Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Fucă Răzvan-Ionuț, Goina Radu
Group: 33432
Email: razvanfuca@gmail.com, r.goina@yahoo.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

This chapter covers the search problem and uses the classic arcade game of Pacman as the backdrop in order to explore multiple common search algorithms.

We have implemented both optimal(A*) and non-optimal(DFS) searching algorithms.

Searching can be applied in different ways depending on the game state and the definition of the goal state. In this project, we've used these algorithms to tackle the following tasks:

- Finding a single position in a maze
- Visiting every corner of in the maze
- Eating all the food in the maze
- All of the above but with the added twist of using warp tunnels

List of implemented algorithms:

- Depth First Search
- Breadth First Search
- Uniform Cost Search
- A*

1.0.1 Corners Problem

This problem's goal is to find a path that reaches every corner in the maze. In order to solve this problem we must define an appropriate state space, a getSuccessors() function and heuristic (for A*).

We chose to encode the state as a tuple of Pacman's current position and a list of the unvisited corners.

In getSuccessors() we remove we remove a corner from the unvisited list as soon as it is reached.

The corners heuristic is designed to take into account the smallest distance between Pacman and the and any of the unvisited corners and the number of walls that lie between him and any unvisited corner.

The number of walls between Pacman and a given corner is computed using the following function:

```
def wallCounting(position, destination, walls):
    start_row, start_col = position
    end_row, end_col = destination
    wall_count = 0

    if start_row > end_row:
        start_row, end_row=end_row,start_row

    if start_col > end_col:
        start_col, end_col=end_col, start_col

    # Count walls in columns
    for row in range(start_row, end_row +1):
        for col in range(start_col, end_col +1):
            if walls[row][col]==True:
                wall_count+=1
    return wall_count
```

It is important to note that the heuristic takes into account the walls between Pacman and the corner only when distance(pacman, corner) ¡ wallCounting(Pacman, corner, walls) in order to preserve it's admissability

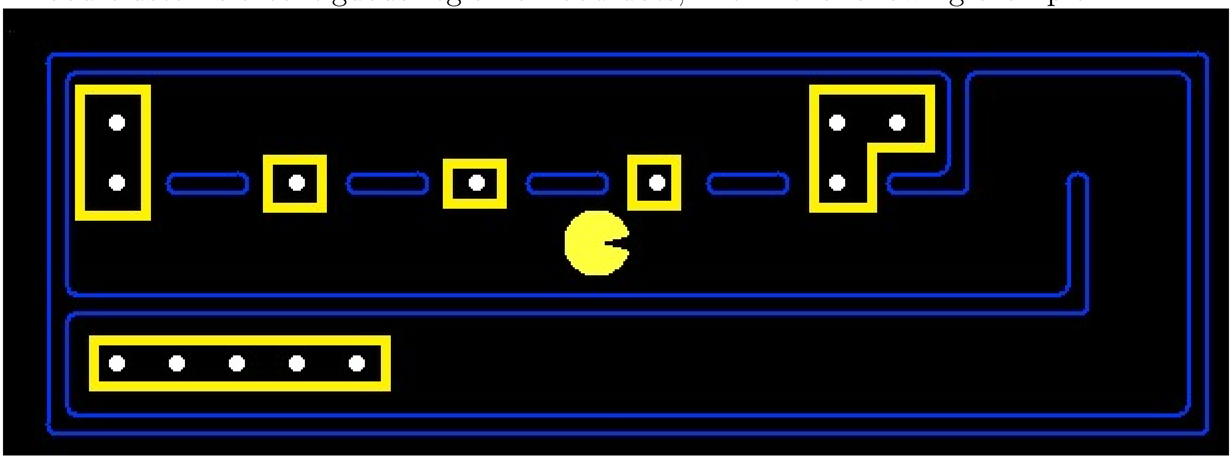
This heuristic proves to be highly effective, requiring only 1041 node expansions in the mediumCorners layout, resulting 6/6 score according to the autograder.

1.0.2 Food Search Problem

This problem's goal is for pacman to eat all of the food on the map

This problem is solved using a cluster approach:

A food cluster is a contiguous region of food dots, like in the following example:



```
    def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):
        position = state[0]
        foodGrid= state[1]
        clusters = state[2]
        foodList=foodGrid.asList()
        walls = problem.walls  # These are the walls of the maze, as a Grid (game.py)

```

```
        flag=0

        if problem.isGoalState(state):
            return 0

        max_distance = 0
        for cluster in clusters:
            distance=9999

            smallest_wall_count=9999
            for elem in cluster:
                if util.manhattanDistance(elem,position) < distance:
                    distance=util.manhattanDistance(elem,position)
                    elem_to_remove=elem

                if wallCounting(position,elem,walls) < smallest_wall_count:
                    smallest_wall_count=wallCounting(position,elem,walls)

            for i in range(len(cluster)):
                if cluster[i]!=elem_to_remove:
                    distance+=1

            if max_distance >= smallest_wall_count:
                distance+=smallest_wall_count

            if distance > max_distance:
                max_distance=distance

        return max_distance
```

