Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Tivadar Maria Simona
Group: 30233
Email: simona-maria05@yahoo.com

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

Contents

Chapter 1

A1: Introducere

In cadrul acestei teme la Inteligenta Artificiala am incecat sa ating cateva din subiectele propuse spre analiza la laborator.Printre acestea se numara "Problema Colturilor", o euristica pentru aceasta problema.Totodata, am scris o euristica simpla si am incercat dezvoltarea unui alt algoritm de cautare - o varianta la A* predat la curs.

Algoritmi de cautare

Initial, am scris functionalitatea algoritmilor de baza de cautare, codul aferent lor se gaseste in ultiam sectiune a documentatiei. Printre algoritmii implementati se numara

1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. Uniform Cost Search(UCS)
4. A* Search(aStarSearch)
5. Weighted A* Search

La nivelul algoritmilor pentru BFS si DFS am incercat sa adaug o nota diferita si am construit calea de la pozitia de start pana la pozitia de final cu ajutorul unei vector de tati. Acest algoritm ne-a fost prezentat anterior la materia de Algoritmi Fundamentali ca fiind o varianta potrivita in vedere implementarii optime a acestor algoritmi. Chiar daca ei sunt, in esenta, algoritmi de baza, acestia i vom folosi mai departe pentru rezolvarea unor alte probleme.

Despre UCS cunoastem ca este un algoritm complet si optimal, spre deosebire ce celelalte doua mentionate anterior. Cu toate acestea, UCS exploreaza optiunile foarte largit, dureaza pana se apropie de solutii, folosindu-se un numar mare de resurse in largirea cautarii, cautarea care nu ajunge la pozitia finala.

Algoritmul A* intra in categoria algoritmilor Informed Search si reprezinta o combinatie a algoritmilor Greedy si UCS. Acesta priveste in avans si exploreaza solutii doar in zona in care se presupune ca ar putea ajunge mai rapid la pozitia finala. Acesta combina importanta euristicii cu importanta costului.

Algoritmul Weighted A* prezinta o varianta imbunatatita a algoritmului A* care pune anumite ponderi, pe functia de cost si pe cea de euristica, modificand astfel spatiul de cautare in favoarea unei optimalitati cat mai ridicate. Anumite valori pentru w (pondere) prezinta diferite cazuri particulare car vor fi detaliate in seciunea 2.

Problema colturilor

Pentru aceastra problema am implementat pe rand functiile necesare aplicarii problemei, care vor fi detlaiate in secitunea 2. Mai mult decat atat, aceasta problema a fost rulata si testata folosind algoritmu de cautare BFS. Cele e functii mentionate sunt getStartingPosition, isGoalState si getSuccessors

Problema colturilor

Pentru aceastra problema am implementat pe rand functiile necesare aplicarii problemei, care vor fi detlaiate in secitunea 2. Mai mult decat atat, aceasta problema a fost rulata si testata folosind algoritmu de cautare BFS. Cele 3 functii mentionate sunt:

1. getStartState(state)
2. isGoalState(state)
3. getSuccessors(state)

Euristici

De departe cea mai importanta euristica scrisa de mine este cea care are ca scop rezolvarea problemei colturilor. Aceasta va fi detaliata asa cum se cuvinte in sectiunea 2. Testare ei a fost facuta cu ajutorul comenzii de autograder.
Scopul acestei euristici este de a obtine o functie, o aproximare, cat mai aproape de costul real, insa sub costul real. Am avut grija ca presupunerea sa nu fie prea pesimista pentru ca acest lucru ar fi pecetluit atingerea scopului nostru. Vrem mereu o euristica cat mai optimista, dar totusi a carei valoare sa fie mai mica decat costul real.

Pe langa aceasta euristica, am mai scris una de baza, menita sa ma ajute in testarea algoritmului de cautare Weighted A* si anume diagonalHeuristic

Chapter 2

A2: Implementare

BFS

- e un algoritm complet, are complexitate buna in timp insa lasa de dorit in ceea ce priveste complexitatea in spatiu. Per total, nu e un algoritm de mare optimalitate BFS se implementeaza cu ajutorul unei structuri de tip Queue()

DFS

- e un algoritm complet. Complexitatea in spatiu este mai buna decat la BFS insa cea in timp lasa de dorit. Totodata, nu e un algoritm optimal. DFS se implementeaza cu ajutorul unei structuri de tip Stack()

UCS

- are ca principal scop explorarea tuturor solutiilor adiacente si in cele din urma alegerea aceleia ce are un cost minim. Implementarea acestuia se face cu ajutorul unei structuri de tip PriorityQueue().

