UNIVERSITY OF ARKANSAS

CSCE 4613/5613: Introduction to Artificial Intelligence

Homework Assignment #3

Submission Deadline: 11:59PM, 10/22/2023

- This homework contains 3 questions. You are given a pdf file **AI_Homework3.pdf** (this one) containing the submission instructions and the questions from 1 to 3.
- **What to submit:**
  - **A single pdf file** contains your solution and your name.
  - Put all the to-be-submitted files into a folder with the name format as:

    $\{LASTNAME\}$_$\{FIRSTNAME\}$_homework3
- You can discuss with your classmates, but **do not copy** each other work.

Problem 1: Constraints Satisfaction Problem [40 pts]

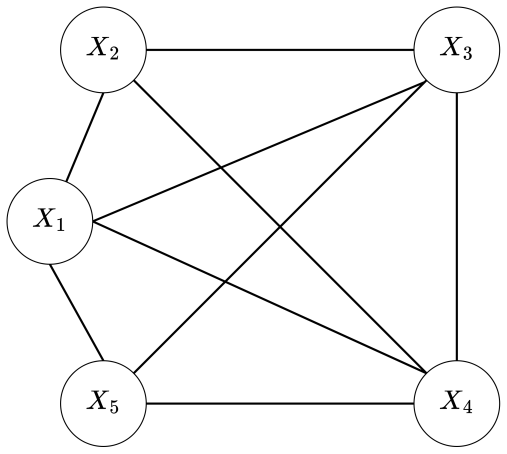


Figure 1: Constraint graph

We are given a constraint graph as can be seen in Figure 1. There are 5 variables $\{X_i : i \in [1, 2, 3, 4, 5]\}$. Denote $D(X_i)$ to be the domain of variable X_i.

$$D(X_1) = [1, 2, 3, 4, 5]$$
$$D(X_2) = [1, 2, 3, 5]$$
$$D(X_3) = [1, 2, 4, 5]$$
$$D(X_4) = [1, 3, 4, 5]$$
$$D(X_5) = [1, 3, 4, 5]$$

Any 2 variables that are connected by an arc in the constraint graph cannot share the same value. For example, if $X_2 = 3$, then X_1, X_3, X_4 cannot take value 3. Denote *assign* to be the assign operation. If we are given the assignment as $assign(X_1 = 3, X_5 = 4)$, that means X_1 is assigned value 3 first, then X_5 is assigned value 4, and other variables are not assigned yet.

Questions:

(a) What are the neighbors of X_4 on the constraint graph?

(b) Assume we are doing backtracking with $assign(X_1 = 1, X_2 = 2)$. If the next chosen variable is X_4, what are the possible values X_4 that could be assigned?

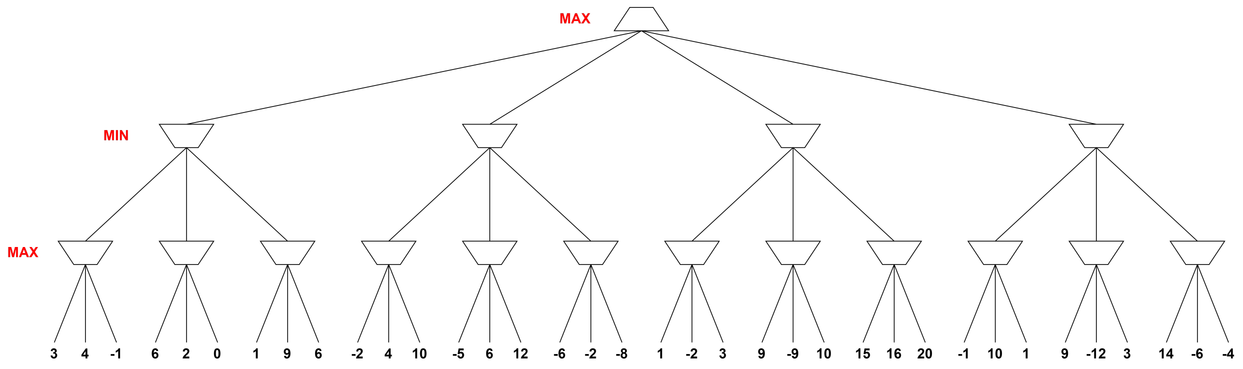
(c) Assume we are doing backtracking with forward checking and $assign(X_1 = 1, X_2 = 2)$, what are the reduced domains of all the variables?

