Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Varvara Raluca Ana-Maria
Tudor Mihai Nicusor
Group: 30232
Email: raluvarvara@yahoo.ro

Teaching Assistant: Anca Iordan
Adrian.Groza@cs.utcluj.ro

Contents

Table 1: Lab scheduling

Activity	**Deadline**
Searching agents, Linux, Latex, Python, Pacman	W_1
Uninformed search	W_2
Informed Search	W_3
Adversarial search	W_4
Propositional logic	W_5
First order logic	W_6
Inference in first order logic	W_7
Knowledge representation in first order logic	W_8
Classical planning	W_9
Contingent, conformant and probabilistic planning	W_{10}
Multi-agent planing	W_{11}
Modelling planning domains	W_{12}
Planning with event calculus	W_{14}

Lab organisation.

1. Laboratory work is 25% from the final grade.
2. There are three deliverables in total: 1. Search, 2. Logic, 3. Planning.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro
4. We use Linux and Latex
5. Plagiarism: Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more.

Chapter 1

A1: Search

Introducere: Acest capitol se refera la problema de cautare cu agenti de inteligenta artificiala in contextul unui puzzle in care am implementat si testat solutiile cele mai usoare si neoptimale (precum DFS - depth first search), dar si solutii optime (precum A*).

Problema de cautare are mai multe aspecte, dar vorbin in general, si de asemenea cel mai important aspect, este gasirea celui mai scurt drum de la o pozitie initiala, la o pozitie finala, pozitie scop.

Cerinta puzzle-ului: : Se considera niste cuburi. Fiecare cub are un identificator, de exemplu o litera.

Cuburile sunt asezate in N stive si putem avea si stive goale. Se de o configuratie initiala a cuburilor asezate in stive si de asemenea o configuratie finala. Se cere Secventa de mutari necesara (configuratiile intermediare) pentru a ajunge de la starea initiala la starea finala. In cadrul unei mutari nu putem lua decat cuburi din varfurile stivelor si putem pune un cub doar in varful altei stive.

Jocul va fi testat pe mai multe nivele: **smallPuzzle** (2 stive si 3 cuburi), **mediumPuzzle** (3 stive si 5 cuburi), **bigPuzzle** (4 stive si 5 cuburi). Pentru fiecare nivel starea initiala va fi generate random si vom avea o stare finala exacta, pe care o vom verifica cu o functie de **isGoal()**.

Stari finale:

SmallPuzzle:
— A —
— B —
— C —
MediumPuzzle:
— A — — B —
— C — — D — — E —
BigPuzzle:
— A — — D — — B —
— C — — E — — F —

Se vor folosi urmatorii algoritmi pentru rezolvarea puzzle-ului:

- DFS (depth first search) - cautare neinformata
- BFS (breadth first search) - cautare neinformata
- UCS (uniform cost search) - cautare neinformata
- A* (folosind euristicile: distanta Euleriana, Manhattan, Cebisev, Chi Squared) - cautare informata

Fiecare algoritm enuntat mai sus poate rezolva problema.

DFS (depth first search)

Aceasta este o **cautare de tip graf/arbore**, care porneste de la o stare particulara, cauta toti succesorii, si alege din unul dintre ei sa fie starea urmatoare. A fost folosita o clasa **Node-search** care retine o stare, un parinte si un cost. Acest algoritm foloseste o **stiva** care sa retina toti succesorii in **frontiera**, stiva fiind structura perfecta deoarece algoritmul vrea sa caute cat mai adanc in fiecare ramura inainte de a face backtrack, asa ca se va lua tot timpul ultimul nod inserat.

```
def depthFirstSearch(problem, no_puzzle):
    global nb_expanded
    nb_expanded = 0

    global nb_generated
    nb_generated = 0

    if problem.isGoalState(problem.getStartState(), no_puzzle):
        # print("Number of expandaded nodes: " + str(nb_expanded))
        return []

    currNode = None
    stack = util.Stack()
    vis = []

    startNode = Node_search(problem.getStartState(), None, 0)
    stack.push(startNode)

    while not stack.isEmpty():

        nb_generated += 1

        currNode = stack.pop()
        if problem.isGoalState(currNode.getState(), no_puzzle):
            break

        if currNode.getState() not in vis:

            vis.append(currNode.getState())
            succ = problem.generateSuccsesors(currNode)

            nb_expanded += 1

            for n in succ:
```

```
                if n.getState() not in vis:
                    stack.push(n)
    sol = []
    while currNode.getParent() != None:
        sol.append(currNode.getState())
        currNode = currNode.getParent()

    sol.reverse()
    return sol
```

Code Listing 1.1: Depth first search

BFS (breadth first search)

Aceasta este o **cautare de tip graf/arbore**, care porneste de la o stare particulara, cauta toti succesorii, ii viziteaza pe toti, si dupa viziteaza toti succesorii acestora si asa mai departe. A fost folosita aceiasi clasa **Node-search**. Acest algoritm foloseste o **coada** care sa retina toti succesorii in **frontiera**, coada fiind structura perfecta deoarece algoritmul vrea sa caute cat mai mult in latime, sa viziteze toti copii prima oara dupa sa ii expandeze si pe ei.

```
def breadthFirstSearch(problem, no_puzzle):
    global nb_expanded
    nb_expanded = 0

    global nb_generated
    nb_generated = 0

    if problem.isGoalState(problem.getStartState(), no_puzzle):
        return []

    currNode = None
    queue = util.Queue()
    vis = []

    startNode = Node_search(problem.getStartState(), None, 0)
    queue.push(startNode)

    while not queue.isEmpty():

        nb_generated += 1

        currNode = queue.pop()

        if problem.isGoalState(currNode.getState(), no_puzzle):
            break

        if currNode.getState() not in vis:

            succ = problem.generateSuccsesors(currNode)
            vis.append(currNode.getState())

            nb_expanded += 1

            for n in succ:
                queue.push(n)
                if problem.isGoalState(n.getState(), no_puzzle):
                    break
```

```
    sol = []
    while currNode.getParent():
        sol.append(currNode.getState())
        currNode = currNode.getParent()
    sol.reverse()
    return sol
```

Code Listing 1.2: Breadth first search

UCS (uniform cost search)

Aceast algoritm este foarte asemanator cu BFS, diferenta fiind ca foloseste o **coada de prioritati** si fiecare intrare in aceasta coada are un **cost atasat**, astfel incat se va extrage de fiecare data intrarea cu costul cel mai mic. Intrarea in coada, de aceasta data, este formata dintr-un Node-search si o lista de noduri care reprezinta lista de noduri prin care s-a trecut pentru a ajunge in acea stare, iar costul tuplei va fi determinat de **lungimea drumului** pana in acel punct. Deci, in acest caz, cel mai scurt drum reprezinta cel mai ieftin. O alta caracteristica de notat a codului este ca in momentul in care se introduce in **frontiera** se foloseste o caracteristica a functiei **update** a structurii **PriorityQueue**, astfel ca, daca nu exista nodul in coada acesta se adauga, dar daca acesta exista, se va updata costul, in cazul in care se gaseste un cost mai mic, astfel compactand codul.

```
def uniformCostSearch(problem, no_puzzle):
    global nb_expanded
    nb_expanded = 0

    global nb_generated
    nb_generated = 0

    queue = util.PriorityQueue()
    vis = []
    sol = []

    if problem.isGoalState(problem.getStartState(), no_puzzle):
        return []

    startNode = Node_search(problem.getStartState(), None, 0)
    queue.push((startNode, sol), startNode.getCost())

    while not queue.isEmpty():

        nb_generated += 1

        t, sol = queue.pop()
        if problem.isGoalState(t.getState(), no_puzzle):
            sol = sol + [t.getState()]
            return sol

        succ = problem.generateSuccsesors(t)

        nb_expanded += 1

        for n in succ:
            if n.getState() not in vis:
                nPath = sol + [n.getParent().getState()]
```

```
                vis.append(n.getState())
                queue.update((n, nPath), len(nPath))  # problem.
    getCostOfActionSequence(nPath)

    return sol
```

Code Listing 1.3: Uniform cost search

Toti algoritmii de mai sus se numesc **neinformati** pentru ca se folosesc doar de informatia provenita din descrierea problemei.