A*

Algoritmul A* functioneaza dupa urmatoarea functie : f(n) = g(n) + h(n)

n - reprezinta nodul urmator
functia g - costul de a ajunge la nodul n
functia h - euristica necesara pt a ajunge la nodul n
A* se implementeaza cu ajutorul unei structuri de tip PriorityQueue().

Weighted A*

Algoritmul Weighted A* functioneaza dupa urmatoarea functie :
f(n) = g(n) + w*h(n)

n - reprezinta nodul urmator
functia g - costul de a ajunge la nodul n
functia h - euristica necesara pt a ajunge la nodul n
Weighted A* se implementeaza cu ajutorul unei structuri de tip PriorityQueue().

Ceea ce este foarte interesant la Weighted A* este ca sptiaul de cautare al starilor se dmiscoareaza in functie de ponderea pe care i-o dam functieti. Astfel incat, acest lucru se observa foarte bine la sectiuena Testare. Spatiul de cautare este mai mic la Weighted A*. Asadar, prin intermediul ponderii se poate modifica importanta euristicii - daca este mai putin optimala sa spunem sau invers.] Functia se mai poate scrie si astfel
f(n) = (1-w)*g(n) + w*h(n)

Ceea ce este foarte interesant la Weighted A* este ca sptiaul de cautare al starilor se dmiscoareaza in functie de ponderea pe care i-o dam functieti. Astfel incat, acest lucru se observa foarte bine la sectiuena Testare. Spatiul de cautare este mai mic la Weighted A*. Asadar, prin intermediul ponderii se poate modifica importanta euristicii - daca este mai putin optimala sa spunem sau invers.

Cazuri particulare:

1. w = 1 avem Greedy Breadth First Search
2. w = 0 avem algoritmul lui Dijkstra

Problema colturilor

1. getStartState(state)
2. isGoalState(state)
3. getSuccessors(state)

1. Este o problema foarte interesanta deoarece, spre deosebire de de problema anterioara, starea initiala este una mult mai complexa. Retinem nu doar pozitia, cat si starea, mai exact, numarul de colturi care au fost deja vizitate.
Pentur a face acest lucru m-am folosit de un obiect de tipul Boolean care retine true sau false in functie de starea coltului - daca a fost vizitat sau nu. Mai mult decat atat, aceasta functie returenaza atat pozitia curenta, cat si starea de vizitare a celor 4 colturi.

2. Am ajuns la goalState doar daca au fost vizitate cele 4 colturi, asta inseamna ca in obiectul de Boolene trebuie sa avem 4 valori setate pe True, cate una pentru fiecare colt - aceasta este conditia care imi spune daca telul a fost atins sau inca nu.

3. Functia de generare a succesorilor este si ea destul de interesanta. Initial verificam daca nu cumva vom lovi vreun zid - aceasta functionalitate exista si in problema anterioara. Cu toate acestea, de aceasta data vom analiza daca starea viitoare in care dorim sa mergem apartine sau nu listei de colturi si in functie de acest lucru populam obiectul nostru de tip boolean, pe care il adaugam starii nodului caruia i dam append in lista de succesori

Euristici

1. Euristica pentru CornersProblem
2. Euristica pe diagonala

1. Euristica mea referitoare la problema colturilor se bazeaza pe cautarea unei solutii care sa fie mai optimala decat euristica Manhattan.
Asadar, trecem pe rand prin fiecare din cele 4 colturi si calculam pentru pozitia in care ma aflu acum cat ar fi distanta Manhattan. In urma compararii, daca obtin o valoare optima, aceasta va fi luata in considerare. Aceaasta euristica se aplica pe rand tuturor colturilor in ordinea in care sunt vizitate

2. Euristica pe diagonala se aseamana euristicii Manhattan sau Euclidiene prin simplitatea ei. Aceasta am definit-o in vederea aplicarii algoritmului Weighted A* si pentur a putea compara intre ele spatiile de cautare pentru mai multi algoritmi de cautare.

