Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name:
Morari Camelia
Trif Oriana Maria
Group: 30433
Email: camelus.morari@gmail.com
oriana.trif04@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

Introduction: Pacman Search Algorithm

Search algorithms play a crucial role in guiding Pacman through the maze to find optimal paths, avoid ghosts, and achieve its goals. The Pacman search problem involves navigating a dynamic environment with obstacles, power pellets, and moving adversaries.

In particular, we focus on the Pacman search algorithm, which is designed to systematically explore the game state space and make decisions that lead to successful navigation. We discuss the algorithm's key components, strategies for efficient exploration, and its application in solving the complex problems posed by the Pacman environment.

The problems solved by the search algorithms:

1.1 Algorithms

- Depth First Search
- Breadth First Search
- Uniform Cost Search
- A* Search
- Fringe Search
- Iterative Deepening A*
- Bidirectional Search
- Eat All The Food
- Go Into All Corners
- Distance Heuristics

1.1.1 Depth First Search

Introduction: Code explanation

The code is an implementation of a depth-first search (DFS) algorithm to solve a problem. DFS is a graph traversal algorithm that explores as far down a branch of the search space as possible before backtracking. It starts from the initial state, uses a stack to explore nodes, and maintains a set of visited nodes to avoid revisiting them. The algorithm pops nodes from the stack, checks if they are the goal state, and returns a sequence of actions if a solution is found. If

not, it expands the node by generating successor states and appends the corresponding actions, pushing them onto the stack. This process continues until a goal state is found, or the stack becomes empty, resulting in an empty list returned to indicate no solution.

```
def depthFirstSearch(problem):
    fringe = util.Stack()
    visited = set()
    actionList = []
    fringe.push((problem.getStartState(), actionList))
    while fringe:
        node, actions = fringe.pop()
        if node not in visited:
            visited.add(node)
            if problem.isGoalState(node):
                return actions
            for successor in problem.getSuccessors(node):
                state, action, cost = successor
                nextActions = actions + [action]
                fringe.push((state, nextActions))
    return []
```

1.1.2 Breadth First Search

Introduction: Code explanation

The function implements a breadth-first search algorithm for solving a problem defined by the problem parameter. It initializes the search with the starting state and an empty list of moves, maintains a queue to manage state exploration, and systematically explores states while tracking the actions taken. The function continues to explore states until either the goal state is reached or the queue becomes empty, at which point it returns the list of moves. The code snippet is missing the definition of the util module and ends with util.raiseNotDefined(), indicating that this code should be part of a larger program where the util module is properly imported and raiseNotDefined() is defined elsewhere.

```
def breadthFirstSearch(problem):
    start = problem.getStartState()
    currentPosition = (start, [])
    explored = []
    moves = []
    queue = util.Queue()
    queue.push(currentPosition)

    while not queue.isEmpty() and not
       problem.isGoalState(currentPosition):
        currentPosition, moves = queue.pop()

        if currentPosition not in explored:
            explored.append(currentPosition)

            if not problem.isGoalState(currentPosition):
                possibleMoves = problem.getSuccessors(currentPosition)

                for position, action, cost in possibleMoves:
                    move = moves + [action]
```

```
                    node = (position, move)
                    queue.push(node)
    return moves
```

1.1.3 Uniform Cost Search

Introduction: Code explanation

The provided Python code defines a function **uniformCostSearch(problem)** that performs a uniform cost search to find the optimal path from the start state to the goal state in a problem. It uses a priority queue to explore states based on their cumulative cost, maintains a set of visited states, and returns the sequence of actions for the optimal path to the goal state. The code requires the util module for data structures and ends with a return statement, returning an empty list if no valid path is found.

```
def uniformCostSearch(problem):
    fringe = util.PriorityQueue()
    visited = set()
    actionList = []
    fringe.push((problem.getStartState(), actionList), 0)
    while fringe:
        node, actions = fringe.pop()
        if node not in visited:
            visited.add(node)
            if problem.isGoalState(node):
                return actions
            for successor in problem.getSuccessors(node):
                state, action, cost = successor
                nextActions = actions + [action]
                nextCost = problem.getCostOfActions(nextActions)
                fringe.push((state, nextActions), nextCost)
    return []
```