(d) Assume we are given the reduced domain $D(X_3) = [2], D(X_5) = [1]$, the assignment is provided as $assign(X_3 = 2, X_5 = 1)$ as they are the only values these variables could take. If we apply the arc consistency on all arcs, what are the reduced domains of the other variables?

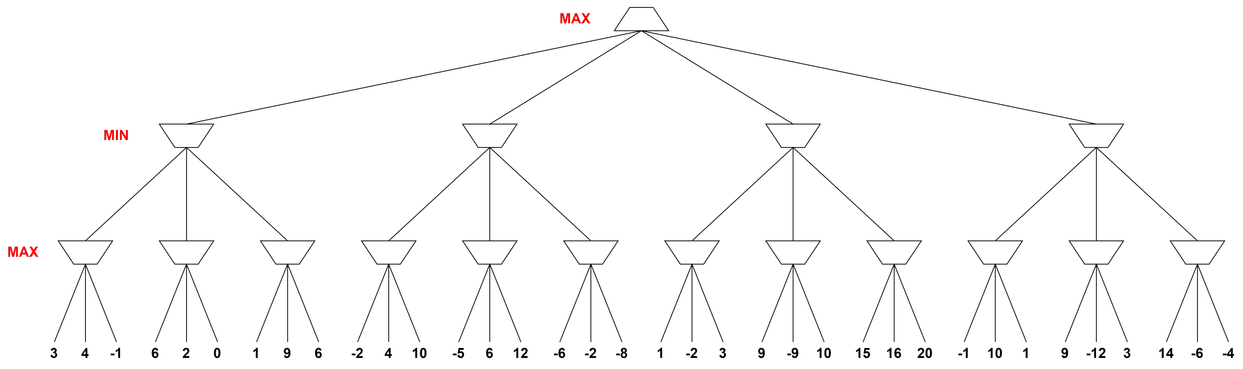
(e) Assume we are doing backtracking and the latest assignment is $assign(X_1 = 1, X_2 = 2, X_4 = 4, X_5 = 5)$, the next chosen variable is of course X_3, what should we do next (go back/backtrack or stop)? Why?

Problem 2: MinMax and Alpha-Beta Pruning [20 pts]

a. (10pts) Fill in each blank trapezoid with the proper minimax search value. Process the tree from left to right.



b. (10pts) Fill in each blank trapezoid and cross out the branches pruned by minimax + alpha-beta pruning. Process the tree from left to right. What are values of α and β?



Question 3: Pacman Finds All Food [40 pts]

In Assignment 2, we implemented searching algorithms that led Pacman to his food. In this assignment, we will lead Pacman to collect all his food. This assignment requires you to complete Assignment 2 because this will build upon that work.

`Filename`	**Purpose**
`search.py`	You will implement your algorithm in this file.
`searchAgents.py`	This file will be used to call your implemented algorithms.
`pacman.py`	This file is the main program that runs the Pacman game.
`util.py`	This file consists of useful data structures.
`game.py`	This file defines how the Pacman is played. It describes supporting types such as `AgentState`, `Agent`, `Direction`, and `Grid`.
`graphicsDisplay.py` `graphicsUtils.py` `textDisplay.py` `ghostDisplay.py` `keyboardAgents.py` `layout.py`	These files are used to render the Pacman game. You can ignore these files.

Acknowledgement: The base code is borrowed from the Introduction to AI course of the University of California, Berkeley.