Formula pe care se bazeaza aceasta euristica este:
max($abs(x_1 - goalX_2), abs(y_1 - goalY_2)$ **)**

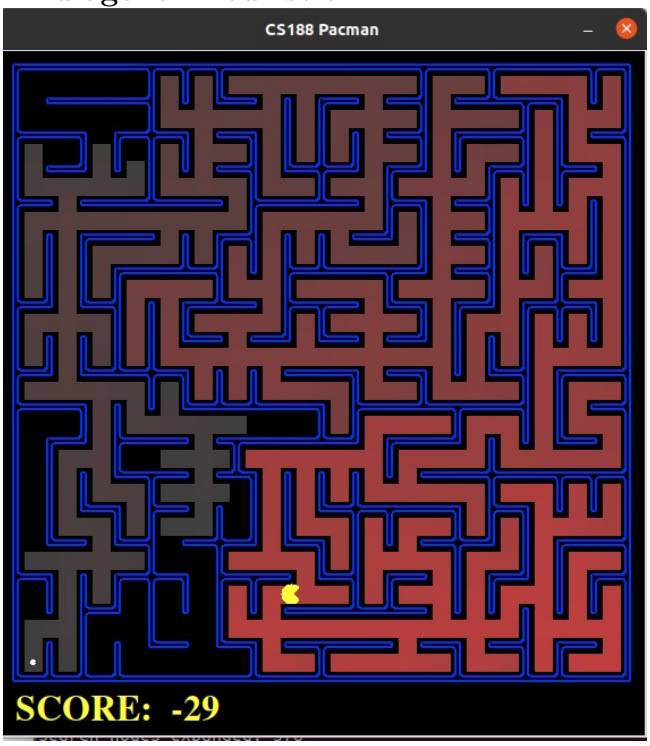
Chapter 3

A3: Testare

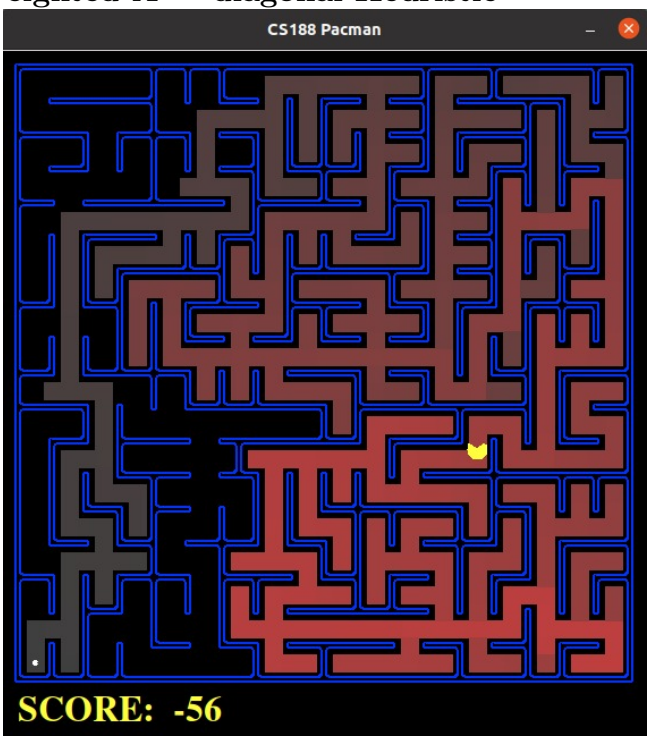
Mai jos voi atasa pe rand, pentru fiecare comanda, starea gridului si eventuale afisari in terminal.] Mentionez ca am modificat doar fisierele search.py si searchAgents.py.
Pentru testare am folosit in principal comenzile mentionate pe site-ul de la Berkeley, dar si cele prezente in fisierul commands.

Mai jos voi atasa pe rand, pentru fiecare comanda, starea gridului si eventuale afisari in terminal.

