tabelul prezentat mai jos, putem observa performanta, cat si complexitatea spatiului in cel mai rau caz.

Best first search	
Performance	Space complexity
$O(\|E\|) = O(b^m)$	$O(\|V\|) = O(b^m)$

Factorul de ramificare poate fi determinat prin masurarea numarului de noduri n si adancimea solutiei, astfel: $T(n) = 1 + b + b^2 + \ldots + b^m$, unde b = factor de ramificare, iar d = numarul de actiuni executate in cazul problemei noastre, lungimea solutiei adica.

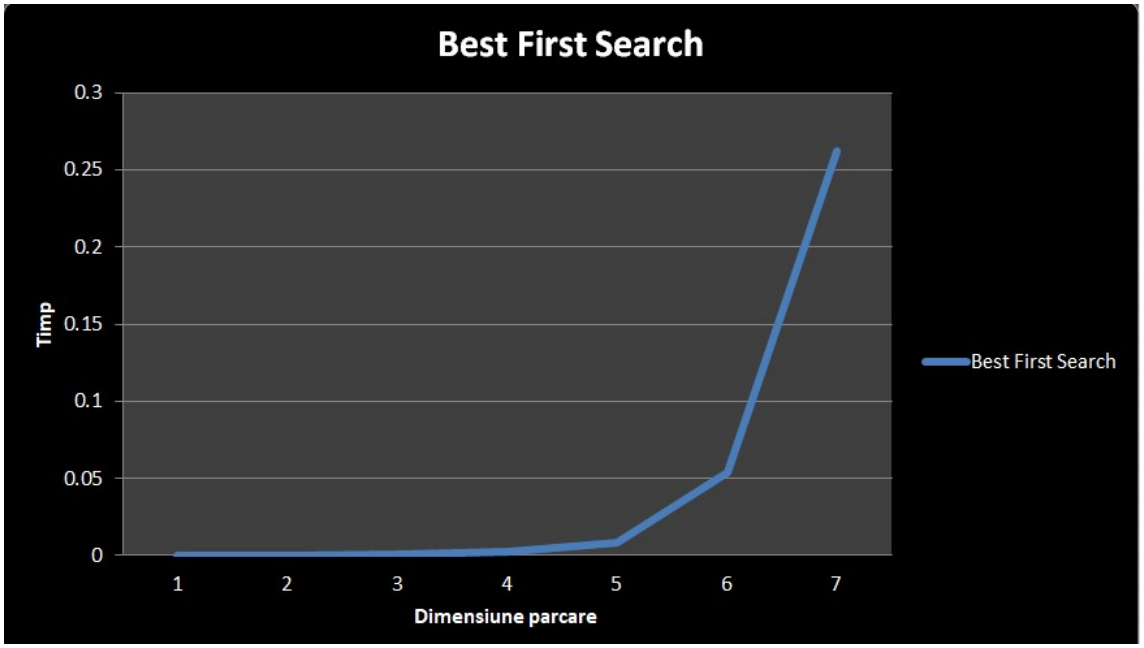


Figura 8: Grafic-Timp executie Best first search

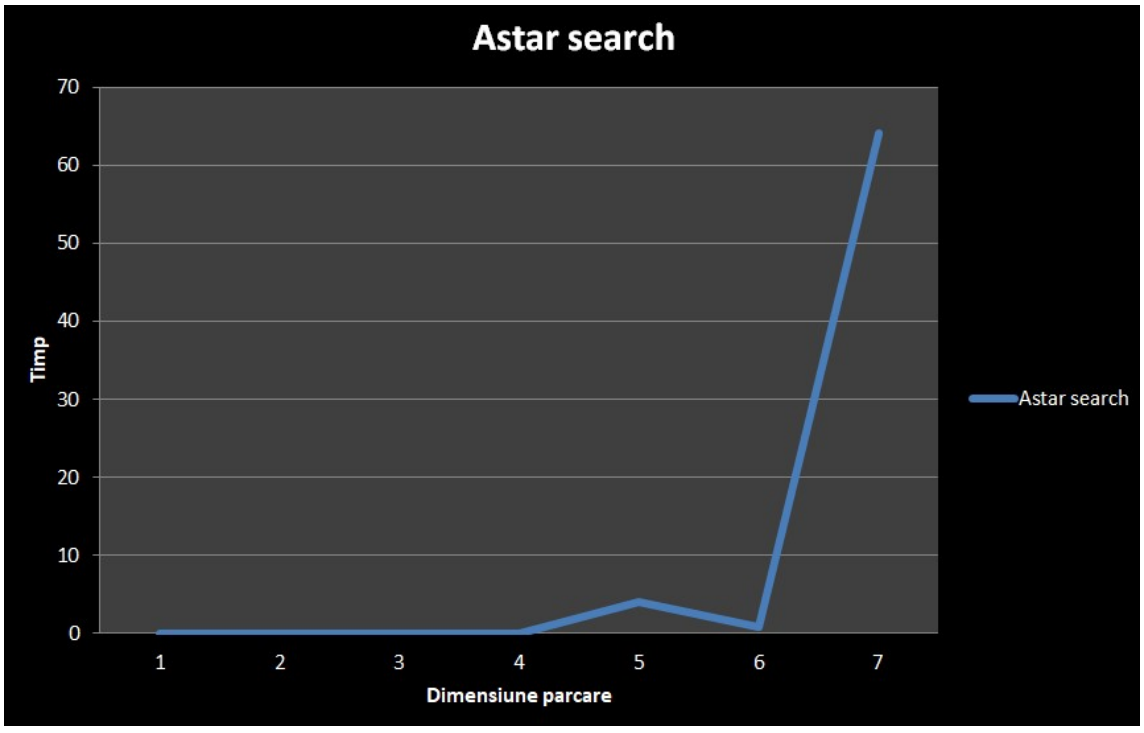
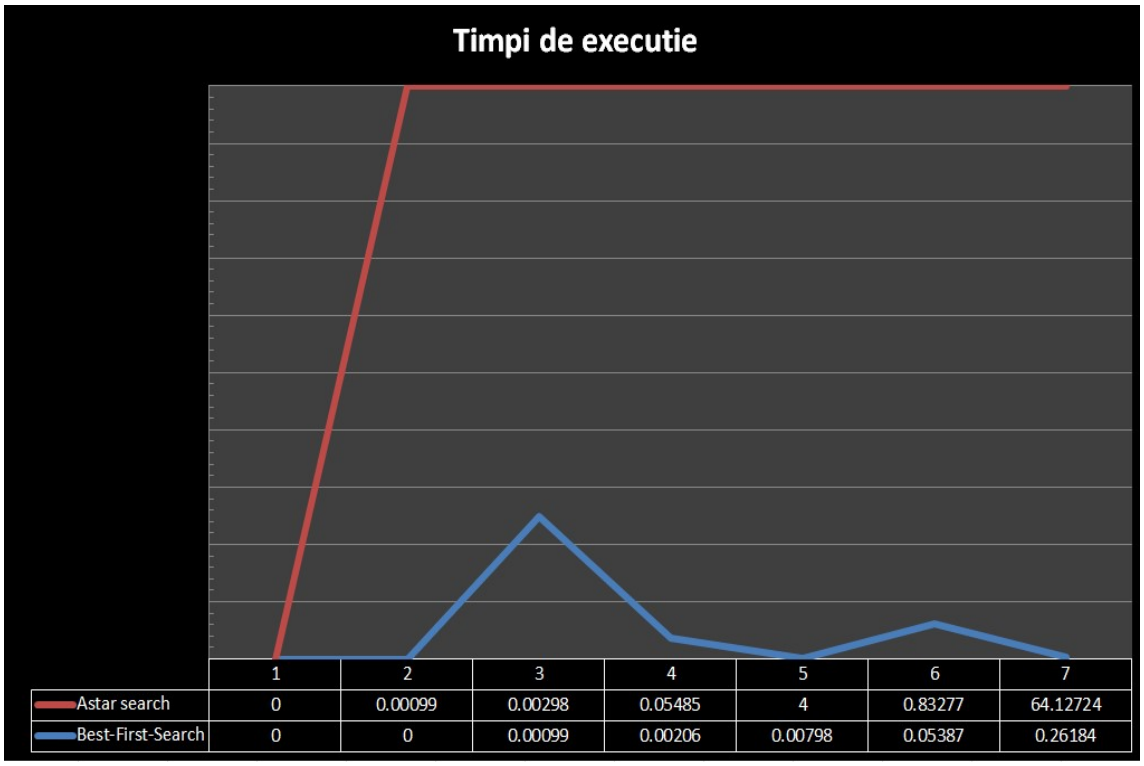