We find the clusters using the find_clusters function that was not written by us but by a large language model by giving it the signature and the requirements of the function. This significantlly reduced the time spent on developing and debugging this functionality.

```

def find_clusters(foodList):
    clusters = []
    visited = set()

    def is_valid_position(position):
        return position in foodList and position not in visited

    def dfs(position, cluster):
        if not is_valid_position(position):
            return
        visited.add(position)
        cluster.append(position)
        row, col = position
```

```
            for dr, dc in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                new_position = (row + dr, col + dc)
                if is_valid_position(new_position):
                    dfs(new_position, cluster)

    for position in foodList:
        if is_valid_position(position):
            cluster = []
            dfs(position, cluster)
            if cluster:
                clusters.append(tuple(cluster))  # Convert the list to a tuple before appending

    return tuple(map(tuple, clusters))  # Convert the list of tuples to a tuple of tuples
```

Keeping this in mind, the heuristic considers the distance to the cluster, the number of elements of the cluster and the number of walls between Pacman and the cluster.

This heuristic appears to excel in the context of layouts like trickySearch, where it expands only 6980 nodes and even earns a bonus point (8/7) from the autograder. However, it does exhibit some limitations.

It is great on sparse layouts like trickySearch due to the high amount of clusters, but on layouts saturated with food (one big cluster) it performs much worse. This is because it needs to perform DFS to find the food clusters (in our case one), thus transforming this heuristic into one where only the distance betwen pacman and a single food dot is taken into account but with the added overhead of performing a DFS.

Despite its limitations, we have chosen to present this solution as it offers an intriguing approach to addressing the specific challenge of maze navigation and search.

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

Appendix A

Your original code

search.py :

```
def depthFirstSearch(problem: SearchProblem):

    frontier = util.Stack()
    visited = set()
    actions = []
    frontier.push( (problem.getStartState(), actions) )
    while not frontier.isEmpty():
        currentNode, actions = frontier.pop()

        if problem.isGoalState(currentNode):
            return actions
        if currentNode not in visited:
            visited.add(currentNode)
            for node, action, _ in problem.getSuccessors(
               currentNode):
                new_path = actions.copy()
                new_path.append(action)
                frontier.push((node, new_path))

    return []


def breadthFirstSearch(problem: SearchProblem):
    visited=set()

    states=util.Queue()
    dir_taken=util.Queue()

    visited.add(problem.getStartState())
    states.push(problem.getStartState())
    curr_path=[]

    while not states.isEmpty():
        curr_state=states.pop()
        if not dir_taken.isEmpty():
```

```python
            curr_path=dir_taken.pop()

        if(problem.isGoalState(curr_state)):
           return curr_path

        for succesor in problem.getSuccessors(curr_state):
            if succesor[0] not in visited:
                states.push(succesor[0])
                visited.add(succesor[0])

                new_path = curr_path[:]
                new_path.append(succesor[1])

                dir_taken.push(new_path)
    return None

def uniformCostSearch(problem: SearchProblem):
    visited = set()
    states_directions = util.PriorityQueue()

    start_state = problem.getStartState()
    states_directions.push((start_state, [], 0), 0)

    while not states_directions.isEmpty():
        curr_state, curr_path, curr_cost = states_directions.pop()

        if problem.isGoalState(curr_state):
            return curr_path

        if curr_state not in visited:
            visited.add(curr_state)
            for state, direction, cost in problem.getSuccessors(curr_state):
                if state not in visited:
                    new_path = curr_path + [direction]
                    new_cost = problem.getCostOfActions(new_path)
                    states_directions.push((state, new_path, new_cost), new_cost)

    return []

def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):
    visited = set()
    states_directions = util.PriorityQueue()

    start_state = problem.getStartState()
    states_directions.push((start_state, [], 0), 0)

    while not states_directions.isEmpty():
```

```python
        curr_state, curr_path, curr_cost = states_directions.pop()

        if problem.isGoalState(curr_state):
            return curr_path

        if curr_state not in visited:
            visited.add(curr_state)
            for state, direction, cost in problem.getSuccessors(curr_state):
                if state not in visited:
                    new_path = curr_path + [direction]
                    new_cost = problem.getCostOfActions(new_path) + heuristic(state,problem)
                    states_directions.push((state, new_path, new_cost), new_cost)

    return []
```

searchAgents.py:

```python
class CornersProblem:
    def getSuccessors(self, state: Any):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            x,y = state[0]
            notVisited=state[1]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                nextCoordonate=(nextx,nexty)
                if nextCoordonate in self.corners and nextCoordonate in notVisited:
                    element_to_remove = nextCoordonate
                    notVisited = tuple(item for item in notVisited if item != element_to_remove)

                if self.portals[nextx][nexty]!=0:
                    for portalCoord,portalType in self.portals.asListNotNull():
                        if portalCoord != nextCoordonate and portalType == self.portals[nextx][nexty]:
                            successors.append(((portalCoord, notVisited),action,1))
                else:
                    successors.append(((nextCoordonate, notVisited),action,1))
```

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


    def cornersHeuristic(state: Any, problem: CornersProblem):
        corners = problem.corners # These are the corner coordinates
        walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

        position, notVisited = state

        min = 999999

        for corner in corners:
            if corner in notVisited:
                manhattanHeuristicVal = util.manhattanDistance(corner, position)
                wallCountingVal = wallCounting(position, corner, walls)

                if manhattanHeuristicVal < wallCountingVal:
                    distanceAprox = manhattanHeuristicVal + wallCountingVal
                else:
                    distanceAprox = manhattanHeuristicVal

                if distanceAprox < min:
                    min = distanceAprox

        return min if min != 999999 else 0
```

Changes to layouts.py to support warp tunnels:

```
    def processLayoutChar(self, x, y, layoutChar):
    if layoutChar == '%':
        self.walls[x][y] = True
    elif layoutChar == '.':
        self.food[x][y] = True        # Added a new Layout for Red and Blue portals.
    elif layoutChar == 'B':           # 0 -> no portal
        self.portals[x][y] = 1        # 1 -> Blue portal
    elif layoutChar == 'R':           # 2 -> Red portal
        self.portals[x][y] = 2
    elif layoutChar == 'o':
        self.capsules.append((x, y))
    elif layoutChar == 'P':
        self.agentPositions.append( (0, (x, y) ) )
    elif layoutChar in ['G']:
        self.agentPositions.append( (1, (x, y) ) )
```

```
            self.numGhosts += 1
        elif layoutChar in  ['1', '2', '3', '4']:
            self.agentPositions.append( (int(layoutChar), (x,y)))
            self.numGhosts += 1
```

Changes to graphicsDisplay.py to support warp tunnels:

```
        def drawPortal(self, portals):
        portalImages={}
        for portalCoord,type in portals.asListNotNull():
            (screen_x, screen_y) = self.to_screen(portalCoord)
            dot = circle( (screen_x, screen_y),
                                  PORTAL_SIZE * self.gridSize,
                                  outlineColor = PORTAL_COLORS[type-1],
                                  fillColor = PORTAL_COLORS[type-1],
                                  width = 1)
            sdot = circle( (screen_x, screen_y),
                                  PORTAL_SIZE* 0.65 * self.gridSize,
                                  outlineColor = PORTAL_COLORS[type+1],
                                  fillColor = PORTAL_COLORS[type+1],
                                  width = 1)
            portalImages[portalCoord]=dot
        return portalImages
```

Changes to game.py to support warp tunnels:

```
    class Configuration:
        def generateSuccessor(self, vector,PortalUsed=False):
            if PortalUsed == False:
                x, y= self.pos
                dx, dy = vector
                direction = Actions.vectorToDirection(vector)
                if direction == Directions.STOP:
                    direction = self.direction # There is no stop direction
                return Configuration((x + dx, y+dy), direction)
            else:
                direction = Directions.WEST
                return Configuration(vector,direction)
```

Changes to pacman.py to support warp tunnels:

```
    def applyAction( state, action ):
        legal = PacmanRules.getLegalActions( state )
        if action not in legal:
            raise Exception("Illegal action " + str(action))

        pacmanState = state.data.agentStates[0]
        (pacmanx, pacmany)=pacmanState.getPosition()
        type =state.data.layout.portals[pacmanx][pacmany]
        # Update Configuration
        if type!=0:
```

```
            twinPortalPosition=None
            for portal,types in state.data.layout.portals.
               asListNotNull():
                if types == type and portal!=(pacmanx, pacmany):
                    twinPortalPosition=portal
            print(twinPortalPosition)
            pacmanState.configuration=pacmanState.configuration.
               generateSuccessor(twinPortalPosition,PortalUsed=
               True)

        vector = Actions.directionToVector( action, PacmanRules.
           PACMAN_SPEED )
        pacmanState.configuration = pacmanState.configuration.
           generateSuccessor( vector )

        # Eat
        next = pacmanState.configuration.getPosition()
        nearest = nearestPoint( next )
        if manhattanDistance( nearest, next ) <= 0.5 :
            # Remove food
            PacmanRules.consume( nearest, state )
    applyAction = staticmethod( applyAction )
```