Middle East Technical University | Department of Computer Engineering

CENG 462

Artificial Intelligence

Fall '2021-2022

Homework 3

Due date: 11 December 2021, Saturday, 23:55

1 Objectives

This assignment aims to familiarize you with optimal decision-making in two-player games at the implementation level and provide hands-on experience with two algorithms (Minimax Algorithm, Alpha-Beta Search) for solving them.

2 Problem Definition

Different from the search operations that were considered in the previous two assignments, where there was only a single decision-maker entity, two-player games involve two decision-makers that contend for defeating each other. In order to be able to make an optimal decision, both parties have to take their opponent's decisions into account during the search operation. Furthermore, if each decision-making step contains any element of chance, probabilistic inference has to be incorporated into the decision process.

In this assignment, you are expected to implement the Minimax algorithm and the Alpha-Beta Search method in order to solve the tic-tac-toe game and other generic two-player games that are represented as trees.

2.1 Generic Two-player Game Trees

For this assignment, a two-player game tree represents all possible actions that can be taken by both opponents and the consequences of these actions in a level-by-level fashion. Each node represents a state of the game during the play. The root of the tree represents the opponent (max player or min player) that is to take and action at the current time step of the game. The next level represents the game state for the other opponent after the first opponent chooses an action. All possible actions are represented as labels on the arrows originating from a particular state, and determine what the next state of the game will be if taken. The leaf nodes of the tree are labeled with a numerical value which indicates a possible utility score that the opponent at the root node can earn. So if the opponent is a max-player in the root state of the game then she has to maximize its utility score with a gameplay strategy that takes her opponent's actions into account, similarly, she is expected to minimize it if she is the min player. Figure 1 depicts such a two-player game tree. If the opponent at the root is a max-player, all small red rectangles represent the possible game states of the max-player, similarly all purple rectangle nodes are the states for the other player (min player).

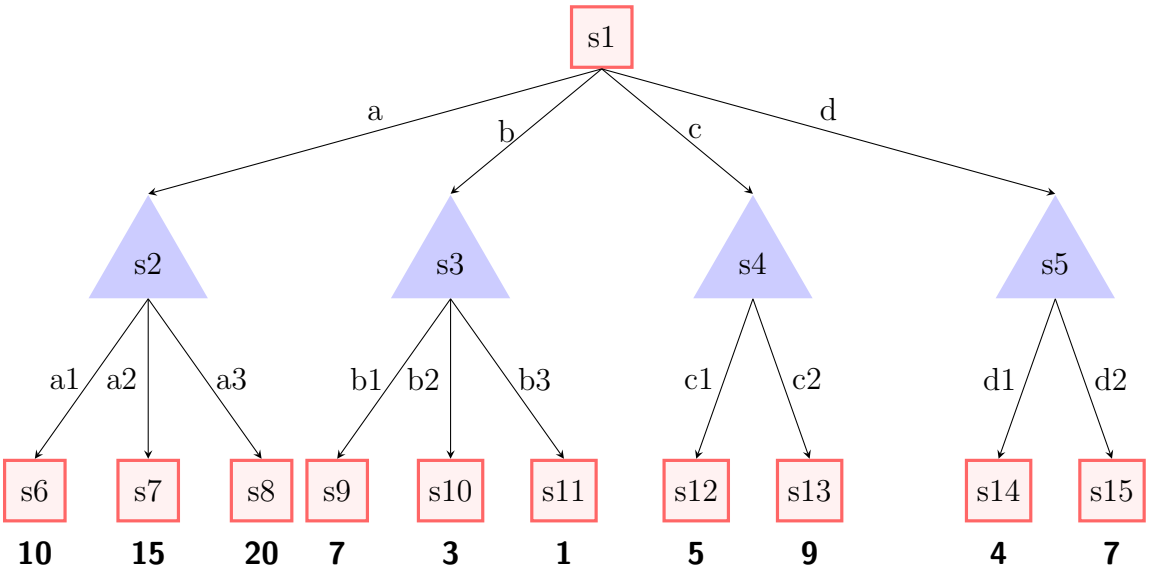

Figure 1: A sample generic game tree.