1.1.4 A* Search

Introduction: Code explanation

The provided Python code defines a function for performing an A* search on a problem instance. It starts by initializing the search with the starting state and an empty list of moves, setting up data structures for explored states and a priority queue for state prioritization. The search proceeds, prioritizing states based on their estimated total cost (actual cost plus a heuristic value). The loop continues until the queue is empty. When a goal state is found, the function returns the list of moves representing the optimal path. If no goal state is reached, the function returns an empty list. The function is designed to accept different heuristics and data structures. It also supports either sets or lists for tracking explored states.

```

def aStarSearch(problem, heuristic=nullHeuristic,
   explored_structure=list, queue_structure=util.PriorityQueue):
    start = problem.getStartState()
    currentPosition = (start, [], 0)
    explored = explored_structure()
    moves = []
    queue = queue_structure()
```

```
    queue.push(currentPosition, heuristic(start, problem))

    while not queue.isEmpty():
        currentPosition, moves, currentCost = queue.pop()

        if problem.isGoalState(currentPosition):
            return moves

        if currentPosition not in explored:
            explored.add(currentPosition) if isinstance(explored,
               set) else explored.append(currentPosition)

            possibleMoves = problem.getSuccessors(currentPosition)

            for position, action, cost in possibleMoves:
                move = moves + [action]
                totalCost = cost + currentCost
                totalCostH = totalCost + heuristic(position, problem)
                node = (position, move, totalCost)
                queue.push(node, totalCostH)

    return []
```

1.1.5 Fringe Search

Introduction: Code explanation

Fringe search implements a search algorithm, which can be viewed as a combination of uniform-cost search and A* search. It initializes a priority queue for managing state exploration and a set to keep track of visited states. The search begins with the starting state, and states are prioritized based on their estimated total cost (actual cost plus a heuristic value). The loop continues until the fringe is empty. When a goal state is found, the function returns the list of actions representing the optimal path. It efficiently avoids revisiting already explored states and incorporates the heuristic function for informed search. If no goal state is reached, the function returns None.

```
def fringeSearch(problem, heuristic=nullHeuristic):
    fringe = util.PriorityQueue()
    visited = set()
    actionList = []
    fringe.push((problem.getStartState(), actionList),
       heuristic(problem.getStartState(), problem))

    while not fringe.isEmpty():
        node, actions = fringe.pop()

        if node in visited:
            continue

        visited.add(node)

        if problem.isGoalState(node):
            return actions

```

```
        for successor in problem.getSuccessors(node):
            state, action, cost = successor
            nextActions = actions + [action]
            nextCost = problem.getCostOfActions(nextActions) +
               heuristic(state, problem)

            if state not in visited:
                fringe.push((state, nextActions), nextCost)

    return None
```

1.1.6 Iterative Deepening A*

Introduction: Code explanation

These two functions implement the Iterative Deepening A* search algorithm. iterativeDeepeningAStar gradually increases the depth limit, repeatedly using depthLimitedAStar for depth-limited A* search. It starts with a heuristic estimate and increments the depth limit while searching for a solution. If found, it returns the optimal path; otherwise, it continues searching at a deeper level. depthLimitedAStar explores states within a depth limit, utilizing a priority queue. It avoids revisiting explored states, prioritizes states based on their estimated cost, and continues until the depth limit is reached or a goal state is found. If the goal is reached, it returns the optimal path; otherwise, it returns None. These functions provide an efficient search strategy for problems with unknown or extensive search spaces.

```
def iterativeDeepeningAStar(problem, heuristic=nullHeuristic):
    start_state = problem.getStartState()
    limit = heuristic(start_state, problem)

    while True:
        result = depthLimitedAStar(problem, start_state, heuristic,
           limit)
        if result is not None:
            return result
        limit += 1

def depthLimitedAStar(problem, start_state, heuristic, limit):
    fringe = util.PriorityQueue()
    visited = set()

    fringe.push((start_state, [], 0), 0)

    while not fringe.isEmpty():
        node, actions, cost = fringe.pop()

        if node in visited:
            continue

        visited.add(node)

        if problem.isGoalState(node):
            return actions

        successors = problem.getSuccessors(node)
```

```
        for succ_state, succ_action, succ_cost in successors:
            if succ_state not in visited:
                next_actions = actions + [succ_action]
                next_cost = cost + succ_cost
                h = next_cost + heuristic(succ_state, problem)
                fringe.push((succ_state, next_actions, next_cost), h)

    return None
```

