Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Știrb Călin-Alexandru
Name: Horst Jaden
Group: 30432
Email: stirbalex@yahoo.com
Email: jadenhorst@gmail.com

Teaching Assistant: Adrian Groza
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

1.1 Introduction:

- **This chapter covers the search algorithms used in a game of Pacman in order to solve a list of problems, and determine which algorithm is better.**
- **One such problem is finding the fastest route to a single Pacman "food" object placed at predefined position in the maze**
- **Another problem is finding an optimal path for Pacman to eat the food objects placed at each of the corners of the maze**
- **Finally we have the problem where we need Pacman to eat all of the multiple food objects scattered inside of the maze**

1.2 Search Algorithms:

The following search algorithms have been implemented in order to solve the problems:

1.2.1 Depth First Search (DFS)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch of a graph before backtracking. This algorithm uses a stack to visit nodes.

```
def depthFirstSearch(problem):
  """
  Search the deepest nodes in the search tree first [p 85].

  Your search algorithm needs to return a list of actions that
     reaches
  the goal.  Make sure to implement a graph search algorithm
     [Fig. 3.7].

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

```
11
12    print("Start:", problem.getStartState())
13    print("Is the start a goal?",
          problem.isGoalState(problem.getStartState()))
14    print("Start's successors:",
          problem.getSuccessors(problem.getStartState()))
15
16    queue = util.Stack()
17    visitedNodes = []
18    start = problem.getStartState()
19    firstNode = (start,[])
20
21    queue.push(firstNode)
22
23    while not queue.isEmpty():
24      state,actions = queue.pop()
25      if state not in visitedNodes:
26        visitedNodes.append(state)
27        if problem.isGoalState(state):
28          return actions
29        else:
30          successors = problem.getSuccessors(state)
31          for state,action,cost in successors:
32            newAction = actions + [action]
33            newNode = (state,newAction)
34            queue.push(newNode)
35    return []
```

1.2.2 Breadth First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all neighbors of a node before moving on to their child nodes. It searches level by level, making it suitable for finding the shortest path in unweighted graphs. This algorithm uses a queue to visit nodes.

```
1  def breadthFirstSearch(problem):
2      """Search the shallowest nodes in the search tree first."""
3      "*** YOUR CODE HERE ***"
4      open_ds = util.Queue()
5
6      start = [problem.getStartState(), ""]
7      open_ds.push([start])
8
9      visited_state = [start[0]]
10
11     while not open_ds.isEmpty():
12         node = open_ds.pop()
13         end = node[-1]
14
15         if problem.isGoalState(end[0]):
16             return [state[1] for state in node[1:]]
```

```
17
18          successors = problem.getSuccessors(end[0])
19          for succ in successors:
20              if succ[0] not in visited_state:
21                  visited_state.append(succ[0])
22                  new_node = copy.deepcopy(node)
23                  new_node.append(succ)
24                  open_ds.push(new_node)
25
26      return []
```

1.2.3 Uniform-Cost Search (UCS)

Uniform-Cost Search (UCS) is a graph traversal algorithm used to find the path with the lowest cost in a weighted graph. It explores nodes in increasing order of path cost, ensuring that it always considers the least costly path first. The algorithm uses a priority queue to visit nodes

```
1  def uniformCostSearch(problem):
2      """Search the node of least total cost first."""
3      open_ds = util.PriorityQueue()
4
5      start = [problem.getStartState(),""]
6      open_ds.push([start], 0)
7      visited_state = []
8
9      while not open_ds.isEmpty():
10         node = open_ds.pop()
11         end = node[-1]
12
13         if problem.isGoalState(end[0]):
14             return [state[1] for state in node[1:]]
15
16         if end[0] not in visited_state:
17             visited_state.append(end[0])
18             successors = problem.getSuccessors(end[0])
19             for succ in successors:
20                 new_node = node + [succ]
21                 total_cost = problem.getCostOfActions([state[1]
                       for state in new_node[1:]])
22                 open_ds.push(new_node, total_cost)
23
24     return []
```

1.2.4 A star (A*)

A* Search is a graph traversal algorithm that finds the shortest path from a start node to a goal node in a weighted graph. It combines the benefits of Dijkstra's algorithm and heuristic approaches. A* uses a priority queue to explore nodes, considering both the cost to reach a node and an estimated cost to reach the goal.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    "*** YOUR CODE HERE ***"
    print "Start:", problem.getStartState()
    print "Is the start a goal?",
       problem.isGoalState(problem.getStartState())
    print "Start's successors:",
       problem.getSuccessors(problem.getStartState())

    current = problem.getStartState()
    priority_queue = util.PriorityQueue()
    visited = []
    solution = []

    if problem.isGoalState(current):
        print "Solution: ", solution
        return solution

    while not problem.isGoalState(current):
        if current not in visited:
            visited.append(current)
            successors = problem.getSuccessors(current)

            for nextState, action, step_cost in successors:
                temp = solution + [action]
                que_pos = problem.getCostOfActions(temp) +
                   heuristic(nextState,problem)
                priority_queue.push((temp,nextState),que_pos)
        aux = priority_queue.pop()
        solution = aux[0]
        current = aux[1]

    return solution
    util.raiseNotDefined()
```