2.2 Tic-tac-toe Game

It is a very-well known two-player game where both opponents contend for marking one of the rows or columns or diagonal cells of the 3x3 game board completely with their corresponding symbols (`X`, `O`). The game is played in turns and each opponent may mark any available location (empty locations) with their symbol. It can be represented as a game tree that we have presented in the previous section as depicted in Figure 3. If we assume the opponent at the root state is the max player, the game should end with `X`'s winning. The terminal nodes (the nodes that the game can no longer progress, e.g. any player has won or they have tied.) have been labeled with the utility scores for the max-player. When a board configuration results in `X`'s victory its utility value is calculated with the following formula: $5 - 0.01 * (depth - 1)$, where *depth* is the state's depth value on the tree (the root node has a depth value of 0.). For a loss (`O` wins), it gets a utility value of -5 and in the case of a tie, it receives 0.

Actions of the game are represented with tuples of two numerical values which are the x-coordinate and y-coordinate values on the board, respectively. Figure 2 depicts which location each action affects on the board. For instance, If `X` player (max player) takes the action `(0,2)`, an `X` will be put to the left bottom of the board.

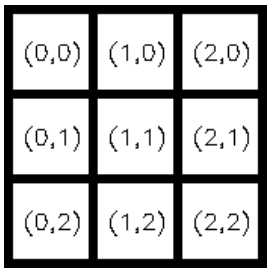

Figure 2: Actions are represented as coordinate values on the tic-tac-toe game board.

For the tic-tac-toe game, we assume the opponent who is at the root state is the max-player (whose marking symbol is X) (so she tries to maximize its utility value during the play).

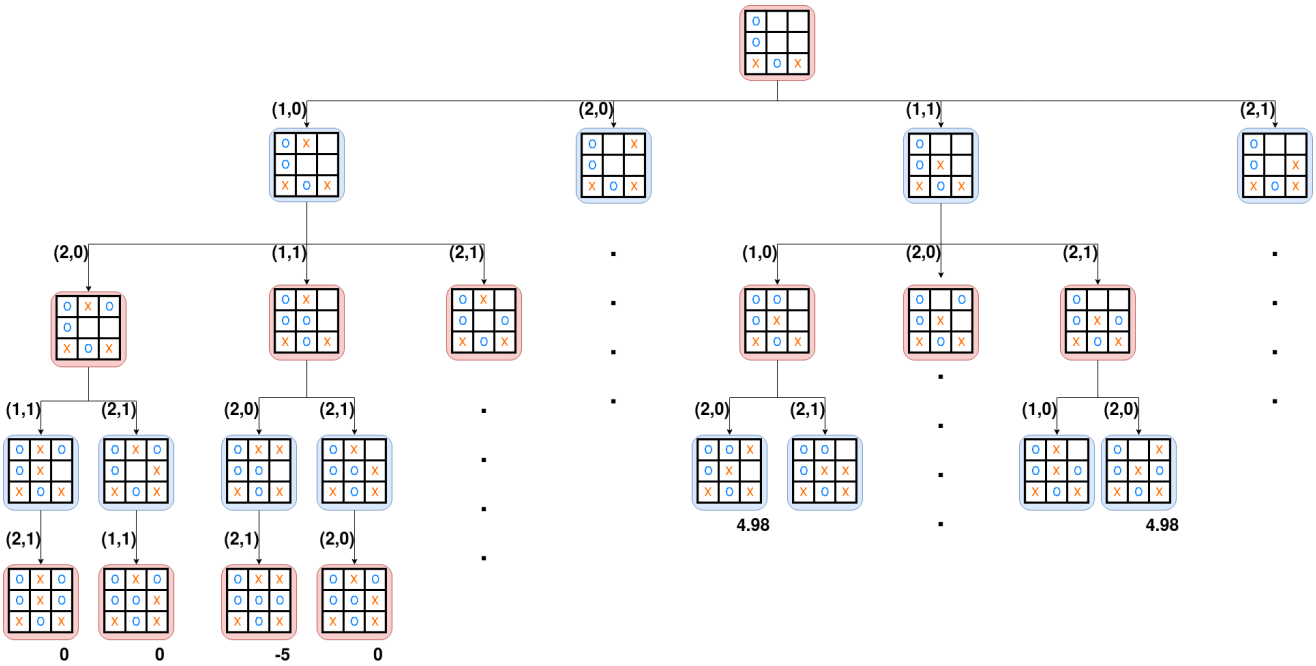

Figure 3: A sample game tree (partial) of the tic-tac-toe game. Actions are represented as arrows labeled with their corresponding coordinate values. Numbers underneath of the leaf nodes specify utility values for the max-player (`X`).