1.1.7 Bidirectional Search

Introduction: Code explanation

This code defines a bidirectional search algorithm for solving a problem by simultaneously exploring from the start and goal states. It maintains two sets of states and moves for the forward and backward searches. It uses two queues to manage exploration in both directions. The algorithm continues until the two searches meet, and when they do, it returns the concatenated path from the start to the goal, effectively finding a solution. If they don't meet, the function returns an empty list.

```
def bidirectionalSearch(problem):
    start = problem.getStartState()
    currentForward = (start, [])
    exploredForward = []
    movesForward = []
    queueForward = util.Queue()
    queueForward.push(currentForward)
    tempForward = []

    end = problem.goal
    currentBackward = (end, [])
    exploredBackward = []
    movesBackward = []
    queueBackward = util.Queue()
    queueBackward.push(currentBackward)
    tempBackward = []

    def oppositeMoves(moves):
        opposite = []

        for move in moves:
            if move == 'North':
                opposite.append('South')
            elif move == 'East':
                opposite.append('West')
            elif move == 'South':
                opposite.append('North')
            elif move == 'West':
                opposite.append('East')

        return opposite

    while not queueForward.isEmpty() and not queueBackward.isEmpty():
        if not queueForward.isEmpty():
```

```
            currentForward, movesForward = queueForward.pop()

            if currentForward not in exploredForward:
                exploredForward.append(currentForward)

                if currentForward not in tempBackward:
                    possibleMoves =
                       problem.getSuccessors(currentForward)
                    for position, action, cost in possibleMoves:
                        move = movesForward + [action]
                        node = (position, move)
                        queueForward.push(node)
                        tempForward.append(position)

                else:
                    while not queueBackward.isEmpty():
                        currentBackward, movesBackward =
                           queueBackward.pop()
                        if currentForward == currentBackward:
                            moves = movesForward +
                               oppositeMoves(movesBackward[::-1])
                            return moves

        if not queueBackward.isEmpty():
            currentBackward, movesBackward = queueBackward.pop()

            if currentBackward not in exploredBackward:
                exploredBackward.append(currentBackward)

                if currentBackward not in tempForward:
                    possibleMoves =
                       problem.getSuccessors(currentBackward)

                    for position, action, cost in possibleMoves:
                        move = movesBackward + [action]
                        node = (position, move)
                        queueBackward.push(node)
                        tempBackward.append(position)

                else:
                    while not queueForward.isEmpty():
                        currentForward, movesForward =
                           queueForward.pop()
                        if currentBackward == currentForward:
                            moves = movesForward +
                               oppositeMoves(movesBackward[::-1])
                            return moves

    return []
```