1.2.5 RandomSearch algorithm

This algorithm, as the name suggests, picks a successors at random from the successor list and traverses them until the goal state of the problem has been reached

```
def RandomSearch(problem):
    print "Start:", problem.getStartState()
    print "Is the start a goal?",
       problem.isGoalState(problem.getStartState())
    print "Start's successors:",
       problem.getSuccessors(problem.getStartState())

    solution =[]
    current = problem.getStartState()

    while not problem.isGoalState(current):
```

```
        successors = problem.getSuccessors(current)
        if not successors:
            return []
        random_successor = random.choice(successors)
        next_state, action, _ = random_successor
        solution.append(action)
        current = next_state
    return solution
```

1.2.6 GreedyBestFirstSearch algorithm

Greedy Best-First Search is an informed search algorithm that aims to find a path from the start state to a goal state by prioritizing states based on a heuristic evaluation of their potential to reach the goal. This algorithm expands states with the lowest heuristic values, as it assumes that the states closer to the goal are more promising.
The algorithm is efficient for finding solutions in scenarios where a good heuristic estimate of the distance to the goal is available. However, it may not guarantee the optimal solution and can get stuck in local optima when the heuristic is not admissible.

```
    def greedyBestFirstSearch(problem, heuristic=nullHeuristic):
    print "Start:", problem.getStartState()
    print "Is the start a goal?",
       problem.isGoalState(problem.getStartState())
    print "Start's successors:",
       problem.getSuccessors(problem.getStartState())

    start_state = problem.getStartState()
    open_list = util.PriorityQueue()
    start_heuristic = heuristic(start_state, problem)
    open_list.push((start_state, [], start_heuristic),
       start_heuristic)
    visited = set()

    while not open_list.isEmpty():
        current_state, actions, h_val = open_list.pop()
        if problem.isGoalState(current_state):
            return actions
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state,action,_ in successors:
                if next_state not in visited:
                    h_next = heuristic(next_state,problem)
                    open_list.push((next_state,actions
                       +[action], h_val), h_val)
    return []
```

1.3 Problem Introductions

In this Section we delve into the logics and hueristics employed by Pacman to navigate the game world's mazes effectively. The main focus of this chapter is to present the main functions implemented in order to achieve optimal results for the Four Corners problem as well as the Food Search problem

1.3.1 Four Corners Problem

One of the key challenges in the Pacman world is to visit all four corners of a given maze. Achieving this involves the use of algorithms that can determine Pacman's path. In this chapter we will explore the logic behind the "Four Corners Problem"

1.3.2 Algorithmic insight

We delve into the algorithms that drive Pacman's journey through the maze. The algorithms we examine include Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS) and A Star Search (A*). These algorithms play a pivotal role in determining Pacman's path, the order in which it explores the maze, and ultimately achieving the goal of visiting all four corners.

1.3.3 Food Search Problem

This problem almost functions like the original game of pacman (minus the ghosts), in which the search agent has to find the best path that collects all of the food objects inside of a maze.

1.4 The four corners problem

This section will go into detail on how the 4 corners problem was solved

1.4.1 The getStartState function

- This function returns the starting state for the Pacman game
- A state is represented as a tuple which consists of the Pacman's current coordinates
- The coordinates are represented as a tuple of the form **(x,y)**
- The second object of the tuple, '()' is used to represent the corners visited corners
- Essentially the function returns the starting coordinates of Pacman and an empty visited corners list

```
1  def getStartState(self):
2    return (self.startingPosition, ())
```

1.4.2 The isGoalState function

- This function checks if the given state is a goal state for the corner's problem
- The state parameter is expected to be the same as the returned value of **getStartState**, meaning a touple of the form **((x,y), [visitedCorners])**
- The goal of the Pacman game is to visit all 4 corners of the maze where food has been placed
- To determine if the state is a goal, the function compares the set of visited corners (set(state[1])) with the set of all corners in the maze (set(self.corners))
- If the sets are equal then the function returns 'True' otherwise it returns 'False'

```
def isGoalState(self, state):
    return set(state[1]) == set(self.corners)
```