3 Specifications

- You are going to implement the Minimax algorithm and Alpha-Beta Search in Python 3.
- Each problem is represented with a file and this file will be fed to your implementation along with one of the methods.
- To be able to run all methods from a single place, you are expected to write the following function:

```
def SolveGame(method_name, problem_file_name, player_type):
```

  The parameter specifications are as follows:

  - `method_name`: specifies which method to be run for the given problem. It can be assigned to one of the values from `["Minimax", "AlphaBeta"]`.
  - `problem_file_name`: specifies the path of the problem file to be solved. File names are dependent on the problem and take values such as "gametree1.txt", "gametree2.txt", "tictactoe1.txt", "tictactoe2.txt".
  - `player_type`: specifies who is going to play in the game (whose turn it is now for the given game state, the player at the root state) (max player or min player) and takes the value of either "MAX" or "MIN".

  During the evaluation process, this function will be called and returned information will be inspected. The returned information is dependent on the method fed and as follows:

  - `method_name=="Minimax"`: `GameSolve` returns the optimal action value, the optimal action, and the list of visited tree nodes during the search.
  - `method_name=="AlphaBeta"`: The same output structure as Minimax should be generated.

- Generic game trees are described in ".txt" files and have the following structure:

```
<root node>
<node> <node> action
<node> <node> action
<node> <node> action
.
.
.
<leaf node>:integer value
<leaf node>:integer value
.
.
.
```

The first line keeps the information of the root node that specifies where the current player is at in the gameplay. The next lines describe the parent-child node relation with a given action label (actions are represented with directed edges originating from the parent nodes). The lines that contain the ":" character specify the numerical value that a particular leaf node has (as a utility value).

- For a particular configuration/state of the tic-tac-toe game is provided in ".txt" files too and has the following structure:

```
abc
efg
hij
```

where a, b, c, d, e, f, g, h, i, j ∈ {'X', 'O', ' '}.

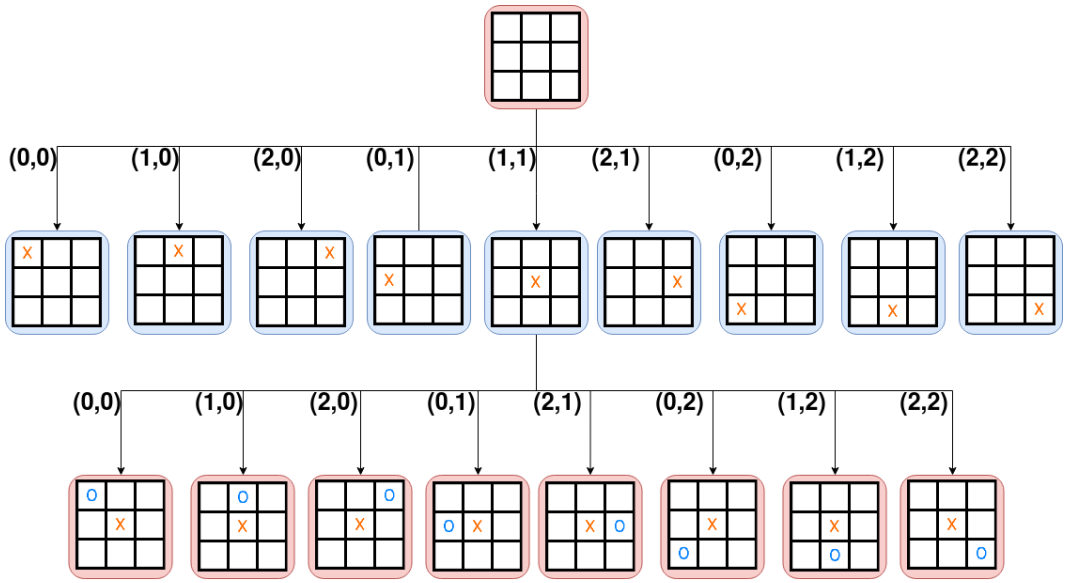

Figure 4: A sample child expansion on a tic-tac-toe game tree. The nodes should be expanded in the following action sequence: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0, 2), (1, 2), (2, 2). If the place targeted by an action on the game state is not available then that action is skipped.

- For the tic-tac-toe game, while expanding a node (children construction), please follow the following action sequence: (0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0, 2), (1, 2), (2, 2). That is to say, the first child of the parent node should be the one that is the result of the application of the action (0,0), the second one is the result of the action (1,0), and so forth.
- For the tic-tac-toe games only the max-player (`X`, `player_type="MAX"`) is going to be tested.