A* - diagonal Heuristic



Weighted A* - diagonal Heuristic



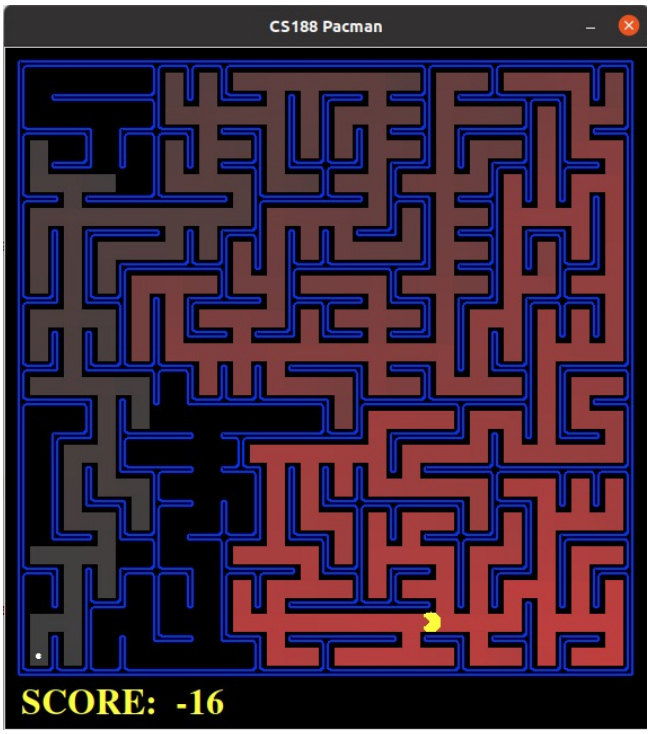
Find all corners

```
simona@ubuntu:~/Downloads/search$ python pacman.py -l tinyCorners -p SearchAgent
 -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
simona@ubuntu:~/Downloads/search$
```

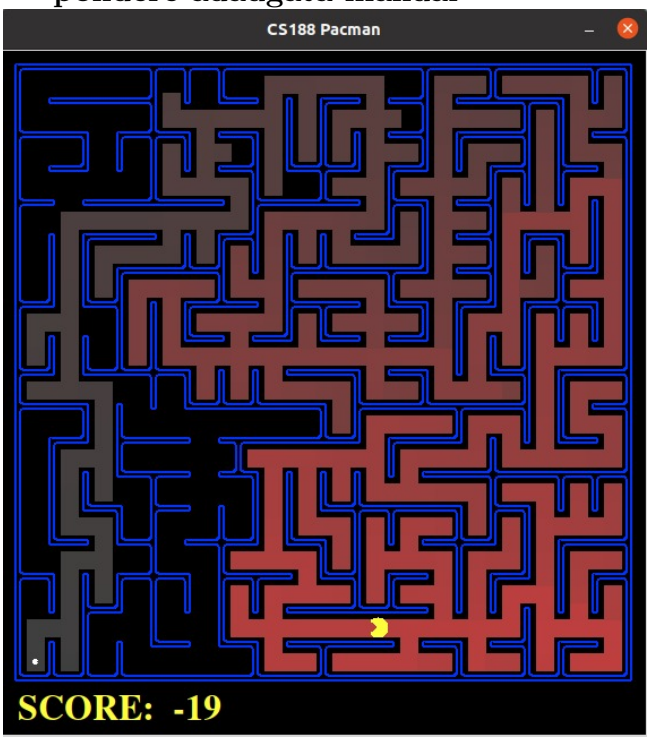
Corners Heuristic


```
simona@ubuntu:~/Downloads/search$ python pacman.py -l mediumCorners -p AStarCorn
ersAgent -z 0.5
((5, 1), (False, False, False, False)) Path found with total cost of 106 in 0.1
seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
simona@ubuntu:~/Downloads/search$
```