1.4.3 The getSuccsessors function

- This function is repsonsible for generating a list of succsessors states based on the Pacman's current state in the game
- The current state is of the same form as it is for the isGoalState function
- The function iterates over each possible action, meaning: **Nort, South, East and West**
- For each action it calculates the next position 'sucPos' based on the current state and the action taken
- It checks whether the position after taking the action hits a wall by examining the 'self.walls' atribute; if the move is hits a wall it is considered an **illegal action**
- If the next position is a valid move, the successor state is generated
- if the found position is a not yet visited corner, it adds the corner into the 'corners' list by updating the state
- The function keeps track of how many search nodes it has expanded for analysis purposes
- The function returns the list of successors states 'successors'

```
def getSuccessors(self, state):
    successors = []
    currentPos,corners = state
    for action in [Directions.NORTH, Directions.SOUTH,
       Directions.EAST, Directions.WEST]:
        x, y = currentPos
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:#if it doesnt hit wall
            sucPos = (nextx, nexty)
            sucCorner = list(corners)
            if sucPos in self.corners and sucPos not in corners:
```

```
                sucCorner.append(sucPos)
            newSuccession = ((sucPos, tuple(sucCorner)), action,
                1)
            successors.append(newSuccession)
    self._expanded += 1 # DO NOT CHANGE
    return successors
```

1.4.4 The cornerHeuristic function

- This function is used to calculate a **heuristic value** for the four corners problem of the Pacman game
- The function takes two parameters: 'state' (the current state, of the same form as the previous functions) and 'problem' (the problem instance, which contains the information about the maze)
- It calculates the set of unvisited corners by subtracting the set of visited corners 'vCorners' from the set of all corners 'problem.corners'
- Within the loop the function calculates the **Manhattan distances** between the current position and all of the unvisited corners
- The index of the closest corner is determined using the **distances.index(min(distances))** operation
- After identifying the closest corner, we use it to calculate the Manhattan distance between it and the current position
- This distance is added to the 'heuristic' variable, which accumulates the estimated cost of reaching the closest corner
- The current position is updated to be the closest corner for the next iteration of the loop
- In summary, the **cornersHeuristic** function calculates the heuristic value based on the Manhattan distances of the unvisited corners

```
def cornersHeuristic(state, problem):
    curPos, vCorners = state
    unvisited_corners = set(problem.corners) - set(vCorners)

    heuristic = 0

    while unvisited_corners:
        distances = [util.manhattanDistance(curPos, corner) for
            corner in unvisited_corners]
        closest_index = distances.index(min(distances))
        closest_corner = unvisited_corners[closest_index]
        heuristic += util.manhattanDistance(curPos,
            closest_corner)
        curPos = closest_corner
        unvisited_corners.remove(closest_corner)

    return heuristic
```

1.5 The Food Search Problem

In this section we will go into detail on how the heuristic function for this problem was implemented

1.5.1 The foodHeuristic function

- The 'foodHeuristic' function calculates a heuristic estimate for the cost to collect all remaining food objects in the Pacman game
- It takes 2 parameters: the current game state and the problem instance
- the function defines 2 helper methods: 'find' and 'union' which are used for finding the connected components and performing union operations for MST
- The algorithm constructs a graph of food object distances by calculating the **Manhattan distance** between pairs of food dots
- It then uses Kruskal's algorithm to find the MST of the graph. This MST represents the minimum total distance needed to visit all unvisited food objects
- The heuristic is determined by adding the distance to the nearest unvisited food dot to the total distance represented by the MST
- In summary the foodHeuristic algorithm combines the distance to the nearest unvisited food dot with the minimum total distance represented by the MST of food dot distances

```
def foodHeuristic(state, problem):
    def find(parent, i):
        if parent[i] is None:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        xroot = find(parent, x)
        yroot = find(parent, y)

        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    position, foodGrid = state
    unvisited_foods = foodGrid.asList()

    if not unvisited_foods:
        return 0

    graph = []
```

```
    for i in range(len(unvisited_foods)):
        for j in range(i + 1, len(unvisited_foods)):
          graphList = [unvisited_foods[i], unvisited_foods[j],
             util.manhattanDistance(unvisited_foods[i],
             unvisited_foods[j])]
            graph.append(graphList)

    mst_length = 0
    i = 0
    e = 0

    graph = sorted(graph, key=lambda x: x[2])

    parent = {}
    rank = {}

    for food in unvisited_foods:
        parent[food] = None
        rank[food] = 0

    while e < len(unvisited_foods) - 1:
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
        y = find(parent, v)

        if x != y:
            e += 1
            mst_length += w
            union(parent, rank, x, y)

    closest_food = min([util.manhattanDistance(position, food)
       for food in unvisited_foods])

    return closest_food + mst_length
```