- No `import` statement is allowed and all methods should be implemented in a single solution file. You are expected to implement all the necessary operations by yourself without utilizing any external library.
- Commenting is crucial for understanding your implementation and decisions made during the process. Your implementation can be manually inspected at random or when it does not satisfy the expected outputs.

4 Sample I/O

gametree1.txt:

```
s1
s1 s2 a
s1 s3 b
s1 s4 c
s1 s5 d
s2 s6 a1
s2 s7 a2
s2 s8 a3
s3 s9 b1
s3 s10 b2
s3 s11 b3
s4 s12 c1
s4 s13 c2
s5 s14 d1
s5 s15 d2
s6:10
s7:15
s8:20
s9:7
s10:3
s11:1
s12:5
s13:9
s14:4
s15:7
```

Expected outputs of the GameSolve function with different methods applied on gametree1.txt:

```
>>> import hw3solutioneXXXXXXX as hw3
>>> print(hw3.SolveGame("Minimax", "gametree1.txt", "MAX"))
(10, 'a', ['s2', 's6', 's7', 's8', 's3', 's9', 's10', 's11', 's4', 's12', 's13
    ', 's5', 's14', 's15'])
>>> print(hw3.SolveGame("Minimax", "gametree1.txt", "MIN"))
(7, 'b', ['s2', 's6', 's7', 's8', 's3', 's9', 's10', 's11', 's4', 's12', 's13',
     's5', 's14', 's15'])
>>> print(hw3.SolveGame("AlphaBeta", "gametree1.txt", "MAX"))
(10, 'a', ['s2', 's6', 's7', 's8', 's3', 's9', 's4', 's12', 's5', 's14'])
>>> print(hw3.SolveGame("AlphaBeta", "gametree1.txt", "MIN"))
(7, 'b', ['s2', 's6', 's7', 's8', 's3', 's9', 's10', 's11', 's4', 's12', 's13',
     's5', 's14', 's15'])
```

gametree1.txt is the problem definition file of Figure 1. Depending on the player type (max or min) the utility-maximizing or minimizing actions are considered. The first number returned specifies the utility

value for the max or min player. The second item is the action picked by the max or min player. The third element is the list of visited nodes during the search (the root state is not included).

Note 1: The generic game trees may be of any size (i.e. they may have 4, 5, 10, etc. depth levels, not only 3 levels as Figure 1 has).
Note 2: There could be more one than one action that results in the same utility value, any of them can be returned as the optimal action.).

tictactoe1.txt:

```
 OX
O O
X
```

Expected outputs of GameSolve function with different methods applied on tictactoe1.txt:

```
>>> import hw3solutioneXXXXXXX as hw3
>>> print(hw3.SolveGame("Minimax", "tictactoe1.txt", "MAX"))
(5.0, (1, 1), ['XOXO OX  ', 'XOXOOOX  ', 'XOXO OXO ', 'XOXOXOXO ', 'XOXO OXOX',
    'XOXOOOXOX', 'XOXO OX O', 'XOXOXOX O', 'XOXO OXXO', 'XOXOOOXXO', ' OXOXOX
   ', ' OXO OXX ', 'OOXO OXX ', 'OOXOXOXX ', 'OOXO OXXX', ' OXOOOXX ', ' OXO
   OXXO', 'XOXO OXXO', 'XOXOOOXXO', ' OXOXOXXO', ' OXO OX X', 'OOXO OX X', '
   OOXOXOX X', 'OOXO OXXX', ' OXOOOX X', ' OXO OXOX', 'XOXO OXOX', 'XOXOOOXOX',
    ' OXOXOXOX'])
>>> print(hw3.SolveGame("AlphaBeta", "tictactoe1.txt", "MAX"))
(5.0, (1, 1), ['XOXO OX  ', 'XOXOOOX  ', 'XOXO OXO ', 'XOXOXOXO ', 'XOXO OX O',
    'XOXOXOX O', ' OXOXOX  ', ' OXO OXX ', 'OOXO OXX ', 'OOXOXOXX ', 'OOXO OXXX
   ', ' OXO OX X', 'OOXO OX X', 'OOXOXOX X', 'OOXO OXXX'])
```