1.1.8 Eat All The Food

Introduction: Code explanation

This is a Python class named FoodSearchProblem designed for solving a search problem within the context of a Pac-Man game. The class represents a problem where Pac-Man needs to find and consume all the food dots on the grid. The class has methods to define the problem, generate successor states, check if a state is the goal state (all food dots eaten), and calculate the cost of a sequence of actions. The class is initialized with the game's starting state, which includes the Pac-Man's position, the layout of walls, and the locations of food dots. It keeps track of the number of expanded nodes and provides a heuristicInfo dictionary for potential heuristic functions. The getSuccessors method generates successor states by considering Pac-Man's movements in the four cardinal directions and deducts the cost of 1 for each step taken. The getCostOfActions method calculates the total cost of a sequence of actions while ensuring that illegal moves result in a high cost (999999).

```
class FoodSearchProblem:
    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(),
           startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0
        self.heuristicInfo = {}

    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0

    def getSuccessors(self, state):
        successors = []
        self._expanded += 1
        for direction in [Directions.NORTH, Directions.SOUTH,
           Directions.EAST, Directions.WEST]:
            x, y = state[0]
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextFood = state[1].copy()
                nextFood[nextx][nexty] = False
                successors.append((((nextx, nexty), nextFood),
                   direction, 1))
        return successors

    def getCostOfActions(self, actions):
        x, y = self.getStartState()[0]
        cost = 0
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost
```

1.1.9 Go Into All Corners

Introduction: Code explanation

The CornersProblem class is designed to represent a search problem within a Pac-Man game where the goal is to visit all four specific corners of the maze. The class inherits from search.SearchProblem and is initialized with the game's starting state, including the layout of walls, Pac-Man's starting position, and the coordinates of the four corners. It also checks if there is food in each corner and issues a warning if not. The class's methods include getStart-State, which initializes the state with Pac-Man's starting position and an empty list for visited corners; isGoalState, which checks if all four corners have been visited; getSuccessors, which generates successor states by considering Pac-Man's movements in four cardinal directions, marking visited corners along the way; and getCostOfActions, which calculates the cost of a sequence of actions while penalizing illegal moves with a high cost. The class helps facilitate the search for a path that visits all four corners while avoiding walls.

```
class CornersProblem(search.SearchProblem):
    def __init__(self, startingGameState):
        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

    def getStartState(self):
        visited = []
        return (self.startingPosition, visited)

    def isGoalState(self, state):
        return len(state[1]) == 4

    def getSuccessors(self, state):
        currentPosition, foundCorners = state[0], state[1]
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
           Directions.EAST, Directions.WEST]:
            x, y = currentPosition
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                if (nextx, nexty) in self.corners and (nextx, nexty)
                   not in foundCorners:
                    visited = foundCorners + [(nextx, nexty)]
                    successors.append(((((nextx, nexty), visited),
                       action, 1))
                else:
                    successors.append((((nextx, nexty),
                       foundCorners), action, 1))
        self._expanded += 1
        return successors

```

```
    def getCostOfActions(self, actions):
        if actions == None: return 999999
        x, y = self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)


def cornersHeuristic(state, problem):
    corners = problem.corners
    walls = problem.walls
    unvisited = []
    visited = state[1]
    node = state[0]
    heuristic = 0
    for corner in corners:
        if not corner in visited:
            unvisited.append(corner)
    while unvisited:
        distance, corner = min([(util.chebyshevDistance(node,
           corner), corner) \
                                for corner in unvisited])
        heuristic += distance
        node = corner
        unvisited.remove(corner)
    return heuristic
```

1.1.10 Distance Heuristics

Introduction: Code explanation

The first function, chebyshevHeuristic, calculates the Chebyshev heuristic for a given position and problem. This heuristic estimates the minimum number of moves required for a state to reach the goal state by considering only the maximum absolute difference between the x and y coordinates of the two positions.

The second function, ourHeuristic, calculates a custom heuristic for a state in the context of a Pac-Man game. It estimates the cost to reach the nearest food pellet (by using the Chebyshev distance) and then adds the count of remaining food pellets to create the heuristic value. This heuristic encourages Pac-Man to move towards the nearest food and favors states with fewer remaining food pellets. It's designed to guide Pac-Man towards efficient food collection while also accounting for proximity to the nearest food source.