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

The 2nd AI course slides: 02 IA search
The examples given on moodle
Documentation template provided on moodle
https://www.github.com/
https://stackoverflow.com/

Appendix A

Your original code

A.1 Chapter A1:

A.1.1 search.py

```
def depthFirstSearch(problem):
  print("Start:", problem.getStartState())
  print("Is the start a goal?",
     problem.isGoalState(problem.getStartState()))
  print("Start's successors:",
     problem.getSuccessors(problem.getStartState()))

  queue = util.Stack()
  visitedNodes = []
  start = problem.getStartState()
  firstNode = (start,[])

  queue.push(firstNode)

  while not queue.isEmpty():
    state,actions = queue.pop()
    if state not in visitedNodes:
      visitedNodes.append(state)
      if problem.isGoalState(state):
        return actions
      else:
        successors = problem.getSuccessors(state)
        for state,action,cost in successors:
          newAction = actions + [action]
          newNode = (state,newAction)
          queue.push(newNode)
  return []

def breadthFirstSearch(problem):
    open_ds = util.Queue()

    start = [problem.getStartState(), ""]
    open_ds.push([start])
```

```
32
33      visited_state = [start[0]]
34
35      while not open_ds.isEmpty():
36          node = open_ds.pop()
37          end = node[-1]
38
39          if problem.isGoalState(end[0]):
40              return [state[1] for state in node[1:]]
41
42          successors = problem.getSuccessors(end[0])
43          for succ in successors:
44              if succ[0] not in visited_state:
45                  visited_state.append(succ[0])
46                  new_node = copy.deepcopy(node)
47                  new_node.append(succ)
48                  open_ds.push(new_node)
49
50      return []
51
52  def uniformCostSearch(problem):
53      open_ds = util.PriorityQueue()
54
55      start = [problem.getStartState(),""]
56      open_ds.push([start], 0)
57      visited_state = []
58
59      while not open_ds.isEmpty():
60          node = open_ds.pop()
61          end = node[-1]
62
63          if problem.isGoalState(end[0]):
64              return [state[1] for state in node[1:]]
65
66          if end[0] not in visited_state:
67              visited_state.append(end[0])
68              successors = problem.getSuccessors(end[0])
69              for succ in successors:
70                  new_node = node + [succ]
71                  total_cost = problem.getCostOfActions([state[1]
                        for state in new_node[1:]])
72                  open_ds.push(new_node, total_cost)
73
74      return []
75
76  def aStarSearch(problem, heuristic=nullHeuristic):
77      open_ds = util.PriorityQueue()
78
79      start_state = problem.getStartState()
```

```
    start = (start_state, "", (0, heuristic(start_state,
       problem)))
    open_ds.push([start], start[2])

    visited_state = {start_state[0]: sum(start[2])}

    while not open_ds.isEmpty():
        node = open_ds.pop()

        if not open_ds.isEmpty():
            next_node = open_ds.pop()
            temp = [node, next_node]

            while not open_ds.isEmpty() and sum(node[-1][2]) ==
               sum(next_node[-1][2]):
                next_node = open_ds.pop()
                temp.append(next_node)

            tie_nodes = temp if sum(node[-1][2]) ==
               sum(next_node[-1][2]) else temp[:-1]

            for n in tie_nodes:
                if node[-1][2][0] < n[-1][2][0]:
                    node = n

            for n in temp:
                if node != n:
                    open_ds.push(n, sum(n[-1][2]))

        end = node[-1]
        if end[0] not in visited_state or sum(end[2]) <=
           visited_state[end[0]]:
            if problem.isGoalState(end[0]):
                return [state[1] for state in node[1:]]

            successors = problem.getSuccessors(end[0])
            for succ in successors:
                gn_succ = end[2][0] + succ[2]
                hn_succ = heuristic(succ[0], problem)
                fn_succ = gn_succ + hn_succ
                if succ[0] not in visited_state or fn_succ <
                   visited_state[succ[0]]:
                    visited_state[succ[0]] = fn_succ
                    new_node = copy.deepcopy(node)
                    new_succ = (succ[0], succ[1], (gn_succ,
                       hn_succ))
                    new_node.append(new_succ)
                    open_ds.push(new_node, fn_succ)
    return []