Acum, vom vorbi de **cautare informata**, in care algoritmul se foloseste de informatii din afara problemei (aproximeaza costul pana la solutie), astfel gasind solutii mai bune.

Acesti algoritmi informati folosesc **euristici** pentru a aproxima costul pana la ajungerea la solutie, de exemplu in problema noastra, com aproxima numarul de pasi pe care mai trebuie sa il faca algoritmul pentru a pune un bloc in locatia lui finala.

O **EURISTICA** este orice abordare a rezolvarii problemelor care foloseste o metoda practica care nu este garantata a fi optimă, perfectă sau rațională, dar este totusi suficienta pentru atingerea unui obiectiv sau aproximare pe termen scurt imediat.

O euristica **admisibila** niciodata nu supraestimeaza costul pana la solutie.

O euristica **consistenta** respecta urmatoarea formula pentru fiecare nod n si n0: $h(n) <= c(n; a; n0) + h(n0)$.

Toate euristicile pe care le vom folosi vor fi atat admisibile cat si consistente pentru o solutie cat mai buna.

A* (A star)

Acest algoritm este foarte similar cu UCS numai ca introduce si **euristici** pentru a aproxima costul pana la o solutie, astfel cand introducem in frontiera, sau updatam un nod din frontiera, la cost adaugam si euristica pentru blocul care a fost mutat.

```
def aStarSearch(problem, no_puzzle, heuristic=searchAgent.manhattanHeuristic):
    global nb_expanded
    nb_expanded = 0

    global nb_generated
    nb_generated = 0

    queue = util.PriorityQueue()  # frontiera
    vis = []  # teritoriu
    sol = []

    if problem.isGoalState(problem.getStartState(), no_puzzle):
        # print("Number of expandaded nodes: " + str(nb_expanded))
        return []

    startNode = Node_search(problem.getStartState(), None, 0)
    queue.push((startNode, sol), startNode.getCost())

    while not queue.isEmpty():

        nb_generated += 1

        t, sol = queue.pop()
```

```
        if problem.isGoalState(t.getState(), no_puzzle):
            sol = sol + [t.getState()]
            # print("Number of expandaded nodes: " + str(nb_expanded))
            return sol

        succ = problem.generateSuccsesors(t)

        nb_expanded += 1

        for n in succ:
            if n.getState() not in vis:
                nPath = sol + [n.getParent().getState()]
                vis.append(n.getState())

                pos = BlockPuzzleSearchProblem.newPosition(t, n)
                block = BlockPuzzleSearchProblem.movedBlok(t, n)
                x = 0
                if heuristic == searchAgent.manhattanHeuristic:
                    x = searchAgent.manhattanHeuristic(pos, block, no_puzzle)
                elif heuristic == searchAgent.chebisevDistance:
                    x = searchAgent.chebisevDistance(pos, block, no_puzzle)
                elif heuristic == searchAgent.chiSquaredDistance:
                    x = searchAgent.chiSquaredDistance(pos, block, no_puzzle)
                else:
                    x = searchAgent.euclideanHeuristic(pos, block, no_puzzle)
                queue.update((n, nPath),
                             len(nPath) + x)  # update daca nu exist nodul de
    updatat ii face push asa ca este nevoide doar de update si nu de 2 if - uri
    # print("Number of expandaded nodes: " + str(nb_expanded))
    return sol
```

Code Listing 1.4: A star search

La acest algoritm euristicile se folosesc pozitiile unui bloc, pozitia din starea actuala si pozitia unui bloc din starea finala, astfel am implementat fisierul/libraria SearchAgent pe care o vom folosi pentru calcularea distantelor pentru implementarea euristicilor in a star. Vom avea cate o metoda pentru implementarea fiecarei distanta, dar de asemenea vom avea nevoie de o metoda aditionala care se ne spuna care este starea finala a fiecarui bloc in functie de ce puzzle este.

Euristici explorate

Distanta Manhattan

d(a, b) = $abs(a.x - b.x) + abs(a.y - b.y)$

Distanța Manhattan reprezintă distanța de la un punct la altul, atunci când deplasarea se face strict pe drepte paralele cu axele de coordonate.

Distanta Euclidiană

d(a, b) = $sqrt(a.x - b.x)^2 + (a.y - b.y)^2$

Distanța Euclidiană reprezintă distanța matematică dintre două puncte (lungimea segmentului ce le unește).

Distanta Cebîșev

d(a, b) = $max(abs(a.x - b.x), abs(a.y - b.y))$

Distanța Cebîșev reprezintă un caz special al distanței diagonale, o euristică ce estimează deplasarea minimă pe diagonală de la poziția curentă spre poziția finală. Astfel, distanța diagonală depinde de doi parametri: D, reprezentând costul unei deplasări nediagonale (orizontale sau verticale) și D2, care exprimă costul unei deplasări pe diagonală. În cazul distanței Cebîșev, valorile parametrilor sunt egale între ele și egale cu 1 (D = D2 = 1). Practic, în definirea distanței Cebîșev se pornește de la premisa că atât costul deplasării nediagonale, cât și cel al deplasării pe diagonală sunt egale și au valoarea 1.

Distanta Chi-Squared

d(a, b) = $1/2 * ((a.x - b.x)^2/(a.x + b.x) + (a.y - b.y)^2/(a.y + b.y))$

Masoara asemanarea intre 2 matrici de caracteristici. O astfel de distanță este utilizată în general în multe aplicații, cum ar fi regăsirea imaginilor similare, textura imaginii, extragerile de caracteristici etc. Aceasta distanta functioneaza pe exemplul nostru deoarece 2 state-uri pot fi vazute ca matrice in care in fiecare patratel este nul daca nu exista bloc si numarul blocului daca acesta exista.

```
def blockGoalPosition(block, no_puzzle):
    position = (0, 0)
    if no_puzzle == 1:
        if block == 0:
            position = (0, 1)
        if block == 1:
            position = (0, 0)
        if block == 2:
            position = (1, 0)

    if no_puzzle == 2:
        if block == 0:
            position = (0, 1)
        if block == 1:
            position = (1, 1)
        if block == 2:
            position = (0, 0)
        if block == 3:
            position = (1, 0)
        if block == 4:
            position = (2, 0)

    if no_puzzle == 3:
        if block == 0:
            position = (0, 1)
        if block == 1:
            position = (2, 1)
        if block == 2:
            position = (0, 0)
        if block == 3:
            position = (1, 0)
        if block == 4:
            position = (1, 2)
        if block == 5:
```

```
            position = (3, 0)

    return position

def manhattanHeuristic(position, block, no_puzzle):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = blockGoalPosition(block, no_puzzle)
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, block, no_puzzle):
    "The Euclidean distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = blockGoalPosition(block, no_puzzle)
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5

def chebisevDistance(position, block, no_puzzle):
    xy1 = position
    xy2 = blockGoalPosition(block, no_puzzle)
    d = max((xy1[0] - xy2[0]), (xy1[1] - xy2[1]))
    return d

def chiSquaredDistance(position, block, no_puzzle):
    xy1 = position
    xy2 = blockGoalPosition(block, no_puzzle)
    s0 = xy1[0] + xy2[0]
    s1 = xy1[1] + xy2[1]
    if s0 == 0:
        s0 = 1
    if s1 == 0:
        s1 = 1
    d = 0.5*(((xy1[0] - xy2[0]) ** 2)/(s0)
            + ((xy1[1] - xy2[1]) ** 2)/(s1))
    return d
```

Code Listing 1.5: fisierul/libraria SearchAgent

In urmatoarea parte vom analiza clasele si metodele cu care am implementat rezolvarea problemei.

Clasa BlockPuzzleState

Este clasa care descrie starea in care poate sa fie puzzle-ul nostru. Starea este reprezentata de un vector de stive in care se afla blocurile noastre numerotate, astfel metode de initializare pur si simplu trimite un vector de stive la atributul nostru stacks. Aceasta clasa am proiectat-o doar cu scopul de a codifica starea unui moment din puzzle-ul nostru.