tictactoe2.txt:

```
O O
 X
O X
```

Expected outputs of GameSolve function with different methods applied on tictactoe2.txt:

```
>>> print(hw3.SolveGame("Minimax", "tictactoe2.txt", "MAX"))
(-5, (1, 0), ['OXO X O X', 'OXOOX O X', 'OXO XOO X', 'OXOXXOO X', 'OXOXXOOOX',
   'OXO XOOXX', 'OXO X OOX', 'OXOXX OOX', 'OXOXXOOOX', 'OXO XXOOX', 'OXOOXXOOX
   ', 'O OXX O X', 'OOOXX O X', 'O OXXOO X', 'OXOXXOO X', 'OXOXXOOOX', 'O
   OXXOOXX', 'OOOXXOOXX', 'O OXX OOX', 'OXOXX OOX', 'OXOXXOOOX', 'O OXXXOOX', '
   O O XXO X', 'OOO XXO X', 'O OOXXO X', 'O O XXOOX', 'OXO XXOOX', 'OXOOXXOOX',
    'O OXXXOOX', 'O O X OXX', 'OOO X OXX', 'O OOX OXX', 'O O XOOXX', 'OXO XOOXX
   ', 'O OXXOOXX', 'OOOXXOOXX'])
>>> print(hw3.SolveGame("AlphaBeta", "tictactoe2.txt", "MAX"))
(-5, (1, 0), ['OXO X O X', 'OXOOX O X', 'OXO XOO X', 'OXOXXOO X', 'OXOXXOOOX',
   'OXO X OOX', 'OXOXX OOX', 'OXOXXOOOX', 'O OXX O X', 'OOOXX O X', 'O O XXO X
   ', 'OOO XXO X', 'O O X OXX', 'OOO X OXX']
```

tictactoe3.txt:

```
OXO
 OX
 OX
```

Expected outputs of GameSolve function with different methods applied on tictactoe3.txt:

```
>>> print(hw3.SolveGame("Minimax", "tictactoe3.txt", "MAX"))
(0, (0, 2), ['OXOXOX OX', 'OXOXOXOOX', 'OXO OXXOX', 'OXOOOXXOX'])
>>> print(hw3.SolveGame("AlphaBeta", "tictactoe3.txt", "MAX"))
(0, (0, 2), ['OXOXOX OX', 'OXOXOXOOX', 'OXO OXXOX', 'OXOOOXXOX'])
```

tictactoe4.txt:

```
XOX
  O
```

Expected outputs of GameSolve function with different methods applied on tictactoe4.txt:

```
>>> print(hw3.SolveGame("Minimax", "tictactoe4.txt", "MAX"))
(4.98, (1, 1), ['XOXX O   ', 'XOXXOO   ', ...])
>>> print(hw3.SolveGame("AlphaBeta", "tictactoe4.txt", "MAX"))
(4.98, (1, 1), ['XOXX O   ', 'XOXXOO   ', ...])
```

For the tic-tac-toe game, only the max player (`X`) is going to be tested. GameSolve returns the optimal action value considered by the max player and its utility value. In addition, it returns the visited nodes during the search on the game tree. In the given sample outputs, the visited nodes are represented with 9-character strings, which basically corresponds to the game state in that particular node. The first 3 characters, the next 3 characters, and the last 3 characters depict the first, second, and third rows of a state, respectively.

Note 1: There could be more one than one action that results in the same utility value, any of them can be returned as the picked action.).

Note 2: The problem files and their outputs here are provided as additional files on ODTUClass.

5 Regulations

1. **Programming Language:** You must code your program in Python 3. Since there are multiple versions and each new version adds a new feature to the language, in order to concur on a specific version, please make sure that your implementation runs on İnek machines.

2. **Implementation:** You have to code your program by only using the functions in the standard module of python. Namely, you **cannot** import any module in your program.

3. **Late Submission:** No late submission is allowed. Since we have a strict policy on submissions of homework in order to be able to attend the final exam, please pay close attention to the deadlines.

4. **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating (any kind of code sharing and codes taken from the internet included) will be punished according to the university regulations.

5. **Discussion:** You must follow ODTUClass for discussions and possible updates on a daily basis. If think that your question concerns everyone, please ask them on ODTUClass.

6. **Evaluation:** Your program will be evaluated automatically using the "black-box" technique so make sure to obey the specifications. A reasonable timeout will be applied according to the complexity of test cases. This is not about the code efficiency, its only purpose is avoiding infinite loops due to an erroneous code. In case your implementation does not conform to expected outputs for solutions, it will be inspected manually so commenting will be crucial for grading.

6 Submission

Submission will be done via OdtuClass system. You should upload a **single** python file named in the format **hw3_e<your-student-id>.py** (i.e. hw3_e1234567.py).

7 References

- For `Minimax Algorithm` and `Alpha-Beta Search`, you can refer to the lecture's textbook.
- Announcements Page
- Discussions Page