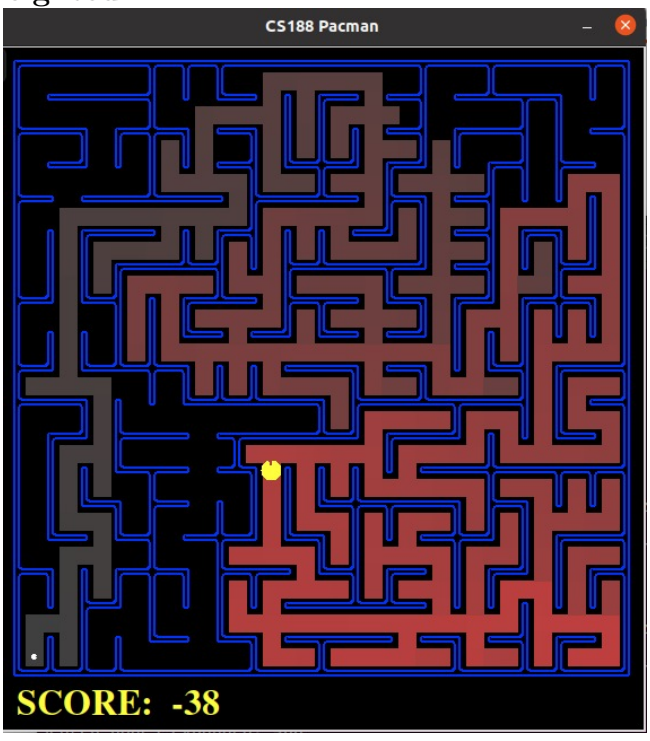
A* - normal



A* - pondere adaugata manual



Weighted A*



Autograder

```
Finished at 7:02:54

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 18/25
```

Appendix A

Your original code

Listing A.1: Basic Search Algorithms

```
def depthFirstSearch(problem):
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    n = Directions.NORTH
    e = Directions.EAST
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:
    """
    from util import Stack

    stiva = Stack();
    visited = [];
    path = []
    father = {} #dictionary that will help me make the output list of directi
    stiva.push(problem.getStartState());
    current = problem.getStartState();
    father[current] = 'Nimic';
    while (not stiva.isEmpty() ):
        current= stiva.pop();
        visited.append(current);
        if(problem.isGoalState(current)):
                path = calculatePath(father,current,problem.getStartState());
                break;
        for node, direc, cost in problem.getSuccessors(current) :
                f = 0;
                if node not in visited:
                        f=1;
                if f==1:
                        stiva.push(node);
```

```
                        father[node] = (current,direc);

    return path
    util.raiseNotDefined()

def calculatePath(father, goal, start):
    path = []
    while(goal != start):
        path.append(father[goal][1])
        goal = father[goal][0];
    path.reverse();
    return path;

def breadthFirstSearch(problem):

    from util import Queue

    queue = Queue();
    visited_grey = [];
    visited = [];
    father = {} #dictionary that will help me make the output list of directi
    path = []

    queue.push(problem.getStartState());
    current = problem.getStartState();
    visited_grey.append(current);
    #father[current] = 0;


    while (not queue.isEmpty() ):
        current = queue.pop();
        if(problem.isGoalState(current)):
                path= calculatePath(father,current,problem.getStartState());
                break;
        for node, direc, cost in problem.getSuccessors(current) :
                f = 0;
                if node not in visited and node not in visited_grey:
                        f=1;
                if f==1:
                        visited_grey.append(node);
                        father[node] = (current,direc);
                        queue.push(node);
        visited.append(current);

    return path
    util.raiseNotDefined()

def breadthFirstSearch2(problem):

    from util import Queue
```

```
    queue = Queue();
    visited = [];
    path = []

    current = problem.getStartState()
    if(problem.isGoalState(current)):
        return path

    queue.push((current, []))
    while not queue.isEmpty():
        current, path = queue.pop()

        if (not current in visited) :
            visited.append(current)

            if problem.isGoalState(current) :
                return path
            succ = problem.getSuccessors(current)
            for node, direc, cost in succ:
                queue.push((node, path + [direc]))

    return path
    util.raiseNotDefined()
def uniformCostSearch(problem):

    #pe asta tre sa l facem amu cu PriorityQueue
    #punem un element in coada si facem suma costului pana acolo


    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    from util import PriorityQueue
    pqueue = PriorityQueue();
    visited = [];
    visited_grey = [];
    father = {} #dictionary that will help me make the output list of directi
    co = {}
    pqueue.push(problem.getStartState(), 0);  #incepem cu cost zero (cost zer
    current = problem.getStartState();
    father[current] = 'Nimic';
    co[current] = 0;

    while (not pqueue.isEmpty() ):
        current = pqueue.pop();
        if(problem.isGoalState(current)):
                path= calculatePath(father,current,problem.getStartState());
                break;
        for node, direc, cost in problem.getSuccessors(current) :
                f = 0;
```

```
                if node not in visited:
                        if node in visited_grey:
                                if co[node] > (co[father[node][0]] + cost):
                                        father[node] = (current,direc,cost);
                                        co[node] = co[father[node][0]] + cost
                                        pqueue.update(node,co[node]);
                        elif node not in visited:
                                f=1;
                        if f==1:
                                visited_grey.append(node);
                                father[node] = (current,direc,cost);
                                co[node] = co[father[node][0]] + cost
                                pqueue.update(node,co[node]);
        visited.append(current);

    return path;
    util.raiseNotDefined()

def nullHeuristic(state, problem=None):
    """
    A heuristic function estimates the cost from the current state to the ne
    goal in the provided SearchProblem.  This heuristic is trivial.
    """
    return 0

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first.
    "*** YOUR CODE HERE ***"

    from util import PriorityQueue

    start = problem.getStartState()
    print start,
    path = [] #drumul care ma duce la goal
    pqueue = PriorityQueue()
    visited = [] #nodurile deja vizitate

    if(problem.isGoalState(start)):
        return path;

    pqueue.push((start,path,0), 0) #primul zero este costul curent, al doilea

    while(not pqueue.isEmpty()):
        current,path,cost_acum = pqueue.pop()
        if not(current in visited):
                visited.append(current)

                if(problem.isGoalState(current)):
                        break;
```

```
            for node, direc, cost in problem.getSuccessors(current) :
                    cost_nou = cost_acum + cost
                    heuristicCost = cost_nou + heuristic(node, problem)
                    pqueue.push((node,path + [direc],cost_nou), heuristic

return path
```

Listing A.2: Weighted A*

```
def aStarWeighted(problem, heuristic=nullHeuristic):
    w = 2 #ponderea
    from util import PriorityQueue

    pqueue = PriorityQueue()
    start = problem.getStartState()
    path = []
    visited = []
    heuristicCost = w * heuristic(start,problem)
    data = (start,[],0)
    pqueue.push(data,0);
    while( not pqueue.isEmpty() ):
        current,path,g = pqueue.pop();
        if not(current in visited):
                visited.append(current)
                if(problem.isGoalState(current)):
                        break;
                for node,direc,cost in problem.getSuccessors(current):
                        heuristicCost = w * heuristic(node,problem)
                        data = (node,path+[direc],g+cost)
                        pqueue.push(data, heuristicCost)

    return path;
    util.raiseNotDefined()
```

Listing A.3: Find all Corners

```
def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman sta
        space)
        """
        "*** YOUR CODE HERE ***"
        start = self.startingPosition
        visited = ()

        try:
            i = self.corners.index(start)
        except:
            i = -1

        for j in range(0,4):
            if j == i:
                visited = visited + (True,)
            else:
                visited = visited + (False,)


        return (start, visited)

def isGoalState(self, state):
        """
        Returns whether this search state is a goal
        state of the problem.
        """
        "*** YOUR CODE HERE ***"
        first_co, second_co, third_co, fourth_co = state[1]
        if first_co and second_co and third_co and fourth_co:
            return True
        return False

def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
        Directions.EAST, Directions.WEST]:
            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitWalls = self.walls[nextx][nexty]

            if not hitWalls:
                next_state = (nextx, nexty)
                visited = state[1]
```

```
            if next_state in self.corners:
                i = self.corners.index(next_state)
                visited_local = ()
                for j in range(0, 4):
                    if j == i:
                        visited_local = visited_local + (True,)
                    else:
                        visited_local = visited_local + (visited[j],)
                visited = visited_local

            successors.append(((next_state, visited), action, 1))


        self._expanded += 1 # DO NOT CHANGE
        return successors
```

Listing A.4: Corners Heuristic

```
x, y = state[0]
    visited = state[1]

    distMax = -1
    for i in range(0,4):
        # nod vizitat
        if visited[i]:
            continue
        coordX, coordY = corners[i]

        #calculez distanta Manhattan
        d = abs(x-coordX) + abs(y-coordY);

        if(distMax < d):
                distMax = d

    if distMax == -1:
        distMax = 0

    return distMax
```

Listing A.5: DiagonalHeuristic

```
def diagonalDistance(position, problem, info={}):
        xy1 = position
        xy2 = problem.goal
        return max(abs(xy1[0]-xy2[0]), abs(xy1[1]-xy2[1]))
```