Metode:

- **random-generation(no-stacks, no-blocks)** este o metoda care genereaza un vector de stive cu blocurile de la 0 la no-blocks - 1 aranjate random. Primul pas este sa initializam cate stive este necesar, iar apoi pentru fiecare i de la 0 la no-blocks - 1, generam o stiva random si introducem blocul in acea stiva
- **getStacks** ne returneaza stivele

- **isGoal(self, puzzle)** este metoda care verifica daca starea stivelor coincide cu starea aleasa de noi ca stare finala in functie de numarul puzzle-ului
- **eq** este metoda care suprascrie eq pentru clasa noastra
- **hash** este metoda care suprascrie eq pentru clasa noastra

```
class BlockPuzzleState:

    def __init__(self, stacks):
        self.stacks = stacks

    def random_generation(no_stacks, no_blocks):
        l_stacks = []
        for i in range(no_stacks):
            l_stacks.append(util.Stack())

        for i in range(no_blocks):
            x = int(random.random() * no_stacks)
            l_stacks[x].push(i)
        return l_stacks

    def getStacks(self):
        return self.stacks

    def isGoal(self, puzzle):
        if puzzle == 1:
            if self.stacks[0] == [] and self.stacks[1] == [2, 1, 0] and self.
   stacks[2] == []:
                return True
            else:
                return False
        if puzzle == 2:
            if self.stacks[0] == [2, 0] and self.stacks[1] == [3, 1] and self.
   stacks[2] == [4]:
                return True
            else:
                return False
        if puzzle == 3:
            if self.stacks[0] == [2, 0] and self.stacks[1] == [1, 4] and self.
   stacks[2] == [3, 5]:
                return True
            else:
                return False

    def __eq__(self, other):
        for i in range(len(self.stacks)):
            if self.getStacks()[i] != other.getStacks()[i]:
                return False
        return True

    def __hash__(self):
        return hash(str(self.stacks))
```

Code Listing 1.6: clasa BlockPuzzleState

Clasa Node-search

Este clasa cu care instantiem nodurile in metodele de cautare prezentate mai sus. Aceasta clasa are nevoie de 2 atribute: **o stare** care reprezinta starea descrisa in clasa anterioara si **un parinte** care este un nod de tip Node-search pentru a putea parcurge solutia generata de arborele returnat de metodele de cautare, parintele este nodul din care a fost expandat nodul curent. Aceasta clasa am implementat-o si are rostul de a ajuta pentru implementarea algoritmilor de cautare.

Metode:

- **init** este o metoda care initializeaza clasa, atribuie fiecarui atribut o valoarea, starii ii atribuie in Blockpuzzlestate si parintelui un nod
- **getState** ne returneaza state-ul
- **getParent** ne returneaza parintele
- **getPath** este o metoda ce ne returneaza drumul de la nodul curent la radacina arborelui (adica nodul initial de la care a inceput cautarea)

```
class Node_search:
    def __init__(self, state, parent, cost):
        self.state = state
        self.parent = parent

    def getState(self):
        return self.state

    def getParent(self):
        return self.parent

    def getPath(self):
        path = [self]
        node = self
        while node.getParent() is not None:
            path.insert(0, node.getParent())
            node = node.getParent()
        return path

    def __str__(self):
        to_print = ""
        for stack in self.state.getStacks():
            to_print += (str(stack)) + "\n"
        to_print += "########################\n"
        return to_print
```

Code Listing 1.7: clasa Node-search

Clasa BlockPuzzleSearch

Este clasa cu care implementam efectiv puzzle-ul si functionalitatile de care avem nevoie pentru a implementa metodele de cautare, functionalitati precum generarea succesorilor, metoda

de verificare a starii finale, si metodele pentru determinarea noii pozitii a blocului si blocul care a fost mutat in unul din succesori, metode care ne ajuta pentru implementarea euristicilor din A*.

Metode:

- **init** este o metoda care initializeaza clasa
- **getStartState** ne returneaza state-ul initial
- **isGoalState** expandeaza metoda anterioara de goal state
- **genereazaSuccesori** este o metoda ce ne returneaza toti succesorii unui nod, ia primul block din fiecare stiva si il aseaza in capul fiecarei celeilalte stive
- **newPostion** ne returneaza pozitia noua a blockului mutat, metoda de care avem nevoie in implementarea lui A* pentru euristici, pentru a calcula distantele dintre noul bloc mutat si pozitia goal a acelui nod

```
class BlockPuzzleSearchProblem:

    def __init__(self, puzzle):
        self.puzzle = puzzle

    def getStartState(self):
        return self.puzzle

    def isGoalState(self, state, no_puzzle):
        return state.isGoal(no_puzzle)

    def generateSuccsesors(self, currentNode):
        succsesors = []
        c_stacks = currentNode.getState()
        no_stacks = len(c_stacks.getStacks())
        i = 0
        for idx in range(no_stacks):
            copy_intern = copy.deepcopy(c_stacks.getStacks())
            # print("Sunt la stiva " + str(idx))
            if copy_intern[idx].isEmpty():
                continue
            block = copy_intern[idx].pop()

            for j in range(no_stacks):
                if idx == j:
                    continue
                stacks_n = copy.deepcopy(copy_intern)
                stacks_n[j].push(block)

                nod_nou = Node_search(BlockPuzzleState(stacks_n), currentNode,
    1)
                if nod_nou not in succsesors:
                    succsesors.append(nod_nou)

            copy_intern[idx].push(block)

            # for stiva in copie_interm:
            #     print("Stiva ramasa contine:")
            #     print(str(stiva))
```

```
        return succsesors

    def newPosition(node, otherNode):
        no_stacks = len(node.getState().getStacks())
        s1 = node.getState().getStacks()
        s2 = otherNode.getState().getStacks()
        for idx in range(no_stacks):
            if s1[idx] == s2[idx]:
                continue
            else:
                if len(s1[idx]) < len(s2[idx]):
                    x = idx
                    y = len(s2[idx])
                    position = (x, y)
                    return position

    def movedBlok(node, otherNode):
        no_stacks = len(node.getState().getStacks())
        s1 = node.getState().getStacks()
        s2 = otherNode.getState().getStacks()
        for idx in range(no_stacks):
            if s1[idx] == s2[idx]:
                continue
            else:
                if len(s1[idx]) < len(s2[idx]):
                    x = s2[idx].top()
                    return x
```

Code Listing 1.8: clasa BlockPuzzleSearch

Interfata jocului

Interfata aplicatiei noastre s-a realizat cu ajutorul modului "pygame" a limbajului de programare python.

Fisierul de initializare, cel cu care pornim rularea aplicatiei, "main.py" se bazeaza pe variabila "g" de tipul clasei "Game" (clasa creata de noi de care o sa vorbesc mai jos), care, in consistenta sa, dispune de atributul "running" (de valoarea booleana care ne indica ca aplicatia inca ruleaza), de functia de desenare a imaginii(display-menu) si de procesarea informatiilor (game-loop), astfel, mergand pe o logica abstracta si simpla, putem defini fisierul acesta asa: cat timp jocul ruleaza afisam meniul curent si procesam datele venite din exterior.

"Game"

Fisierul "Game.py" contine clasa "Game" care reprezinta baza aplicatiei noastre. Pasul de pornire in acesta clasa este apelare pygame.init(), care ne initializeaza acest modul. Atributele acestei clase ne ofera tot sprijinul necesar pentru a realiza o aplicatie cat mai functionala si usor de inteles: running - ne indica daca aplicatia inca ruleaza, playing - ne indica daca aplicatia s-a deschis spre utilizare DISPLAY-W - constanta pentru lungimea ferestrei de afisare, DISPLAY-H - constanta pentru latimea ferestrei de afisare, display - reprezinta suprafata de desenare a obiectelor si se initializeaza cu pygame.surface (lungime, latime), window - reprezinta fereastra pe care se aseaza suprafata de desenare si se initializeaza prin pygame.display.set-mode(tupla de (lungime, latime)), font-name1 - constanta care trimite la locatia fontului cu care afisam scrisul meniului, font-name2 - constanta care trimite la locatia fontului cu care afisam scrisul animatiei,

prin pygame.display.set-caption - atribui un nume ferestrei de afisare, constante pentru culoarea negru si alb in format RGB, constante (UP-KEY, DOWN-KEY, START-KEY, BACK-KEY) folosite pentru identificarea apasarii tatelor de catre utilizator, urmatoarele atribute reprezinta instante ale claselor din fisierul "Menu.py" (despre care o sa vorbesc mai tarziu), aceste instante ne ajuta individualizam fiecare pagina de afisare, astfel contine datele caracteristice lor (help-menu, level-menu, alg-menu, animation-menu), iar instanta de curr-menu ne precizeaza mereu la care pagina suntem in momentul respectiv.

