Department of Computer Science
Technical University of Cluj-Napoca

Introduction to Artificial Intelligence

Laboratory activity

Name: Nechifor Cosmin-Petru
Group: 30232
Email: cosmin.nechifor96@gmail.com

Assoc. Prof. dr. eng. Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

Rules and policies

Lab organisation.

1. Laboratory work is 20% from the final grade.
2. There are 3 deliverables in total.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro

Class:	Introducere in Inteligenta Artificiala
Enrollment key:	Iia2017-2018

4. *Laptop policy*: you can use your own laptop as long you have Linux. One goal of the laboratory is to increase your competency in Linux. It is **your** task to set static IPs:

	IP:	192.168.1.51[1]
	MASK:	255.255.255.0
	GATEWAY:	192.168.1.2
	DNS:	192.168.1.2
	PROXY	192.168.1.2:3128
Wifi:	Network:	isg
	Password:	inteligentaartificiala

5. *Group change policy.* Maximum number of students in a class is 14.
6. *For students repeating the class*: A discussion for validating the previous grade is mandatory in the first week. I usually have no problem to validate your previous grades, as long you request this in the first week. Failing to do so, leads to the grade 1 for the laboratory work in the current semester.

Grading. Assessment aims to measure your knowledge and skills needed to function in realistic AI-related tasks. Assessment is based on your written report explaining the nature of the project, findings, and recommendations. Meeting the deadlines is also important. Your report is comparable to ones you would write if you were a consultant reporting to a client.

Grade inflation makes difficult to distinguish between students. It also discourages the best students to do their best. In my quest for "optimal ranking of the students", I do not use the following heuristics:

Table 1.1: Lab scheduling.

Activity	**Deadline**
Searching agents, linux, latex, python	W_1
Uninformed search	W_2
Informed Search	W_3
Adversial 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}
Individual feedback to clarify the good/bad issues related to student activity/results during the semester.	W_{14}

- "He worked hard at the project". Our society do not like anymore individuals that are *trying*, but individual that *do* stuff. Such heuristic is not admissible in education, except the primary school.
- "I knew he could do much better". Such a heuristic is not admissible because it does not encourage you to spread yourself.
- 7 means that you: i) constantly worked during classes, ii) you proved competent to use the tool and its expressivity for a realistic scenario, iii) you understood theoretical concepts on which the tool rely on.
- 8, 9 mean that your code is large enough and the results proved by your experiments are significant.
- 10 means that you did very impressive work or more efficient that I expected or handled a lot of special cases for realistic scenarios.
- 5 means that you managed to develop something of your own, functional, with your own piece of code substantially different from the examples available.
- You obtain less than 5 in one of the following situations:
  1. few code written by yourself.
  2. too much similarity with the provided examples.
  3. non-seriosity (i.e. re-current late at classes, playing games, worked for other disciplines, poor/unprofessional documentation of your work, etc.)[2].
- You get 2 if you present the project but fail to submit the documentation or code. You get 1 if you do not present your project before the deadline. You get 0 for any line of code

[2]Consider non-seriosity as a immutable boolean value that is unconsciously activated in my brain when one of the above conditions occurs for the first time.

taken from other parts that appear in section *My own code.* For information on TUCN's regulations on plagiarism do consult the active norms.

If your grade is 0, 1, or 2, you do not satisfy the preconditions for participating to the written exam. The only possibility to increase your laboratory grade is to take another project in the next year, at the same class, and to make all the steps again.

However, don't forget that focus is on learning, not on grading.

Using Latex in your documentation. You have to show some competency on writing documentation in Latex. For instance, you have to employ various latex elements: lists, citations, footnotes, verbatim, maths, code, etc.

Plagiarism. Most of you consider plagiarism only a minor form of cheating. This is far from accurate. Plagiarism is passing off the work of others as your own to gain unfair advantage.

During your project presentation and documentation, I must not be left with doubts on which parts of your project are your work or not. Always identify both: 1) who you worked with and 2) where you got your part of the code or solution. You should sign the declaration of originality.

Describe clearly the starting point of your solution. List explicitly any code re-used in your project. List explicitly any help (including debugging help, design discussions) provided by others (including colleagues or teaching assistant). Keep in mind that it is your own project and not the teaching assistant's project. Learning by collaborating does remain an effective method. You can use it, but don't forget to mention any kind of support. Learning by exploiting various knowledge-bases developed by your elder colleagues remain also an effective method for "learning by example". When comparing samples of good and poor assignments submitted by your colleagues in earlier years try to identify which is better and why. You can use this repository of previous assignments, but don't forget to mention any kind of inspiration source.

The assignment is designed to be individual and to pose you some difficulties (both technological and scientific) for which you should identify a working solution by the end of the semester. Each semester, a distinct AI tool is assigned to two students. Your are encouraged to collaborate, especially during the the installation and example understanding phases (W_1-W_4). The quicker you get throughout these preparatory stages, the more time you have for your own project.

Class attendance. I expecte active participation at all activities. Keep in mind the exam can include any topic that was covered during class, explained on the board, or which emerged from discussions among participants. Missing lab assignments or midterm leads to minimum grade for that part. You are free to manage your laboratory classes - meaning that you can submit the project earlier - as long as you meet all the constraints and deadlines.

Chapter 2

A1: Search

Q1: DFS - Depth First Search. Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. It can also be applied in a maze because, after all, a maze is a graph. It can also be used to generate graphs.

In other words: After you are in the maze and you have multiple ways, choose anyone and move forward, keep choosing a way which was not seen so far till you exit the maze or reach dead end. If you exit maze, you are done. If you reach dead end, this is wrong path, so take one step back, choose different path. If all paths are seen in this, take one step back and repeat.

Time complexity: $O(|V| + |E|)$ for explicit graphs traversed without repetition
$O(b^d)$ $O(b^d)$ - implicit graphs with brenching factor b and searched to depth d For more information about timecomplexity you can visit: https://en.wikipedia.org/wiki/Depth-firstsearch

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Figure 2.1: DFS pseudocode

One thing I realisez after implementing the algorithm is that it doesn't always give the optimal path. I would have expected dfs to be optimal.

In the implementation I used Object Oriented Principles(OOP) in order to make my life easier.

The class that I created is called CustomNode, and it looks like that:

```python
# Defining a class node which will help  me implement the alg in a much easier way
class CustomNode:
    def __init__(self, parent, action, state):
        self.parent = parent
        self.action = action
        self.state = state

    def getParent(self):
        return self.parent
    def getAction(self):
        return self.action
    def getState(self):
        return self.state
```

Figure 2.2: CustomNode class

The parrent field will remember where the node came from. Action is the move that was taken and State is the position in labyrinth.

Implementation of the DFS:

```python
# DFS implementation for pacman
def depthFirstSearch(problem):

    visited = dict() #used to keep track of visited nodes
    state = problem.getStartState()
    stack = util.Stack() # we will need a stack in order to implement the algorithm
    node = CustomNode(None, None, state) # CustomNode was design only for DFS, BFS
    stack.push(node)

    while not stack.isEmpty():
        # getting the next node in the stack
        node = stack.pop()
        state = node.getState()

        # if is visited we try something else
        if visited.has_key(hash(state)):
            continue

        # else if is not visited we assign it as visited
        visited[hash(state)] = True

        # if we end up in the goal state, we return the path to it
        if problem.isGoalState(state) == True:
            return getPath(node)

        for child in problem.getSuccessors(state):
            if not visited.has_key(hash(child[0])):
                # child[0] position/state of the next node
                # child[1] direction
                # we create a new node
                nextNode = CustomNode(node, child[1], child[0])
                # we add it to the stack
                stack.push(nextNode)
    return [] # if it doesn't find any path we will return an empty list
```

Figure 2.3: DFS code

Every node has a list of actions and a parent. When the pacman is in the goal state, the algorithm will stop and it will return the path. This was done with the help of the getPath function. It recives the ending node. Then it goes parent by parent and puts every action that was taken in a list of actions at position 0 because we have to reverse the path.

Results:

```python
def getPath(node):
    path = []
    while node.getAction() != None:
        path.insert(0, node.getAction())
        node = node.getParent()
    return path
```

Figure 2.4: getPath function

```
Question q1
===========

*** PASS: test_cases/q1/graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'D', 'C']
*** PASS: test_cases/q1/graph_bfs_vs_dfs.test
***     solution:       ['2:A->D', '0:D->G']
***     expanded_states:    ['A', 'D']
*** PASS: test_cases/q1/graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases/q1/graph_manypaths.test
***     solution:       ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:    ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases/q1/pacman_1.test
***     pacman layout:      mediumMaze
***     solution length: 130
***     nodes expanded:     146

### Question q1: 3/3 ###
```

Figure 2.5: DFS autograder results

Q2: BFS - Breadth-first search For the implementation of the Breadth First Search algorithm I've reused the code from DFS implementation. The only thing that was different it's the auxiliar structure. DFS uses a Stack, and BFS uses a Queue.

Pseudocode:

```
BFS (Graph, root):
    create empty queue Q
    Q.enqueue(root)
    while Q is not empty:
        current = Q.dequeue()
        for each node n that is adjacent to current:
            Q.enqueue(n)
```

Figure 2.6: BFS pseudocode

As I said i replaced the stack with a queue. The implementation is almost the same.

```python
# BFS implementation for pacman
def breadthFirstSearch(problem):
    visited = dict()
    # getting the initial possition of the pacman
    state = problem.getStartState()

    queue = util.Queue() # we will need a queue in order to implement the algorithm
    node = CustomNode(None, None, state) # CustomNode was design only for DFS, BFS
    queue.push(node)

    while not queue.isEmpty():
        # getting the next node in the queue
        node = queue.pop()
        state = node.getState()

        # if is visited we try something else
        if visited.has_key(state):
            continue

        # else if is not visited we assign it as visited
        visited[state] = True

        # if we end up in the goal state, we return the path to it
        if problem.isGoalState(state) == True:
            return getPath(node)

        for child in problem.getSuccessors(state):
            if not visited.has_key(hash(child[0])):
                # child[0] position/state of the next node
                # child[1] direction
                # we create a new node
                nextNode = CustomNode(node, child[1], child[0])
                # we add it to the queue
                queue.push(nextNode)

    return [] # if it doesn't find any path we will return an empty list
```

Figure 2.7: BFS implementation

After running the algorithm I realised that this one gives the optimal solution, but it expands more nodes. (check fig 2.8)

```
Question q2
===========

*** PASS: test_cases/q2/graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***     solution:       ['1:A->G']
***     expanded_states:    ['A', 'B']
*** PASS: test_cases/q2/graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases/q2/graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     269

### Question q2: 3/3 ###
```

Figure 2.8: BFS result

Q3: UCS - Uniform Cost Search Uniform Cost Search is the best algorithm for a search problem, which does not involve the use of heuristics. It can solve any general graph for optimal cost. Uniform Cost Search as it sounds searches in branches which are more or less the same in cost.

Uniform Cost Search again demands the use of a priority queue. Recall that Depth First Search used a priority queue with the depth upto a particular node being the priority and the path from the root to the node being the element stored. The priority queue used here is similar with the priority being the cumulative cost upto the node. Unlike Depth First Search where the maximum depth had the maximum priority, Uniform Cost Search gives the minimum cumulative cost the maximum priority.

(Text from : https://algorithmicthoughts.wordpress.com/2012/12/15/artificial-intelligence-uniform-cost-searchucs/)

The algorithm using this priority queue is the following:

```
Insert the root into the queue
While the queue is not empty
    Dequeue the maximum priority element from the queue
    (If priorities are same, alphabetically smaller path is chosen)
    If the path is ending in the goal state, print the path and exit
    Else
        Insert all the children of the dequeued element, with the cumulative costs as priority
```

Figure 2.9: Uniform Cost Search pseudocode

Implementation:

I have used a new class CustomNodeUniform which inherits from CustomNode, and has one more field called cost. Class implementation:

```python
class CustomNodeUniform(CustomNode):
    def __init__(self, parent, action, state, cost):
        CustomNode.__init__(self, parent=parent, action=action, state=state)
        self.cost = cost
    def getCost(self):
        return self.cost
```

Figure 2.10: Uniform Cost Search code

Using this cost and the priority queue we are able to always get the best action.

```python
# Uniform cost search implementation for pacman
def uniformCostSearch(problem):
    visited = dict()
    state = problem.getStartState()
    # This algorithm uses a Priority Queue
    queue = util.PriorityQueue()

    # CustomNodeUnform was designed for this tipe of problem
    # It inherits from CustomNode
    node = CustomNodeUnform(parent=None, action=None, state=state, cost=0)
    queue.push(node, node.getCost())

    while not queue.isEmpty():
        node = queue.pop()
        state = node.getState()
        cost = node.getCost()

        if visited.has_key(state):
            continue
        visited[state] = True
        # we find the point
        if problem.isGoalState(state) == True:
            return getPath(node)

        for child in problem.getSuccessors(state):
            if not visited.has_key(hash(child[0])):
                # child[2] = cost
                # UNIFORM-COST SEARCH expands the node n with the lowest path cost g(n)
                nextNode = CustomNodeUnform(node, child[1], child[0], cost=(cost + child[2]))
                queue.push(nextNode, nextNode.getCost())

    return []  # if it doesn't find any path we will return an empty list
```

Figure 2.11: Uniform Cost Search code

I am always takeing the cost required from start to the state node and adding the cost required to get in the next node, then adding them to the priority queue, which will put on the first postion the node that has the lowest path cost. As I observed, this algorithm is optimal and takes less space then BFS.

Autograder result:

```
Question q3
===========

*** PASS: test_cases/q3/graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases/q3/graph_bfs_vs_dfs.test
***     solution:       ['1:A->G']
***     expanded_states:    ['A', 'B']
*** PASS: test_cases/q3/graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases/q3/graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q3/ucs_0_graph.test
***     solution:       ['Right', 'Down', 'Down']
***     expanded_states:    ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q3/ucs_1_problemC.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     269
*** PASS: test_cases/q3/ucs_2_problemE.test
***     pacman layout:      mediumMaze
***     solution length: 74
***     nodes expanded:     260
*** PASS: test_cases/q3/ucs_3_problemW.test
***     pacman layout:      mediumMaze
***     solution length: 152
***     nodes expanded:     173
*** PASS: test_cases/q3/ucs_4_testSearch.test
***     pacman layout:      testSearch
***     solution length: 7
***     nodes expanded:     14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***     solution:       ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C']

### Question q3: 3/3 ###
```

Figure 2.12: Autograder result

Q4: A* search algorithm A* is widely used in pathfinding and graph traversal, the process of plotting an efficiently directed path between multiple points, called nodes. It enjoys widespread use due to its performance and accuracy.

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes.

f(n)=g(n)+h(n), where n is the last node on the path g(n) is the cost of the path from the start node to n, and h(n) is a heuristic that estimates the cost of the cheapest path from n to the goal. (Wikipedia)

In order to implement this I've used a new class called CustomNodeAStar which inherits from CustomNode, and has two more fields called cost that keeps track of the cost, exactly as in uniform cost search and eval which will evaluate(estimate) the distance to the goal.

```
class CustomNodeAStar(CustomNode):
    def __init__(self, parent, action, state, cost, eval):
        CustomNode.__init__(self, parent, action, state)
        self.cost = cost
        self.eval = eval

    def getCost(self):
        return self.cost
    def getEval(self):
        return self.eval
```

Figure 2.13: CustomNodeAStar class

Implementation of the algorithm:

```python
# A* implementation for pacman
# The big difference between A* an UC is that, A* has a brain
def aStarSearch(problem, heuristic=nullHeuristic):

    # this algo. also uses a PriorityQueue
    priorityQueue = util.PriorityQueue()
    visited = dict()

    state = problem.getStartState()
    # CustomNodeAStar was designed for this tipe of problem
    # It inherits from CustomNodeAStar
    node = CustomNodeAStar(None, None, state, 0, heuristic(state, problem))
    priorityQueue.push(node, node.getCost() + node.getEval())

    while not priorityQueue.isEmpty():
        node = priorityQueue.pop()
        state = node.getState()
        cost = node.getCost()

        # if is visited we try something else
        if visited.has_key(state):
            continue

        # else -> make it visited
        visited[state] = True
        if problem.isGoalState(state) == True:
            return getPath(node)


        for child in problem.getSuccessors(state):
            if not visited.has_key(child[0]):
                # createing the node, using the formula f = g + h(estimated cost from state to goal)
                nextNode = CustomNodeAStar(parent=node, action=child[1], state=child[0], cost= (child[2] + cost), eval=heuristic(child[0], problem))
                priorityQueue.push(nextNode, nextNode.getCost() + nextNode.getEval())

    return []  # if it doesn't find any path we will return an empty list
```

Figure 2.14: A* code

Autograder result:

```
Question q4
===========

*** PASS: test_cases/q4/astar_0.test
***     solution:        ['Right', 'Down', 'Down']
***     expanded_states:    ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:        ['0', '0', '2']
***     expanded_states:    ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:        ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:        ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:        ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

Figure 2.15: A* autograder

From all the implemented search algorithms A* is the only one which has a brain. It is optimal and runs pretty fast.

Q5: Corners Problem implementation of functions In this part we had to implement the constructor, and the methods getStartState, isGoalState and getSuccessors.

In the constructor we will have a list of corner state which will have 4 values initialised to 0, meaning that the pacman hasn't visited any corner yet. Top and right are used to keep track of maze size. And there is a validation which checks if start position is actualy a corner, and if it is it will change the value of corner state which is at that index.

The code is very explicit. The idea of how to remember visited corners I got it from Github by looking at other people projects.

```
def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()
    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height-2, self.walls.width-2
    self.corners = ((1,1), (1,top), (right, 1), (right, top))
    for corner in self.corners:
        if not startingGameState.hasFood(*corner):
            print 'Warning: no food in corner ' + str(corner)
    self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
    # Please add any code here which you would like to use
    # in initializing the problem
    self.visited, self.visitedList = {}, []
    corner_state = [0, 0, 0, 0]
    # check if start position is in any corner
    if self.startingPosition in self.corners:
        idx = self.corners.index(self.startingPosition)
        corner_state[idx] = 1
    self.startState = (self.startingPosition, tuple(corner_state))

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

Figure 2.16: Corners Problem init

After I realised how init works it was easy to implement the following functions.

GetStartState and isGoalStat were not so hard to implement. I tried to be as clear as possible in my comments.

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

def isGoalState(self, state):
    # cheks if there is no 0 in corner_state
    # in other words, if we visited all 4 corners this means we are in the goal state
    isGoal = not (0 in state[1])

    if isGoal:
        self.visitedList.append(state[0])
        import __main__
        if '_display' in dir(__main__):
            if 'drawExpandedCells' in dir(__main__._display):
                __main__._display.drawExpandedCells(self.visitedList)
    return isGoal
```

Figure 2.17: Corners Start and Goal init

And the last function was getSuccessors which had to gave me a list of successors of the current node. I hope I was clear enough in the comments. Implementation:

```python
def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.
    """
    # the list of successors that we will return
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
        # coordinates of the pacman
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        # finding the next position
        nextx, nexty = int(x + dx), int(y + dy)
        # and if that move doesn't bring us in a wall
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            # we chose it as next state
            nextState = (nextx, nexty)
            cost = 1
            # if that state is a corner then we change the value in the list
            # so it will show that it was visited
            if nextState in self.corners:
                # we get the index of the corner in corners_state (needed in order to make a change)
                index = self.corners.index(nextState)
                corner_state = list(deepcopy(state[1]))
                # makeing the corner visited
                corner_state[index] = 1
            # else we just repeat the algorithm
            else:
                corner_state = list(deepcopy(state[1]))
            successors.append(((nextState, tuple(corner_state)), action, cost))

    # Bookkeeping for display purposes
    if state not in self.visited:
        self.visited[state] = True
        self.visitedList.append(state[0])

    self._expanded += 1
    return successors
```

Figure 2.18: Corners getSuccessors

Result:

```
Question q5
===========

*** PASS: test_cases/q5/corner_tiny_corner.test
***     pacman layout:      tinyCorner
***     solution length:        28

### Question q5: 3/3 ###
```

Figure 2.19: Q5 Result - Autograder

Q6: Heuristic for CornersProblem In this function i have applied the heuristic that was taught at the Laboratory and it proved to be very efficient. I have tried 3-4 heuristics till I got to this point. Those heuristics could be found in the body of the function comented. It works on a simple principle, first we found the closest corner, and then we calculate using the labyrinth length and height the remaining moves required to achive the goal state.

```python
def cornersHeuristic(state, problem):
    corners = problem.corners  # These are the corner coordinates
    walls = problem.walls  # These are the walls of the maze, as a Grid (game.py)
    position = state[0]
    corner_state = list(state[1])
    # print corner_state
    if problem.isGoalState(state):
        return 0
    cost = 0
    numberOfCorners = 0
    # for each corner
    for i in range(0, 4):
        # if corner not visited
        if corner_state[i] == 0:
            numberOfCorners += 1
            # getting the value of the distance from position to each
            # unvisited corner
            dist = util.manhattanDistance(position, corners[i])
            if cost == 0 or cost > dist:
                # getting the distance to the closest corner
                cost = dist
    height = corners[3][1] # height
    length = corners[3][0] # length
    small = None
    if height < length:
        small = height
    else:
        small = length
    # depending on the number of corners we will apply a different formula
    if numberOfCorners == 2:
        cost += (small - 1)
    if numberOfCorners == 3:
        if height < length:
            cost += (height - 1) + (length - 1)
        else:
            cost += (length - 1) + (height - 1)
    if numberOfCorners == 4:
        if height < length:
            cost += 2 * (height - 1) + (length - 1)
        else:
            cost += 2 * (length - 1) + (height - 1)
    return cost
```

Figure 2.20: Corners heuristic

Result:

```
Question q6
===========

*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North', 'North', 'North', 'North', 'Nor
path length: 106
*** PASS: Heuristic resulted in expansion of 846 nodes

### Question q6: 3/3 ###
```

Figure 2.21: Q6 Result - Autograder

Q7: Heuristic for the problem of eating all the food-dots In this problem we try to find the longest manhattanDistance in the labyrinth. between pacman's position and any point which has a dot.

```
# 3rd
pacmanPossition, foodGrid = state
score = 0

for row in enumerate(foodGrid):
    # col = (col id, col)
    for col in enumerate(row[1]):
        matrix_position = (row[0], col[0])
        if col[1]:
            score = max(score, util.manhattanDistance(pacmanPossition, matrix_position))
return score
```

Figure 2.22: Food heuristic

Result:

```
Question q7
===========

*** PASS: test_cases/q7/food_heuristic_1.test
*** PASS: test_cases/q7/food_heuristic_10.test
*** PASS: test_cases/q7/food_heuristic_11.test
*** PASS: test_cases/q7/food_heuristic_12.test
*** PASS: test_cases/q7/food_heuristic_13.test
*** PASS: test_cases/q7/food_heuristic_14.test
*** PASS: test_cases/q7/food_heuristic_15.test
*** PASS: test_cases/q7/food_heuristic_16.test
*** PASS: test_cases/q7/food_heuristic_17.test
*** PASS: test_cases/q7/food_heuristic_2.test
*** PASS: test_cases/q7/food_heuristic_3.test
*** PASS: test_cases/q7/food_heuristic_4.test
*** PASS: test_cases/q7/food_heuristic_5.test
*** PASS: test_cases/q7/food_heuristic_6.test
*** PASS: test_cases/q7/food_heuristic_7.test
*** PASS: test_cases/q7/food_heuristic_8.test
*** PASS: test_cases/q7/food_heuristic_9.test
*** FAIL: test_cases/q7/food_heuristic_grade_tricky.test
***     expanded nodes: 9551
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 3/4 ###
```

Figure 2.23: Q7 result - autograder

Table 2.1: Results

Algorithm	Parameters	Maze	Agents	Cost	Running time	Exp. nodes
DFS	dfs	tinyMaze	SearchAgent	10		15
DFS	dfs	mediumMze	SearchAgent	130		146
DFS	dfs	bigMaze	SearchAgent	210	0,2	390
BFS	bfs	tinyMaze	SearchAgent	8		15
BFS	bfs	mediumMze	SearchAgent	68	0,1	269
BFS	bfs	bigMaze	SearchAgent	210	0,4	620
UCS	ucs	tinyMaze	SearchAgent	8		15
UCS	ucs	mediumMze	SearchAgent	68	0,1	269
UCS	ucs	bigMaze	SearchAgent	210	0,8	620
UCS	ucs	mDottedMaze	StayEastSA	1	0,1	186
UCS	ucs	mDottedMaze	StayWestSA	1,70E+10	0,1	169
UCS	ucs	mScaryMaze	StayEastSA	1	0,2	230
UCS	ucs	mScaryMaze	StayWestSA	6,90E+10	0,012	108
aStart	astar, h=M	bigMaze	SearchAgent	210	0,7	549
aStart	astar, h=null	bigMaze	SearchAgent	210	0,8	620
aStart	bfs,p=CP	tinyMaze	SearchAgent	28	0,01	252
aStart	bfs,p=CP	mediummze	SearchAgent	106	0,2	1966
aStart	a*,p=CP h=cH	tinyMaze	SearchAgent	28	0,1	173
aStart	a*,p=CP h=cH	mediummze	SearchAgent	106	0,6	846

Chapter 3

A2: Logics

Chapter 4

A3: Planning

Bibliography

Appendix A

Your original code

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. Including in this section any line of code taken from someone else leads to failure of IS class this year.

Intelligent Systems Group