Figura 9: Grafic-Timp executie Astar search



	1	2	3	4	5	6	7
Astar search	0	0.00099	0.00298	0.05485	4	0.83277	64.12724
Best-First-Search	0	0	0.00099	0.00206	0.00798	0.05387	0.26184

Figura 10: Grafic comparativ intre timpii de executie

6.2 Concluzii rezultate comparative

- Din cele doua tabele, respectiv din graficele atasate pe urmatoarele doua pagini se poate observa faptul ca utilizand best first search-ul avem un timp de executie mai mic, comparativ cu timpul de executie obtinut cand folosim astar search;

- Se poate observa ca atunci cand apelam functia astar search, costul caii este mult mai mic, inclusiv numarul de pasi efectuati de fiecare vehicul pentru a ajunge la starea finala, dar timpul de executie este mult mai mare;

- Ambii algoritmi genereaza acelasi rezultat, dar difera costul caii, actiunile executate de fiecare vehicul, cat si timpul executat pentru a ajunge la starea finala;
- Ambii algoritmi apeleaza functia pentru distanta Manhattan care calculeaza euristica pentru problema cautarii in programul nostru;

- Un avantaj al utilizarii Best First Search in problema noastra este dat de viteza mai mare de a ajunge la starea finala;

- Un dezavantaj al utlizisarii Astar search in problema noastra este utilizarea unei cantitati mari de timp;

- Din primul grafic realizat, se poate observa ca atunci cand dimensiunea parcarii este *1* x *1*, timpul executat, cat si costul caii sunt 0, cu cat dimensiunea parcarii este mai mare rezulta ca si timpul executat creste, respectiv si costul caii. Se poate observa o crestere semnificativa a timpului de executie pentru o dimensiune de *7* x *7* a parcarii, *7* x *7* fiind si ultima valoare obtinuta pentru care algoritmul ofera un raspuns. Rezultatele din primul grafic sunt pentru algoritmul de cautare Astar search;

- Din al doilea grafic realizat, se poate observa ca atunci cand dimensiunea parcarii este *1* x *1*, timpul executat, cat si costul caii sunt 0, cu cat dimensiunea parcarii este mai mare rezulta ca si timpul executat creste cate putin, respectiv si costul caii. Se poate observa o crestere semnificativa a costului pentru cale la o dimensiune de *7* x *7* a parcarii, *7* x *7* fiind si ultima valoare obtinuta pentru care algoritmul ofera un raspuns. Rezultatele din al doilea grafic sunt pentru algoritmul de cautare BEST-FIRST-SEARCH.