Functiile acestei clase sunt create pentru interactiunea utilizatorului cu aplicatia si pentru a desena obiecte simple in cadrul ferestrei.

- Functia "check-events" se foloseste de functie predefinita pygame.event.get() (care returneaza evenimentul ce a avut loc legat de aplicatie), iar prin tipul acestei eveniment decidem urmatoarele: daca evenimentul este de tip quit(s-a apasat pe butonul rosu "x" a ferestrei) atribuim valorilor care ne tin aplicatia deschisa valoarea fals, iar daca tipul evenimentului este de "KEYDOWN" (tasta apasata), indentificam care dintre tastea a fost apasate (K-RETURN - enter, iar la restul numele este sugestiv) si odata cu aceasta indetificare atribuim valoarea de true.

- Functia "reset-keys" atribui valoarea de fals tuturor atributelor pentru taste.

- Functia "game-loop" ne asigura ca atat timp cat aplicatia noasta este "playing" se verifica evenimentele, iar in rest se ocupa de colorarea completa a backgroundului cu negru (display.fill(black)), de colorarea ferestrei cu continului scenei (window.blit(display)), de actualizarea imaginii noi a ferestrei in locul celei vechi (display.update()) si de resetarea valoriilor atributelor pentru taste (pentru ca nu dorim ca programul sa creada ca tastele sunt tinute apasat si deoarece nu tratam evenimentul de ridicare a degetelui de pe tasta, inlocuindul cu apelul lui reset-keys()).

- Functia "draw-text" se ocupa cu "colorarea" ferestrei cu texte (text - continutul ce dorim sa fie afisat, font-name - tipul de font pe care dorim sa il aiba textul afisat, size - dimensiunea textului, x si y - reprezentand locatia unde dorim sa fie acesta afisat), astfel ne folosim de initializarea unei variabile de tip font (pygame.font.Font()) transformam in pixeli textul prin (fonr.render()), ne cream un dreptunghi din suprafata ocupata de text (text-surface.ge-rect()), il centram in interiorul acestui dreptunghi prin atribuirea centrului ca fiind parametrii x si y si dupa il adaugam (il desenam efectiv) pe fereasta prin "display.blit()".

- Functia "draw-block", dupa cum ii spune si numele, ne deseneaza printr-o alta functie implementata de acest modul un dreptunghi pe ecranul precizat, de o anumita culoare,la o pozitie x,y si de o dimensiune w,h.

```
import pygame
from Menu import *

class Game():
    def __init__(self):
        pygame.init()
        self.running = True
        self.playing = False

        self.DISPLAY_W = 480
        self.DISPLAY_H = 270
        self.display = pygame.Surface((self.DISPLAY_W, self.DISPLAY_H))
```

```
        self.window = pygame.display.set_mode(((self.DISPLAY_W, self.DISPLAY_H))
    )
        self.font_name1 = '8-BIT WONDER.TTF'
        self.font_name2 = 'Calibri.TTF'
        pygame.display.set_caption("Blocks Puzzle")
        self.BLACK = (0, 0, 0)
        self.WHITE = (255, 255, 255)

        self.UP_KEY = False
        self.DOWN_KEY = False
        self.START_KEY = False
        self.BACK_KEY = False
        self.PAUSE_KEY = False

        self.help_menu = HelpMenu(self)
        self.level_menu = LevelMenu(self)
        self.alg_menu = AlgorithmsMenu(self)
        self.animation_menu = AnimationMenu(self)
        self.curr_menu = self.level_menu

    def check_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False
                self.playing = False
                self.curr_menu.run_display = False
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RETURN:
                    self.START_KEY = True
                if event.key == pygame.K_BACKSPACE:
                    self.BACK_KEY = True
                if event.key == pygame.K_UP:
                    self.UP_KEY = True
                if event.key == pygame.K_DOWN:
                    self.DOWN_KEY = True
                if event.key == pygame.K_p:
                    self.PAUSE_KEY = True

    def reset_keys(self):
        self.UP_KEY = False
        self.DOWN_KEY = False
        self.START_KEY = False
        self.BACK_KEY = False
        self.PAUSE_KEY = False

    def game_loop(self):
        while self.playing:
            self.check_events()
            if self.START_KEY:
                self.playing = False

        self.display.fill(self.BLACK)
        self.window.blit(self.display, (0,0))
        pygame.display.update()
        self.reset_keys()

    def draw_text(self, text, font_name, size, x, y):
        font = pygame.font.Font(font_name, size)
        text_surface = font.render(text, True, self.WHITE)
        text_rect = text_surface.get_rect() # x,y,w,h
```

```
        text_rect.center = (x,y)
        self.display.blit(text_surface, text_rect)

    def draw_block(self, colour, x ,y, w, h):
        pygame.draw.rect(self.display, colour, [x, y, w, h])
```

Code Listing 1.9: game.py

"Menu"

Fisierul "Menu.py" reprezinta tot ce tine de aspectul aplicatiei, tot ce tine de redirectionare si de legatura dintre algoritmii prezenti in fisierele "search.py" si "searchAgent.py".

In acest fisier plecam de la o clasa parinte "Menu", in care ne-am dat definit caracteristici comune intre toate celelalte pagini ce se afiseaza: ne cream legatura cu fisierul "game.py" prin atributul game (la inceput va fi doar o singura instanta de tipul Game care va preluat de toate paginile - pentru ca doar un singur joc ruleaza), contante pentru mijlocul latimii si lungimii(mid-h, mid-w), un atribut ce ne precizeaza daca fereastra aceasta este cea care este vizibila utilizatorului (run-display), un atribut ce reprezinta cursorul (la origine un dreptunghi de 20x20) cu ajutorul caruia utilizatorul poate sa isi dea seama de alegerile facute si o constanta pentru pozitonarea aspectuala a cursorului (offset).

- Functia "draw-cursor" se foloseste de functia definita in game "draw-text" pentru a transforma acel dreptunghi intr-un caracter "*".
- Funnctia "blit-screen" se foloseste de toate functiile (functii deja prezentate) predefinite in pygame si de atributele instantei "game" necesare pentru actualizare imaginii vechi cu cea noua si de a reseta atributele specifice tastelor.

```
class Menu():
    def __init__(self, game):
        self.game = game
        self.mid_w = self.game.DISPLAY_W / 2
        self.mid_h = self.game.DISPLAY_H /2
        self.run_display = True
        self.cursor_rect = pygame.Rect(0, 0, 20, 20)
        self.offset = -100 # ca sa fie in staga scrisului

    def draw_cursor(self):
        self.game.draw_text('*', self.game.font_name1, 18, self.cursor_rect.x -
    20, self.cursor_rect.y)

    def blit_screen(self):
        self.game.window.blit(self.game.display, (0,0))
        pygame.display.update()
        self.game.reset_keys()
```

Code Listing 1.10: menu.py

"LevelMenu"

Prima clasa care mosteneste clasa "Menu" este clasa "LevelMenu", aceasta fiind prima pagina ce ii este afisata utilizatorului, pe langa atributele mostenite, mai contine si constante pentru pozitia celor 4 optiuni (x si y-ul pentru help, easy, medium, hard), un atribut care ne

specifica la care dintre optiuni ne aflam (state, acesta ne ajuta si la redesenare si redirectare) si tot aici ne pozitionam cursorul in dreptul optiunii de "Help", folosindu-ne de pozitia predefinita a unui dreptunghi in pygame "midtop" si de coordonatele optiunii de help.