Both of these heuristics can be employed in search algorithms like A* to inform the search process and potentially find more efficient solutions in Pac-Man game scenarios.

```
def ourHeuristic(state, problem):
    position, foodGrid = state

    foodList = foodGrid.asList()

    if len(foodList) == 0:
        return 0
```

```
    closestFood = min(foodList, key=lambda food:
        util.chebyshevDistance(position, food))

    distanceToClosestFood = util.chebyshevDistance(position,
        closestFood)

    heuristic = distanceToClosestFood + len(foodList)

    return heuristic
```

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

1.2 Results

Notes:
SA = SearchAgent.
SESA = StayEastSearchAgent.
SWSA = StayWestSearchAgent.
ManhattanH = ManhattanHeuristic.
EuclideanH = EuclideanHeuristic.
ChebyshevH = ChebyshevHeuristic.
A*CA = A* Corners Agent.
A*FSA = A* Food Search Agent.

Algorithm	Maze	Agent	Score	Cost	Searched Nodes	Time(s)
DFS	tinyMaze	SA	500	10	15	0.0
DFS	mediumMaze	SA	380	130	146	0.0
DFS	bigMaze	SA	300	210	390	0.0
BFS	tinyMaze	SA	502	8	15	0.0
BFS	mediumMaze	SA	442	68	269	0.1
BFS	bigMaze	SA	300	210	620	0.2
UCS	tinyMaze	SA	502	8	15	0.0
UCS	mediumMaze	SA	442	68	269	0.1
UCS	bigMaze	SA	300	210	620	0.3
UCS	mediumDottedMaze	SESA	646	1	186	0.1
UCS	mediumScaryMaze	SWSA	418	68719479864	108	0.0
A*	tinyMaze	SA	512	28	435	0.0
A*	mediumMaze	SA	434	106	2448	0.3
A*	bigMaze	SA	378	162	9904	4.0
A*	mediumCorners	A*CA	434	106	1069	0.1
A*	bigCorners	A*CA	378	162	2211	0.3
A*	tinySearch	A*FSA	573	27	2698	0.9
FGS	tinyMaze	SA	502	8	15	0.0
FGS	mediumMaze	SA	442	68	269	0.1
FGS	bigMaze	SA	300	210	620	0.3
IDA	tinyMaze	SA	502	8	15	0.0
IDA	mediumMaze	SA	442	68	269	0.0
IDA	bigMaze	SA	300	210	620	0.0
BDS	tinyMaze	SA	502	8	12	0.0
BDS	mediumMaze	SA	442	68	170	0.0
BDS	bigMaze	SA	300	210	596	0.0
ManhattanH	mediumMaze	SA	442	68	221	0.0
EuclidianH	mediumMaze	SA	442	68	228	0.0
ChebyshevH	mediumMaze	SA	442	68	226	0.0

Table 1.1: Results

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

Appendix A

Your original code

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

Depth-First Search

```
def depthFirstSearch(problem):
    fringe = util.Stack()
    visited = set()
    actionList = []
    fringe.push((problem.getStartState(), actionList))
    while fringe:
        node, actions = fringe.pop()
        if node not in visited:
            visited.add(node)
            if problem.isGoalState(node):
                return actions
            for successor in problem.getSuccessors(node):
                state, action, cost = successor
                nextActions = actions + [action]
                fringe.push((state, nextActions))
    return []
```

Breadth-First Search

```
def breadthFirstSearch(problem):
    start = problem.getStartState()
    currentPosition = (start, [])
    explored = []
    moves = []
    queue = util.Queue()
    queue.push(currentPosition)

    while not queue.isEmpty() and not
       problem.isGoalState(currentPosition):
        currentPosition, moves = queue.pop()

```

```
        if currentPosition not in explored:
            explored.append(currentPosition)

            if not problem.isGoalState(currentPosition):
                possibleMoves = problem.getSuccessors(currentPosition)

                for position, action, cost in possibleMoves:
                    move = moves + [action]
                    node = (position, move)
                    queue.push(node)
    return moves
```