Format of Search Problem Class

```
# search.py
class SearchProblem:

    def getStartState(self):
        util.raiseNotDefined()

    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        util.raiseNotDefined()
```

The SearchProblem class is an abstract class that provides four important instance methods:

- getStartState(self): returns the start state.

- isGoalState(self, state): returns true if the state is a destination.

- getSuccessors(self, state): return all successors of the given state. This function will a list of successors where each element is a tuple of next state, action, and cost.
- getCostOfActions(self, actions): returns the total cost when Pacman executes actions.

Since this is an abstract class, you are NOT allowed to modify or implement this class. In addition, you are free to define the "state" concept. It can be a position, or even any additional information that you think it's needed. You should read the PositionSearchProblem class in the searchAgent.py script to understand the example of a search problem and how to define four instance methods.

Finding All Corners [20 pts]

Now, you have already had searching algorithms implemented in the second assignment. In this section, you will implement a search problem, specifically, CornersProblem in the `searchAgent.py` script.

```
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
        "*** YOUR CODE HERE ***"
    def getStartState(self):
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
    def isGoalState(self, state):
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
    def getSuccessors(self, state):
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
        self._expanded += 1 # DO NOT CHANGE
        return successors
    def getCostOfActions(self, actions):
        "*** YOUR CODE HERE ***"
        util.raiseNotDefined()
```

The `init(self, startingGameState)` will initialize the information for the game, i.e., walls, a starting position, and four corners that contain food. You are free to define any additional information. The goal state is a state so that food at four corners has already been collected by Pacman. Now, your task is implementing four instance methods mentioned in Section 1. You can test your implementation by the command as follows.

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Corners Heuristic [15 points]

In the corners problem, the A* searching algorithm will illustrate its performance when we choose the correct heuristic. In this section, you will implement a `cornersHeuristic` function in the `searchAgent.py` script that returns the estimated cost from the current state to the final states.

```
def cornersHeuristic(state, problem):
    """
      state:   Your current search state
      problem: The CornersProblem instance for this layout.
    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem;
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py
    "***␣YOUR␣CODE␣HERE␣***"
    return 0 # Default to trivial solution
```

The grade of this section will be calculated based on how your optimal heuristic is. In particular, we will consider the number of nodes expanded:

More than 2000 nodes: 0%

From 1501 to 2000 nodes: 50%

At most 1500 nodes: 100%

You can test your heuristic by the command as follows.

```
python pacman.py -l mediumCorners -p SearchAgent -a \
    fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Note: It may take a few seconds (or a few mins) to run the algorithm, please be patient when you don't see your Pacman.

Eating All Food [5 points])

You have already led Pacman to collect his food in four corners. Now, we will do a similar problem but a harder version. In this section, we will let Pacman collect all his food in the maze. The food may be located everywhere in the maze. For this problem, we will use FoodSearchProblem already implemented in

Number of Nodes Expanded	**Grade**
More than 17000 nodes	25%
From 12001 to 17000 nodes	50%
At most 12000 nodes	100%
At most 7000 nodes	+10% (Bonus grade)

the `searchAgent.py` script. We will find a list of actions that leads Pacman to collect all his food. Your task is to implement a `foodHeuristic` function that returns a lower bound on the actual shortest path cost to the final state.

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    return 0
```

This function receives two arguments state and problem. State is a tuple including Pacman position and food grid (Grid data structure is in `game.py`). Problem is a `FoodSearchProblem` instance. Interestingly, if your solution in the previous section (your cornersHeuristic function) is an optimal and general solution. You can reuse corner heuristic for this food heuristic. Therefore, you try to implement your cornersHeuristic function as optimal as possible. The grade of this section will be calculated based on the number of nodes expanded.

You can test your heuristic by the command as follows.

```
python pacman.py -l trickySearch -p SearchAgent \
    -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

Note: It may take a few seconds (or a few mins) to run the algorithm, please be patient when you don't see your Pacman.