- Functia de "display-menu" se ocupa de functionalitatii paginii, astfel ne asiguram ca aceasta pagina este cea care ruleaza (run-display = true) si atunci cat timp ruleaza verificam interactiune utilizatorului (provenita de la obiectul Game), iar odata cu aceasta variabilele specifice tastelor au fost setate corespunzator si ne putem folosi direct de ele in functia personala "check-input" (o sa vorbesc mai jos de ea), refacem backgroundul negru, afisam fiecare optiune folosindu-ne de constantele definite anterior redesanam cursorul in functie de parametrii actuali si, desigur, trebuie sa actualizam ecranul pentru a fi vizibile noile modificari.

- Functia "move-cursor" verifica daca utilizatorul doreste sa se deplaseze printre optiune (sageata sus sau sageata jos) si ne repozitioneaza cursorul in functie de optiunea curenta la urmatoarea optiune, totul cu ajutorul constantelor de pozitie si aitrubutului "state".

- Functia "check-input" se asigura de mutarea cursorului (practic verifica cele 2 taste) si in acelasi timp verifica tasta de enter si prin intermediul atributului de state creeaza un nou "AlgorithmsMenu" cu atributul specific "no-puzzle" (numarul puzzelui necesar pentru rezolvarea problemei) (o sa vorbesc mai jos de el) si il seteaza ca nou menu de afisat (game.curr-menu = alg-menu) pentru optiunile de easy, medium si hard, iar pentru help creeaza un nou "HelpMenu" (o sa vorbesc imediat si de el); pe final setandu-si atributul ce ne specifica ca este pagina vizibila la fals.

```
class LevelMenu(Menu):
    def __init__(self, game):
        Menu.__init__(self, game)
        self.state = "Easy"
        self.helpx = self.mid_w
        self.helpy = self.mid_h - 10
        self.easyx = self.mid_w
        self.easyy = self.mid_h + 30
        self.mediumx = self.mid_w
        self.mediumy = self.mid_h + 50
        self.hardx = self.mid_w
        self.hardy = self.mid_h + 70
        self.cursor_rect.midtop = (self.easyx + self.offset, self.easyy)

    def display_menu(self):
        self.run_display = True
        while self.run_display:
            self.game.check_events()
            self.check_input()
            self.game.display.fill(self.game.BLACK)
            self.game.draw_text('Choose the LEVEL', self.game.font_name1, 20,
    self.game.DISPLAY_W / 2, self.game.DISPLAY_H / 2 - 100)
            self.game.draw_text('Help', self.game.font_name1, 20, self.helpx,
    self.helpy)
            self.game.draw_text('Easy Mode', self.game.font_name1, 20, self.
    easyx, self.easyy)
            self.game.draw_text('Medium Mode', self.game.font_name1, 20, self.
    mediumx, self.mediumy)
            self.game.draw_text('Hard Mode', self.game.font_name1, 20, self.
    hardx, self.hardy)
```

```
            self.draw_cursor()
            self.blit_screen()

    def move_cursor(self):
        if self.game.DOWN_KEY:
            if self.state == 'Easy':
                self.cursor_rect.midtop = (self.mediumx + self.offset, self.
    mediumy)
                self.state = 'Medium'
            elif self.state == 'Medium':
                self.cursor_rect.midtop = (self.hardx + self.offset, self.hardy)
                self.state = 'Hard'
            elif self.state == 'Hard':
                self.cursor_rect.midtop = (self.helpx + self.offset, self.helpy)
                self.state = 'Help'
            elif self.state == 'Help':
                self.cursor_rect.midtop = (self.easyx + self.offset, self.easyy)
                self.state = 'Easy'
        if self.game.UP_KEY:
            if self.state == 'Easy':
                self.cursor_rect.midtop = (self.helpx + self.offset, self.helpy)
                self.state = 'Help'
            elif self.state == 'Medium':
                self.cursor_rect.midtop = (self.easyx + self.offset, self.easyy)
                self.state = 'Easy'
            elif self.state == 'Hard':
                self.cursor_rect.midtop = (self.mediumx + self.offset, self.
    mediumy)
                self.state = 'Medium'
            elif self.state == 'Help':
                self.cursor_rect.midtop = (self.hardx + self.offset, self.hardy)
                self.state = 'Hard'
    def check_input(self):
        self.move_cursor()
        if self.game.START_KEY:
            if self.state == 'Easy':
                self.game.alg_menu = AlgorithmsMenu(self.game, 1)
                self.game.curr_menu = self.game.alg_menu
            elif self.state == 'Medium':
                self.game.alg_menu = AlgorithmsMenu(self.game, 2)
                self.game.curr_menu = self.game.alg_menu
            elif self.state == 'Hard':
                self.game.alg_menu = AlgorithmsMenu(self.game, 3)
                self.game.curr_menu = self.game.alg_menu
            elif self.state == 'Help':
                self.game.help_menu = HelpMenu(self.game)
                self.game.curr_menu = self.game.help_menu
            self.run_display = False
```

Code Listing 1.11: Level Menu

"HelpMenu"

Clasa de "HelpMenu", si ea copil al clasei "Menu", nu dispune de alte atribute specifice, deoarece este folosita doar pentru a oferi un suport de utilizare a aplicatiei jucatorului.

- Functia "display-menu" prezinta aceeasi functionalitate ca si anterior, dar dispune de o redimensionare a ecranului, urmand aceeasi pasi ca la initiliazare acestuia in cadrul clasei

"Game" si doar cu lunigmea si latimea schimbate; astfel, cat timp aceast menu este vizibil se verifica inputul utilizatorului si se afiseaza textului oferit spre ajutor.

- Functia "check-input", de data aceasta verifica doar daca a fost apasata tasta backspace prin intermediul atributului boolean specific si odata cu acesta, se va reveni la meniul anterior ("LevelMenu") prin schimbarea atributului "curr-menu" si se va redimensiona la proportiile initiale ale paginii urmatoare.

```
class HelpMenu(Menu):
    def __init__(self, game):
        Menu.__init__(self, game)

    def display_menu(self):
        self.run_display = True
        self.game.DISPLAY_W = 1800
        self.game.DISPLAY_H = 600
        self.game.display = pygame.Surface((self.game.DISPLAY_W, self.game.
    DISPLAY_H))
        self.game.window = pygame.display.set_mode(((self.game.DISPLAY_W, self.
    game.DISPLAY_H)))
        pygame.display.update()
        while self.run_display:
            self.game.check_events()
            self.check_input()
            self.game.display.fill(self.game.BLACK)
            self.game.draw_text('*You can go through the pages by pressing the
    enter key and backspace key*', self.game.font_name1, 20, self.game.DISPLAY_W
    / 2,
                                self.game.DISPLAY_H / 2 - 80)
            self.game.draw_text('*By pressing the backspace key you are
    redirrected to the previous meniu*',
                                self.game.font_name1, 20, self.game.DISPLAY_W /
    2,
                                self.game.DISPLAY_H / 2 - 55)
            self.game.draw_text('*By pressing  the enter key you are redirrected
     to the next meniu which have been selected by you*',
                                self.game.font_name1, 20, self.game.DISPLAY_W /
    2,
                                self.game.DISPLAY_H / 2 - 30)
            self.game.draw_text('*You can go through the options of a meniu
    using the arrow up and down keys*',
                                self.game.font_name1, 20, self.game.DISPLAY_W /
    2,
                                self.game.DISPLAY_H / 2 - 5)
            self.game.draw_text('*To start the animation press the enter key and
     to close the animation press the backspace key*',
                                self.game.font_name1, 20, self.game.DISPLAY_W /
    2,
                                self.game.DISPLAY_H / 2 + 20)
            self.draw_cursor()
            self.blit_screen()

    def check_input(self):
        if self.game.BACK_KEY:
            self.game.curr_menu = self.game.level_menu
            self.run_display = False

            self.game.DISPLAY_W = 480
```

```
            self.game.DISPLAY_H = 270
            self.game.display = pygame.Surface((self.game.DISPLAY_W, self.game.
    DISPLAY_H))
            self.game.window = pygame.display.set_mode(((self.game.DISPLAY_W,
    self.game.DISPLAY_H)))
```

Code Listing 1.12: HelpMenu

"AlgorithmsMenu"

Clasa de "AlgoritmsMenu", mostenitoare a clasei "Menu", dispune de un atribut specific (no-puzzle, initializat cu 0 pentru a nu interferi cu rezolvarea problemei, care ne indica ce optiune a jocului a fost aleasa 1-easy, 2-medium, 3-hard) si, ca si clasa "LevelMenu" contine constante referitoare la pozitiile optiunilor acestui menu si atributul "state" pentru a indetifica optiunea (optiunile sunt reprezentate de algoritmii cu care dorim sa gasim solutia problemei: bfs, dfs, ucs, A* cu euristicile manhattan, euclidean, chebisev si chiSquared) si tot aici pozitionam cursorul in dreptul primei optiuni - "bfs". Un atribut important din aceasta clasa il considera "START", prin care se face prima legatura cu fisierul de rezolvare a problemei si care reprezinta pozitia initiala de la care se porneste rezolvarea problemei si care, in functie, de atributul "no-puzzle" isi genereaza un numar random de 3,5 sau 6 blocuri pe un numar de 3 stive.