Uniform Cost Search

```
def uniformCostSearch(problem):
    fringe = util.PriorityQueue()
    visited = set()
    actionList = []
    fringe.push((problem.getStartState(), actionList), 0)
    while fringe:
        node, actions = fringe.pop()
        if node not in visited:
            visited.add(node)
            if problem.isGoalState(node):
                return actions
            for successor in problem.getSuccessors(node):
                state, action, cost = successor
                nextActions = actions + [action]
                nextCost = problem.getCostOfActions(nextActions)
                fringe.push((state, nextActions), nextCost)
    return []
```

A* Search

```
def aStarSearch(problem, heuristic=nullHeuristic,
   explored_structure=list, queue_structure=util.PriorityQueue):
    start = problem.getStartState()
    currentPosition = (start, [], 0)
    explored = explored_structure()
    moves = []
    queue = queue_structure()
    queue.push(currentPosition, heuristic(start, problem))

    while not queue.isEmpty():
        currentPosition, moves, currentCost = queue.pop()

        if problem.isGoalState(currentPosition):
            return moves

        if currentPosition not in explored:
```

```
            explored.add(currentPosition) if isinstance(explored,
               set) else explored.append(currentPosition)

            possibleMoves = problem.getSuccessors(currentPosition)

            for position, action, cost in possibleMoves:
                move = moves + [action]
                totalCost = cost + currentCost
                totalCostH = totalCost + heuristic(position, problem)
                node = (position, move, totalCost)
                queue.push(node, totalCostH)

    return []
```

Fringe Search

```
def fringeSearch(problem, heuristic=nullHeuristic):
    fringe = util.PriorityQueue()
    visited = set()
    actionList = []
    fringe.push((problem.getStartState(), actionList),
       heuristic(problem.getStartState(), problem))

    while not fringe.isEmpty():
        node, actions = fringe.pop()

        if node in visited:
            continue

        visited.add(node)

        if problem.isGoalState(node):
            return actions

        for successor in problem.getSuccessors(node):
            state, action, cost = successor
            nextActions = actions + [action]
            nextCost = problem.getCostOfActions(nextActions) +
               heuristic(state, problem)

            if state not in visited:
                fringe.push((state, nextActions), nextCost)

    return None
```

Iterative Deepening A* Search

```
def iterativeDeepeningAStar(problem, heuristic=nullHeuristic):
    start_state = problem.getStartState()
    limit = heuristic(start_state, problem)
```

```
4
5       while True:
6           result = depthLimitedAStar(problem, start_state, heuristic,
                limit)
7           if result is not None:
8               return result
9           limit += 1
10
11  def depthLimitedAStar(problem, start_state, heuristic, limit):
12      fringe = util.PriorityQueue()
13      visited = set()
14
15      fringe.push((start_state, [], 0), 0)
16
17      while not fringe.isEmpty():
18          node, actions, cost = fringe.pop()
19
20          if node in visited:
21              continue
22
23          visited.add(node)
24
25          if problem.isGoalState(node):
26              return actions
27
28          successors = problem.getSuccessors(node)
29          for succ_state, succ_action, succ_cost in successors:
30              if succ_state not in visited:
31                  next_actions = actions + [succ_action]
32                  next_cost = cost + succ_cost
33                  h = next_cost + heuristic(succ_state, problem)
34                  fringe.push((succ_state, next_actions, next_cost), h)
35
36      return None
```