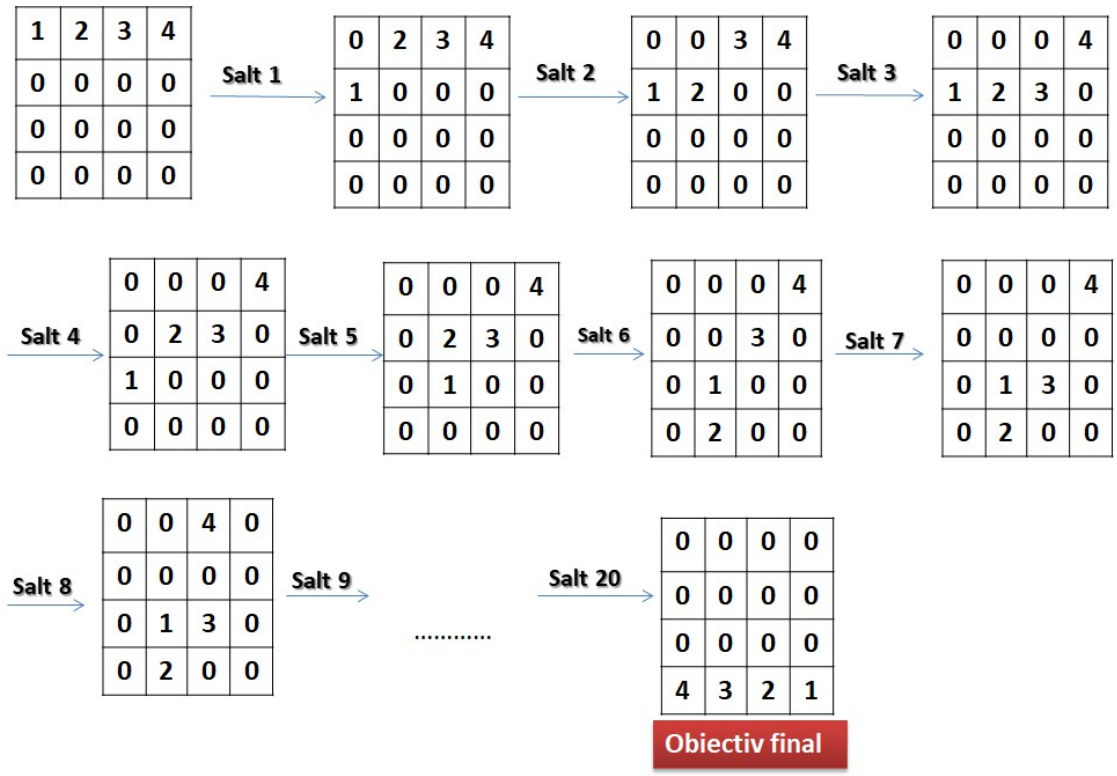


Figura 11: Succesiunea de miscari de la starea initiala la starea finala

Daca ar fi sa presupunem ca avem o parcare de *4* x *4* si *4* vehicule in aceasta parcare, numerotate cu cifrele 1, 2, 3, 4 pentru a ajunge la obiectivul nostru, aceste vehicule ar trebui sa execute mai multe mutari astfel incat la final sa fie asezate in ordine inversa.

Mai sus este prezentata o diagrama realizata in PowerPoint cu scopul de a evidentia numarul de salturi efectuate de cele *4* vehicule pana la obiectivul final. Se poate observa ca pentru o dimensiune de *4* x *4*, cele *4* vehicule executa 20 de salturi/mutari pana ajung la obiectivul final, deoarece fiecare vehicul poate executa cel mult 5 miscari/salturi posibile din orice pozitie.

Starea din care pornesc nu se adauga la numaratoare, deoarece reprezinta starea initiala, starea din care vehiculele incep sa execute salturile, astfel fiind dat starea initiala reprezinta de fapt pozitia fiecarui vehicul, iar starea finala reprezinta pozitiile in care trebuie sa ajunga fiecare dintre vehiculele noastre la randul de sus, dar in ordine inversa. Astfel aceste salturi, respectiv mutari pot fi privite ca actiuni, acestea reprezentand actiunile pe care fiecare vehicul le poate executa in functie de locatia sa in parcare si de pozitia altor vehicule.

7 CONCLUZII FINALE

- Consider ca aceasta tema a reprezentat o provocare(din care am incercat sa invat si sa acumulez cat mai multe detalii, notiuni), atat intelegerea cerintei, abordarea problemei, cat si implemenatarea algoritmului.

- In urma acestei teme, mi-am imbunatatit abilitatile de coding in limbajul Python, cat si notiunile de sintaxa in LaTex.

- O alta experienta a reprezentat-o adaptarea problemei din tema de casa folosind framework-ul de la 15-puzzle problem aferent laboratorului 5. Aceasta a reprezentant cea mai mare provocare, atat modificarea si adaptarea codului, cat si gasirea unei functii euristice admisibile in raport cu cerinta temei de casa, deoarece fiecare vehicul executa maxim 5 actiuni.

- Am ales sa dezvolt implementarea in limbajul Python, deoarece modalitatea de implementare in acest limbaj este mai usor de inteles, si mi-am dorit sa imi imbunatatesc mai mult abilitatile in acest limbaj.

- Consider ca euristica pe care am folosit-o in cadrul acestei rezolvari ofera cel mai admisibil rezultat pentru aceasta problema. Dar cred ca in viitor acest algoritm se poate imbunatati mai mult, astfel incat sa functioneze cat mai optim.

- Ca o extindere a studiului pe termen mai lung, consider ca referinta ajutatoare in intelegerea si parcurgerea temelor de casa, cartea Artificial Intelligence: A Modern Approach, Fourth edition, 2020 by Stuart Russell and Peter Norvig, cat si materialele auxiliare din platformele de laborator, sau din cadrul cursurilor aferente acestei discipline.

- In scopul rezolvarii acestei probleme, cel mai mult m-a ajutat framework-ul de la 15-puzzle problem, cat si materialele auxiliare din platforma de laborator 5 pentru a intelege mai bine cerinta problemei, cat si pentru o abordare mai ampla a rezolvarii.

- O alta concluzie este reprezentata de termenul limita al temei de casa. Acest parametru a avut un impact mai bun asupra organizarii timpului, cat si asupra unei coordonari mai responsabile a celorlalte aspecte realizate pentru a rezolva aceasta tema, ceea ce constituie un beneficiu mai amplu pentru dezvoltarea mea in acest domeniu.

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

8 Bibliografie

Mai jos se pot observa cateva referinte bibliografice, capitole din cursuri, carti, site-uri web, referinte ce au ajutat in parcurgerea, documentarea si intelegerea mai ampla a temei de casa. De pe site-urile respective m-am documentat, link-urile sunt atasate mai jos:

Referinte bibliografice

(1) Artificial Intelligence: A Modern Approach, Fourth edition, 2020 by Stuart Russell and Peter Norvig,
http://aima.cs.berkeley.edu/

(2) Introducing Overleaf and LaTeX,
https://www.overleaf.com/learn/latex/Tutorials
https://oeis.org/wiki/List_of_LaTeX_mathematical_symbols
https://mirrors.nxthost.com/ctan/macros/latex/required/graphics/grfguide.pdf
https://sharelatex.psi.ch/learn/XeLaTeX

(3) Python Documentation contents,
https://www.w3schools.com/python/
http://www.pacosv.ro/informatica/Fisiere_Python.pdf
http://docs.groovy-lang.org/next/html/documentation/working-with-collections.html
https://stackoverflow.com/questions/37400974/unicode-error-unicodeescape-codec-cant-decode-bytes-in-position-2-3-trunca

(4) Capitolul 5 din curs
Capitolul 6 din curs - Cautare euristica

(5) Laborator 5 - Cautare informata folosind Python

(6) Heuristicars Problem
http://www.cs.cmu.edu/afs/cs/academic/class/15780-s15/www/hw/hw1key.pdf
https://ai.stackexchange.com/questions/18355/if-h-i-are-consistent-and-admissible-are-their-sum-maximum-minimum-and-aver

(7) Solving problems by searching,
https://www.pearsonhighered.com/assets/samplechapter/0/1/3/6/0136042597.pdf
https://people.cs.pitt.edu/~milos/courses/cs1571/Lectures/Class3.pdf
https://xlinux.nist.gov/dads/
https://www.geeksforgeeks.org/a-search-algorithm/

(8) PSEUDOCODE DESCRIPTIONS OF THE ALGORITHMS,
https://github.com/aimacode/aima-pseudocode/blob/master/aima3e-algorithms.pdf
https://github.com/ishaansharma/AIMA-Solutions

(9) INTELIGENTA ARTIFICIALA-REZOLVAREA PROBLEMELOR DE CAUTARE,
http://www.cs.ubbcluj.ro/~lauras/test/docs/school/IA/2018-2019/lectures/01_search_uninformed.pdf

(10) VISUALIZATION A GRID,
https://cse442-17f.github.io/A-Star-Search/
https://www.redblobgames.com/grids/hexagons/distances