- Functiile "display-menu" si "move-cursor" au aceleasi functionalitati, atat ca sunt schimbate denumirle afisate.
- Functia "check-input", de data aceasta data, are functionalitate atat pentru tasta "backspace" (ne intoarce la "LevelMenu" prin aceeasi modalitate ca la "HelpMenu") si pentru tasta "enter" ne redirectioneaza la urmatoare pagina "AnimationMenu" prin aceeasi modalitate ca si "LevelMenu", oferindu-i atributul de no-puzzle curent si in functie de state, atributul referitor la optiunea aleasa de catre utilizator (necesar pentru a stii care algoritm va rezolva problema) si, desigur, atributul "START" pentru a pastra o buna comparare intre functionalitatea algoritmilor, plecand toti de la aceeasi stare.

```
class AlgorithmsMenu(Menu):
    def __init__(self, game, no_puzzle=0):
        Menu.__init__(self, game)
        self.no_puzzle = no_puzzle
        self.state = "BFS"
        self.bfsx = self.mid_w
        self.bfsy = self.mid_h - 30
        self.dfsx = self.mid_w
        self.dfsy = self.mid_h - 10
        self.ucsx = self.mid_w
        self.ucsy = self.mid_h + 10
        self.manx = self.mid_w
        self.many = self.mid_h + 30
        self.eucx = self.mid_w
        self.eucy = self.mid_h + 50
        self.chex = self.mid_w
        self.chey = self.mid_h + 70
        self.chix = self.mid_w
        self.chiy = self.mid_h + 90
        self.cursor_rect.midtop = (self.bfsx + self.offset, self.bfsy)
```

```
        self.START = BlockPuzzleState(None)
        if self.no_puzzle == 1:
            self.START = BlockPuzzleState(BlockPuzzleState.random_generation(3,
3))
        elif self.no_puzzle == 2:
            self.START = BlockPuzzleState(BlockPuzzleState.random_generation(3,
5))
        elif self.no_puzzle == 3:
            self.START = BlockPuzzleState(BlockPuzzleState.random_generation(3,
6))

    def display_menu(self):
        self.run_display = True
        while self.run_display:
            self.game.check_events()
            self.check_input()
            self.game.display.fill(self.game.BLACK)
            self.game.draw_text('Choose the ALGORITHM', self.game.font_name1,
20, self.game.DISPLAY_W / 2, self.game.DISPLAY_H / 2 - 70)
            self.game.draw_text('BFS', self.game.font_name1, 20, self.bfsx, self
.bfsy)
            self.game.draw_text('DFS', self.game.font_name1, 20, self.dfsx, self
.dfsy)
            self.game.draw_text('UCS', self.game.font_name1, 20, self.ucsx, self
.ucsy)
            self.game.draw_text('Manhattan', self.game.font_name1, 20, self.manx
, self.many)
            self.game.draw_text('Euclidean', self.game.font_name1, 20, self.eucx
, self.eucy)
            self.game.draw_text('Chebisev', self.game.font_name1, 20, self.chex,
 self.chey)
            self.game.draw_text('ChiSquared', self.game.font_name1, 20, self.
chix, self.chiy)

            self.draw_cursor()
            self.blit_screen()

    def move_cursor(self):
        if self.game.DOWN_KEY:
            if self.state == 'BFS':
                self.cursor_rect.midtop = (self.dfsx + self.offset, self.dfsy)
                self.state = 'DFS'
            elif self.state == 'DFS':
                self.cursor_rect.midtop = (self.ucsx + self.offset, self.ucsy)
                self.state = 'UCS'
            elif self.state == 'UCS':
                self.cursor_rect.midtop = (self.manx + self.offset, self.many)
                self.state = 'MAN'
            elif self.state == 'MAN':
                self.cursor_rect.midtop = (self.eucx + self.offset, self.eucy)
                self.state = 'EUC'
            elif self.state == 'EUC':
                self.cursor_rect.midtop = (self.chex + self.offset, self.chey)
                self.state = 'CHE'
            elif self.state == 'CHE':
                self.cursor_rect.midtop = (self.chix + self.offset, self.chiy)
                self.state = 'CHI'
            elif self.state == 'CHI':
                self.cursor_rect.midtop = (self.bfsx + self.offset, self.bfsy)
                self.state = 'BFS'
```

```
        if self.game.UP_KEY:
            if self.state == 'BFS':
                self.cursor_rect.midtop = (self.chix + self.offset, self.chiy)
                self.state = 'CHI'
            elif self.state == 'DFS':
                self.cursor_rect.midtop = (self.bfsx + self.offset, self.bfsy)
                self.state = 'BFS'
            elif self.state == 'UCS':
                self.cursor_rect.midtop = (self.dfsx + self.offset, self.dfsy)
                self.state = 'DFS'
            elif self.state == 'MAN':
                self.cursor_rect.midtop = (self.ucsx + self.offset, self.ucsy)
                self.state = 'UCS'
            elif self.state == 'EUC':
                self.cursor_rect.midtop = (self.manx + self.offset, self.many)
                self.state = 'MAN'
            elif self.state == 'CHE':
                self.cursor_rect.midtop = (self.eucx + self.offset, self.eucy)
                self.state = 'EUC'
            elif self.state == 'CHI':
                self.cursor_rect.midtop = (self.chex + self.offset, self.chey)
                self.state = 'CHE'

    def check_input(self):
        self.move_cursor()
        if self.game.BACK_KEY:
            self.game.curr_menu = self.game.level_menu
            self.run_display = False
        if self.game.START_KEY:
            if self.state == 'BFS':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"bfs", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            elif self.state == 'DFS':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"dfs", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            elif self.state == 'UCS':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"ucs", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            elif self.state == 'MAN':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"man", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            elif self.state == 'EUC':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"euc", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            elif self.state == 'CHE':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"che", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            elif self.state == 'CHI':
                self.game.animation_menu = AnimationMenu(self.game, self.game.
alg_menu.no_puzzle,"chi", self.game.alg_menu.START)
                self.game.curr_menu = self.game.animation_menu
            self.run_display = False
```

Code Listing 1.13: AlgorithmsMenu

"AnimationMenu"

Clasa "AnimationMenu", ultimul copil al clasei si ultima clasa din acest proiect, reprezinta vizualizare efectiva a rezolvarii problemei prin optiunile dorite/specificate anterior de catre utilizator. Aceast dispune de o clasa interna de tip Enum care contine tuple RGB ca valori si "COLOURNR" (unde nr=0-14) ca si chei. Totodata, am implementat in cadrul acesteia functia "give-rgb", care ne ofera o comoditate in extragerea valorii in functie de pozitia sa (pozitie corespunzatoare cu codificare blocului) - cauta prin taote declarile clasei si ne ofera valoarea declararii de pe pozitia "poz" trimisa ca si parametru a functiei.