Bidirectional Search

```
1   def bidirectionalSearch(problem):
2       start = problem.getStartState()
3       currentForward = (start, [])
4       exploredForward = []
5       movesForward = []
6       queueForward = util.Queue()
7       queueForward.push(currentForward)
8       tempForward = []
9
10      end = problem.goal
11      currentBackward = (end, [])
12      exploredBackward = []
13      movesBackward = []
14      queueBackward = util.Queue()
15      queueBackward.push(currentBackward)
```

```
    tempBackward = []

    def oppositeMoves(moves):
        opposite = []

        for move in moves:
            if move == 'North':
                opposite.append('South')
            elif move == 'East':
                opposite.append('West')
            elif move == 'South':
                opposite.append('North')
            elif move == 'West':
                opposite.append('East')

        return opposite

    while not queueForward.isEmpty() and not queueBackward.isEmpty():
        if not queueForward.isEmpty():
            currentForward, movesForward = queueForward.pop()

            if currentForward not in exploredForward:
                exploredForward.append(currentForward)

                if currentForward not in tempBackward:
                    possibleMoves =
                       problem.getSuccessors(currentForward)
                    for position, action, cost in possibleMoves:
                        move = movesForward + [action]
                        node = (position, move)
                        queueForward.push(node)
                        tempForward.append(position)

                else:
                    while not queueBackward.isEmpty():
                        currentBackward, movesBackward =
                           queueBackward.pop()
                        if currentForward == currentBackward:
                            moves = movesForward +
                               oppositeMoves(movesBackward[::-1])
                            return moves

        if not queueBackward.isEmpty():
            currentBackward, movesBackward = queueBackward.pop()

            if currentBackward not in exploredBackward:
                exploredBackward.append(currentBackward)

                if currentBackward not in tempForward:
                    possibleMoves =
                       problem.getSuccessors(currentBackward)

                    for position, action, cost in possibleMoves:
```

```
                        move = movesBackward + [action]
                        node = (position, move)
                        queueBackward.push(node)
                        tempBackward.append(position)

                else:
                    while not queueForward.isEmpty():
                        currentForward, movesForward =
                           queueForward.pop()
                        if currentBackward == currentForward:
                            moves = movesForward +
                               oppositeMoves(movesBackward[::-1])
                            return moves

    return []
```

Food Search Problem

```
class FoodSearchProblem:
    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(),
           startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0
        self.heuristicInfo = {}

    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0

    def getSuccessors(self, state):
        successors = []
        self._expanded += 1
        for direction in [Directions.NORTH, Directions.SOUTH,
           Directions.EAST, Directions.WEST]:
            x, y = state[0]
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextFood = state[1].copy()
                nextFood[nextx][nexty] = False
                successors.append((((nextx, nexty), nextFood),
                   direction, 1))
        return successors

    def getCostOfActions(self, actions):
        x, y = self.getStartState()[0]
        cost = 0
        for action in actions:
```

```
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost
```

Corners Problem

```
class CornersProblem(search.SearchProblem):
    def __init__(self, startingGameState):
        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

    def getStartState(self):
        visited = []
        return (self.startingPosition, visited)

    def isGoalState(self, state):
        return len(state[1]) == 4

    def getSuccessors(self, state):
        currentPosition, foundCorners = state[0], state[1]
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
           Directions.EAST, Directions.WEST]:
            x, y = currentPosition
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                if (nextx, nexty) in self.corners and (nextx, nexty)
                   not in foundCorners:
                    visited = foundCorners + [(nextx, nexty)]
                    successors.append((((nextx, nexty), visited),
                       action, 1))
                else:
                    successors.append((((nextx, nexty),
                       foundCorners), action, 1))
        self._expanded += 1
        return successors

    def getCostOfActions(self, actions):
        if actions == None: return 999999
        x, y = self.startingPosition
        for action in actions:
```

```
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)


def cornersHeuristic(state, problem):
    corners = problem.corners
    walls = problem.walls
    unvisited = []
    visited = state[1]
    node = state[0]
    heuristic = 0
    for corner in corners:
        if not corner in visited:
            unvisited.append(corner)
    while unvisited:
        distance, corner = min([(util.chebyshevDistance(node,
           corner), corner) \
                                for corner in unvisited])
        heuristic += distance
        node = corner
        unvisited.remove(corner)
    return heuristic
```

Our Heuristic

```
def ourHeuristic(state, problem):
    position, foodGrid = state

    foodList = foodGrid.asList()

    if len(foodList) == 0:
        return 0
    closestFood = min(foodList, key=lambda food:
       util.chebyshevDistance(position, food))

    distanceToClosestFood = util.chebyshevDistance(position,
       closestFood)

    heuristic = distanceToClosestFood + len(foodList)

    return heuristic
```

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