```

```
124 def RandomSearch(problem):
125     print "Start:", problem.getStartState()
126     print "Is the start a goal?",
          problem.isGoalState(problem.getStartState())
127     print "Start's successors:",
          problem.getSuccessors(problem.getStartState())
128
129     solution =[]
130     current = problem.getStartState()
131
132     while not problem.isGoalState(current):
133         successors = problem.getSuccessors(current)
134         if not successors:
135             return []
136         random_successor = random.choice(successors)
137         next_state, action, _ = random_successor
138         solution.append(action)
139         current = next_state
140     return solution
141
142 def greedyBestFirstSearch(problem, heuristic=nullHeuristic):
143     print "Start:", problem.getStartState()
144     print "Is the start a goal?",
          problem.isGoalState(problem.getStartState())
145     print "Start's successors:",
          problem.getSuccessors(problem.getStartState())
146
147     start_state = problem.getStartState()
148     open_list = util.PriorityQueue()
149     start_heuristic = heuristic(start_state, problem)
150     open_list.push((start_state, [], start_heuristic),
          start_heuristic)
151     visited = set()
152
153     while not open_list.isEmpty():
154         current_state, actions, h_val = open_list.pop()
155         if problem.isGoalState(current_state):
156             return actions
157         if current_state not in visited:
158             visited.add(current_state)
159             successors = problem.getSuccessors(current_state)
160             for next_state,action,_ in successors:
161                 if next_state not in visited:
162                     h_next = heuristic(next_state,problem)
163                     open_list.push((next_state,actions
                        +[action], h_val), h_val)
164     return []
```

A.1.2 searchAgents.py

```
1
2  def getStartState(self):
3      return (self.startingPosition, ())
4
5  def isGoalState(self, state):
6      return set(state[1]) == set(self.corners)
7
8  def getSuccessors(self, state):
9          successors = []
10         currentPos,corners = state
11         for action in [Directions.NORTH, Directions.SOUTH,
               Directions.EAST, Directions.WEST]:
12             x, y = currentPos
13             dx, dy = Actions.directionToVector(action)
14             nextx, nexty = int(x + dx), int(y + dy)
15             if not self.walls[nextx][nexty]:#if it doesnt hit
                   wall
16                 sucPos = (nextx, nexty)
17                 sucCorner = list(corners)
18                 if sucPos in self.corners and sucPos not in
                       corners:
19                     sucCorner.append(sucPos)
20                 newSuccession = ((sucPos, tuple(sucCorner)),
                       action, 1)
21                 successors.append(newSuccession)
22         self._expanded += 1 # DO NOT CHANGE
23         return successors
24
25 def cornersHeuristic(state, problem):
26     curPos, vCorners = state
27     unvisited_corners = set(problem.corners) - set(vCorners)
28
29     heuristic = 0
30
31     while unvisited_corners:
32         distances = [util.manhattanDistance(curPos, corner) for
               corner in unvisited_corners]
33         closest_index = distances.index(min(distances))
34         closest_corner = unvisited_corners[closest_index]
35         heuristic += util.manhattanDistance(curPos,
               closest_corner)
36         curPos = closest_corner
37         unvisited_corners.remove(closest_corner)
38
39     return heuristic
40
41 def foodHeuristic(state, problem):
42     def find(parent, i):
```

```
        if parent[i] is None:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        xroot = find(parent, x)
        yroot = find(parent, y)

        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    position, foodGrid = state
    unvisited_foods = foodGrid.asList()

    if not unvisited_foods:
        return 0

    graph = []
    for i in range(len(unvisited_foods)):
        for j in range(i + 1, len(unvisited_foods)):
          graphList = [unvisited_foods[i], unvisited_foods[j],
             util.manhattanDistance(unvisited_foods[i],
             unvisited_foods[j])]
            graph.append(graphList)

    mst_length = 0
    i = 0
    e = 0

    graph = sorted(graph, key=lambda x: x[2])

    parent = {}
    rank = {}

    for food in unvisited_foods:
        parent[food] = None
        rank[food] = 0

    while e < len(unvisited_foods) - 1:
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
        y = find(parent, v)

        if x != y:
```

```
91              e += 1
92              mst_length += w
93              union(parent, rank, x, y)
94
95      closest_food = min([util.manhattanDistance(position, food)
            for food in unvisited_foods])
96
97      return closest_food + mst_length
```