Pe langa atributele primite prin initializarea in "AlgorithmsMenu", aceasta clasa mai dispune si de un atribut care ne precizeaza daca animatia ruleaza (animation-running), un atribut care ne specifica daca animatie este completa (animation-done), unul care indica daca animatia a fost pusa pe pauza (animation-paused), un atribut pentru a tine cont la care pas al animatiei suntem (iterate-animation) si atributele referitoare la solutionare problemei: numarul de noduri expandate si generate prin rezolvarea acesteia (n-expanded, n-generated), un atribut pentru a initializa un obiect de tip "BlockPuzzleSearchProblem" (problem) si un atribut in care vom primi solutia sub forma unei lista de noduri de cautare (SOL). Tot aici, dorim sa primim soluita la problema noastra si datele anexe ei, astfel, pe baza atributului de "type", vom apela functia specifica algoritmului ales (breadthFirstSearch(), depthFirstSearch(), uniformCostSearch(), aStarSearch(-,-,1 din cele 4 euristici)). Urmatoarele atribute reprezinta constante pentru dimensiunea unui bloc din problema noastra (BLOCK-H, BLOCK-W), pozitiei scorului si valoarea acestuia, a pozitiei legendei pe ecran (scorex, scorey, score-value, legendax, legenday), atribute sub forma de liste care ne indica pozitiile fixate la care dorim sa se fac desenarea blocurilor (stivex, position - inaltimele disponibile ale unei stive) si ultimele 3 atribute care reprezinta pozitia si culoarea oricarui bloc in unul di oricare moment al animatiei (blockx, blocky, blockc).

- Functia "display-menu" are aceeasi functionalitate: redimensioneaza ecranul la fel ca si la "HelpMenu", atribuie valorile constantelor referitoare la pozitii, iar pe langa functionalitatile comune din bulca de rulare, se mai ocupa de afisarea scorului, a legendei (show-score-and-values, show-legend - o sa prezint imediat si aceste functii) si de logica referitoare la animatie; astfel, daca animatia nu ruleaza sau nu este gata pregatim pozitia de start (prepare-the-scene), atlfel daca animatia ruleaza, dorim sa afisam starea curenta la care s-a ajuns cu ajutorul atributului de iterare (start-animation), si, totodata, daca animatia nu este pauzata, dorim sa penalizam la scor pentru fiecare mutare cu 10 puncte si sa mergem la urmatorul pas al acesteia; daca cumva animatia a luat sfarsit, vom dori sa fie vizibil faptul ca s-a terminat cu succes (final-scene).

- Functia "check-input" trateaza la fel cazul pentru tasta "backspace" (se asigura si ca animatie va lua sfarsit prin animation-running = False), dar prin apasarea tastei "enter" pornim rularea animatiei prin atribuirea valorii de adevar atributului specific acestei functionalitati (animation-running).

- Functia "show-score-and-values" ne deseneaza pe ecran valorile referitoare la valoarea scorului, numarul de noduri generate si expandate de catre algoritm in scopul gasirii solutiei.

- Functia "show-legend" ne deseneaza codificarea folosita in rezolvarea problemei (nr bloc - culoare - reprezentata de un bloc 15x15 de culoarea repsectiva), iar in functie de valoarea atributului "no-puzzle" desneaza blocurile respective fiecareia.

- Functia "prepare-the-scene", dupa cum ii spune si numele, se foloseste de atributul "START" pentru a redesena pozitiile initiale a fiecarui bloc - parcurgem fiecare stiva

si luam fiecare bloc pe rand si intermediul "game.draw-block()" si a "Colour.give-rgb()", a "stivex", a "position" si constantelor pentru dimensiunea unui bloc plasam blocul respectiv pe ecran. Odata ce blocul a fost plasat actualizam listele de informatii personale a fiecarui bloc prin atribuirea valorilor folosite la desenare.

- Functia "final-scene" functioneaza pe acelasi principiu ca si "prepare-the-scene", doar atat ca aceasta functia ne pune la dispozitie ultima stare in care ne adus algoritmul, astfel putem vizualiza cu usurinta daca algoritmul a avut succes sau nu; fiind si ultima stare nu mai este necesar sa schimbam valorile din liste blocurilor.

- Functia "start-animation" prezinta functionalitate de desenare a starii curente (acelasi principiu cu "prepare-the-scene", dar va desena starea precizata de atributul de iterare din interiorul solutiei atata timp cat aceasta nu este egala cu ultima; deja la ultima stare oprim rularea animatiei (animation-running va primi valoare de fals, iar animation-done valoare de true) si vom oferi la scorul algoritmului un punctaj aferent greutatii puzzlelui rezolvat (easy - +300, medium - +500, hard - +1000). Pentru ca utilizatorul sa poata urmari cu usurinta redesenarea starilor, incetinim timpul rularii acestei functii prin punerea acesteia in starea de sleep pentru 0,5 secunde (time.sleep(0.5)).

```
class AnimationMenu(Menu):
    class Colour(Enum):
        COLOUR0 = (6, 77, 135)  # nuanta de albastru
        COLOUR1 = (0, 83, 98)  # nunata de albastru
        COLOUR2 = (255, 130, 67)  # mango tango
        ##########################################
        COLOUR3 = (226, 114, 91)  # terra cotta
        COLOUR4 = (221, 160, 216)  # nuanta de roz
        COLOUR5 = (147, 112, 219)  # nuanta de mov
        ##########################################
        COLOUR6 = (21, 119, 40)  # nuanta de verde
        COLOUR7 = (135, 135, 0)  # nuanta de galben
        COLOUR8 = (123, 156, 169)  # nuanta de albastru
        COLOUR9 = (196, 0, 0)  # nuanta de rosu
        COLOUR10 = (0, 150, 141)  # nuanta de verde
        COLOUR11 = (114, 73, 32)  # nuanta de maro
        COLOUR12 = (246, 29, 98)  # nuanta de roz
        COLOUR13 = (131, 38, 26)  # nuanta de maro
        COLOUR14 = (42, 42, 42)
        COLOUR15 = (69, 69, 69)

        def give_rgb(poz):
            i = 0
            for data in AnimationMenu.Colour:
                if i == poz:
                    return data.value
                i += 1

    def __init__(self, game, no_puzzle=0, type="", start=None):
        Menu.__init__(self, game)
        self.no_puzzle = no_puzzle
        self.type = type
        self.START = start
        self.animation_running = False
        self.animation_done = False
        self.animation_paused = False
        self.iterate_animation = 0
```

```
        self.n_expanded = 0
        self.n_generated = 0

        self.problem = BlockPuzzleSearchProblem(self.START)
        self.SOL = []
        if self.type == "bfs":
            self.SOL = breadthFirstSearch(self.problem, self.no_puzzle)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        elif self.type == "dfs":
            self.SOL = depthFirstSearch(self.problem, self.no_puzzle)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        elif self.type == "ucs":
            self.SOL = uniformCostSearch(self.problem, self.no_puzzle)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        elif self.type == "man":
            self.SOL = aStarSearch(self.problem, self.no_puzzle, searchAgent.
manhattanHeuristic)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        elif self.type == "euc":
            self.SOL = aStarSearch(self.problem, self.no_puzzle, searchAgent.
euclideanHeuristic)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        elif self.type == "che":
            self.SOL = aStarSearch(self.problem, self.no_puzzle, searchAgent.
chebisevDistance)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        elif self.type == "chi":
            self.SOL = aStarSearch(self.problem, self.no_puzzle, searchAgent.
chiSquaredDistance)
            self.n_expanded = search.nb_expanded
            self.n_generated = search.nb_generated
        # print(len(self.SOL))

        self.BLOCK_H = 100
        self.BLOCK_W = 300

        self.score_value = 0
        self.scorex = self.mid_w
        self.scorey = self.mid_h

        self.legendax = self.mid_w
        self.legenday = self.mid_h

        self.stivex = [self.mid_w] * 3
        self.position = [self.mid_h] * 6
        self.blockx = [0] * 6
        self.blocky = [0] * 6
        self.blockc = [(255, 255, 255)] * 6

    def display_menu(self):
        self.run_display = True

        if self.type :
            self.game.DISPLAY_W = 1400
```

```
            self.game.DISPLAY_H = 800
            self.game.display = pygame.Surface((self.game.DISPLAY_W, self.game.
DISPLAY_H))
            self.game.window = pygame.display.set_mode(((self.game.DISPLAY_W,
self.game.DISPLAY_H)))
            pygame.display.update()

            self.scorex = 60
            self.scorey = 20

            self.legendx = 250
            self.legendy = 20

            self.position[0], self.position[1], self.position[2], self.position
[3], self.position[4], self.position[5] = 675, 575, 475, 375, 275, 175
            self.stivex[0], self.stivex[1], self.stivex[2] = 200, 550, 900


        while self.run_display:
            self.game.check_events()
            self.check_input()
            self.game.display.fill(self.game.BLACK)
            self.show_score_and_values()
            self.show_legend()

            if not self.animation_running and not self.animation_done:
                self.prepare_the_scene()
            elif self.animation_running:
                self.start_animation(self.iterate_animation, len(self.SOL))
                if not self.animation_paused:
                    self.iterate_animation += 1
                    self.score_value -= 10
            elif self.animation_done:
                self.final_scene(len(self.SOL) - 1)

            self.blit_screen()


    def check_input(self):
        if self.game.BACK_KEY:
            self.game.curr_menu = self.game.alg_menu
            self.run_display = False

            self.game.DISPLAY_W = 480
            self.game.DISPLAY_H = 270
            self.game.display = pygame.Surface((self.game.DISPLAY_W, self.game.
DISPLAY_H))
            self.game.window = pygame.display.set_mode(((self.game.DISPLAY_W,
self.game.DISPLAY_H)))

        if self.game.START_KEY:
            self.animation_running = True

        # if self.game.PAUSE_KEY:
        #     self.animation_paused = not self.animation_paused

    def show_score_and_values(self):
        self.game.draw_text("Score: " + str(self.score_value), self.game.
font_name2, 20, self.scorex, self.scorey)
        self.game.draw_text("Number of generated nodes: " + str(self.n_generated
```

```
), self.game.font_name2, 20, self.scorex + 105, self.scorey + 40)
        self.game.draw_text("Number of expandaded nodes: " + str(self.n_expanded
), self.game.font_name2, 20, self.scorex + 105, self.scorey + 60)

    def show_legend(self):
        self.game.draw_text('Legend: ', self.game.font_name2, 20, self.legendx,
self.legendy)
        self.game.draw_text('Block0: ', self.game.font_name2, 20, self.legendx +
 100, self.legendy)
        self.game.draw_text('Block1: ', self.game.font_name2, 20, self.legendx +
 250, self.legendy)
        self.game.draw_text('Block2: ', self.game.font_name2, 20, self.legendx +
 400, self.legendy)

        self.game.draw_block(self.Colour.give_rgb(0), self.legendx + 100 + 50,
self.legendy - 10, 15, 15)
        self.game.draw_block(self.Colour.give_rgb(1), self.legendx + 250 + 50,
self.legendy - 10, 15, 15)
        self.game.draw_block(self.Colour.give_rgb(2), self.legendx + 400 + 50,
self.legendy - 10, 15, 15)

        if self.no_puzzle == 2 or self.no_puzzle == 3:
            self.game.draw_text('Block3: ', self.game.font_name2, 20, self.
legendx + 550, self.legendy)
            self.game.draw_text('Block4: ', self.game.font_name2, 20, self.
legendx + 700, self.legendy)
            self.game.draw_block(self.Colour.give_rgb(3), self.legendx + 550 +
50, self.legendy - 10, 15, 15)
            self.game.draw_block(self.Colour.give_rgb(4), self.legendx + 700 +
50, self.legendy - 10, 15, 15)

        if self.no_puzzle == 3:
            self.game.draw_text('Block5: ', self.game.font_name2, 20, self.
legendx + 400, self.legendy + 20)
            self.game.draw_block(self.Colour.give_rgb(5), self.legendx + 400 +
50, self.legendy + 10, 15, 15)

    def prepare_the_scene(self):
        the_stacks = self.START.getStacks()
        l1 = len(the_stacks)
        for i in range(l1):
            the_stack = the_stacks[i].list
            l2 = len(the_stack)
            for j in range(l2):
                self.game.draw_block(self.Colour.give_rgb(the_stack[j]), self.
stivex[i], self.position[j], self.BLOCK_W, self.BLOCK_H)
                self.blockx[the_stack[j]] = self.stivex[i]
                self.blocky[the_stack[j]] = self.position[j]
                self.blockc[the_stack[j]] = self.Colour.give_rgb(the_stack[j])

    def final_scene(self, max):
        the_stacks = self.SOL[max].getStacks()
        l1 = len(the_stacks)
        for j in range(l1):
            the_stack = the_stacks[j].list
            l2 = len(the_stack)
            for l in range(l2):
                self.game.draw_block(self.blockc[the_stack[l]], self.blockx[
the_stack[l]], self.blocky[the_stack[l]], self.BLOCK_W, self.BLOCK_H)
```

```
    def start_animation(self, i, max):
        time.sleep(0.5)
        if i == max:
            self.animation_running = False
            self.animation_done = True
            if self.no_puzzle == 1:
                self.score_value += 300
            if self.no_puzzle == 2:
                self.score_value += 500
            if self.no_puzzle == 3:
                self.score_value += 1000
        else:
            the_stacks = self.SOL[i].getStacks()
            l1 = len(the_stacks)
            for j in range(l1):
                the_stack = the_stacks[j].list
                l2 = len(the_stack)
                for l in range(l2):
                    self.game.draw_block(self.blockc[the_stack[l]], self.blockx[
    the_stack[l]], self.blocky[the_stack[l]], self.BLOCK_W, self.BLOCK_H)
                    self.blockx[the_stack[l]] = self.stivex[j]
                    self.blocky[the_stack[l]] = self.position[l]
                    self.blockc[the_stack[l]] = self.Colour.give_rgb(the_stack[l
    ])
```

Code Listing 1.14: AnimationMenu

Aceasta prezentare a fost facuta bazandu-se pe cunoasterea notiunilor de baza a limbajului python de catre cititor.

Compararea performanțelor. Concluzii

Pentru a putea realiza o comparație cât mai relevantă între algoritmii studiați, și pentru a concluziona care dintre aceștia este cel mai performant pentru problema de căutare pe Block Game s-a analizat performanța tuturor algoritmilor pentru toate nivelele puzzle-ului .

Pentru algoritmul A* Search s-a comparat pentru fiecare tip de euristica, pentru o analiza mai in detaliu a actiunii euristicilor pentru acest algoritm.

Rezultatele testelor sunt înregistrate în tabelul de mai jos:

Small puzzle

Algoritm	Noduri Expandate	Noduri generate	Timp de execuție (sec)
Depth-First	13	14	0.005
Breadth-First	31	63	0.008
Uniform-Cost	59	60	0.045
A* euclidean	221	26	0.015
A* manhattan	52	53	0.042
A* chi squared	25	35	0.030
A* chebisev	60	61	0.046

Medium puzzle

Algoritm	Noduri Expandate	Noduri generate	Timp de execuție (sec)
Depth-First	1132	1134	12.80
Breadth-First	2105	6733	20.93
Uniform-Cost	611	612	3.90
A* euclidean	511	512	4.30
A* manhattan	407	408	2.90
A* chi squared	1401	1402	19.70
A* chebisev	756	757	8.14

Big puzzle

Algoritm	Noduri Expandate	Noduri generate	Timp de execuție (sec)
Depth-First	9349	9353	765.6
Breadth-First	13492	41941	731.37
Uniform-Cost	7154	7155	489.48
A* euclidean	2779	2781	120.2
A* manhattan	6180	6188	463.1
A* chi squared	5010	5012	306.59
A* chebisev	4287	4289	229.9

Dupa cum se vede din analiza de mai sus cel mai eficient algoritm din toate punctele de vedere (noduri generate, expandate si timp de rulare) este A* cu euristica mahattan.

Se poate observa ca cu cat adaugam ceva se comporta mai bine, adica BFS si DFS sunt comparabile, in functie de nivelul ales, se comporta asemanator, ambii fiind algoritmi neinformati, apoi UCS se comporta comparativ mai bien doarece am introdus ideea de cost pana la nodul curent expandat, iar apoi A* se comporta cel mai bine, desi la el un element definitoariu este euristica. Cum la acest joc se poate vorbi de distante se vede ca Manhattan se comporta cel mai bine, iar euristica CHI nu este foarte buna, deoarece ea nu se foloseste atat de mult pe probleme de distante si cautare de acest gen.

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

Appendix A

Your original code

Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more. This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained.