Johns Hopkins University
Whiting School of Engineering
Engineering for Professionals
685.621 Algorithms for Data Science

Tic-Tac-Toe Game Framework - Algorithm Description Document (ADD)

Contents

1 Introduction

In this algorithm description document the game of Tic-Tac-Toe is described from a software development perspective using Python. The provided environment contains two run files tic_tac_toe.py and experiment.py, the three classes are game.py, board.py, and player.py. These files give the ability to play the game of Tic-Tac-Toe. The document is expanded with the use of various algorithm descriptions in which the term *agents* is used to describe the artificial player of the game [5]. In the game of Tic-Tac-Toe there are set know rules to help play the game when two-players sit at a table to play on a piece of paper, https://en.wikipedia.org/wiki/Tic-tac-toe. The rules of Tic-Tac-Toe have been set in his framework to ensure the user only has to focus on the development and implementation of algorithms for advancest agents.

1.1 Process Flow

The process flow should consider at least three components.

1. The algorithm design should allow flexibility for reworking the algorithm(s) as well as the software implementation.
2. The decisions of the algorithm(s) should be described in a manner that allows the developers to gain insight into the reasoning of the development process.
3. The process flow description should provide the a clear understanding of the data inputs and the data outputs (results).

1.2 Software Development Requirements

One of the most important aspects to keep in mind while developing softwares is readability, which often means following well-known coding conventions and structures. Consistency is especially important in group projects for efficient communication and collaboration. Throughout this document, we will refer to the PEP 8 Python Style Guide.

1.2.1 Indentation

Always use four spaces per indentation level.

```
//game.py
class Game:

    # Initializes the game of tic-tac-toe
    def __init__(self, player1: Player, player2: Player):
        self.board = Board()
        self.player1 = player1
        self.player2 = player2
        self.players = [player1, player2]
```

1.2.2 Comments

Block comments apply to all the code that follow them and are at the same indentation level. All comments begin with a # and single space and should contain thorough explanations of the code below. Multi-line comments should follow the same conventions.

```
// game.py
# Plays the game and returns the results
# A "1" means player 1 wins, "2" means player 2 wins; "0" means a draw
```

```python
def play(self) -> int:

        # Display the game start messages and a grid of numbered space indexes
        print("Starting a new game of tic-tac-toe.")
```

1.2.3 Naming Conventions

Class names should follow the CapWords convention, meaning that the first letter of each word in the class name should be capitalized.

```python
//minimax_player.py
# Represents a brute-force minimax agent
class MinimaxPlayer(Player):
```

Since exceptions are classes, the CapWords convention applies here too. You should only use the "Error" suffix if the exception is truly an error.

Function and variable names should be all lowercase with words separated by underscores. The first argument to a class method should be *cls* and the first argument to an instance method should be *self*. If any of the function arguments' names clashes with an existing python keyword, it is best to append a trailing "_" instead of truncating or abbreviating the spelling. For example instead of abbreviating *class* to *clss*, use *class_* .

```python
//board.py
def copy(self):
        state = str().join(self.spaces)
        new_board = Board(state)
        return new_board
```

Constants are written in all capital letters with underscores to separate words.

1.3 Software Development Challenges

The software development challenges for the overall framework have been addressed. The next set of challenges are with new agents that are to be developed. It is essential to use the `[class]_test.py` classes to be updated for edge cases to ensure the correct development has been accomplished.

2 Algorithm Description

2.1 Algorithm Input(s)

The program has a very simple console-based user interface. It starts by presenting the user with the index of each state of the board and asks them to place their mark using the numbers [0-8] corresponding to the numeric index of each space on the board.

```
Starting a new game of tic-tac-toe.
Spaces are numbered as follows:
0 1 2
3 4 5
6 7 8
Player 1, place your mark [0-8]:
```

When the user chooses a space to place their mark, the UI updates the game board with their selection. Next, the opponent plays their move and the UI updates again with the state of the board after the opponents move. Then, the UI prompts the user for their next move.

```
Player 1, place your mark [0-8]: 0
Player 1 chooses space 0.
X - -
- - -
- - -
Player 2 chooses space 4.
X - -
- O -
- - -

Player 1, place your mark [0-8]:
```

Invalid moves are prevented by the system and a warning message is displayed for the user

```
Player 1, place your mark [0-8]: 0
0 is an invalid move.
Player 1, place your mark [0-8]:
```

Finally, when the game is finished, the UI displays the final state of the game board and which player won – or a draw if there was no winner. The UI then asks the player if they would like to play another game, to which they can respond either “Y” or “N” for “Yes” or “No” respectively.

```
Player 1 chooses space 2.
X O X
- O X
- - -

Player 2 chooses space 7.
X O X
- O X
- O -

Player 2 wins!

Play another game? Y/N:
```

2.2 Environments

- **TicTacToe** (*Run File*) – the main program to play the game of tic-tac-toe against an agent.
- **Experiment** (*Run File*) – a program to run n trials (i.e. games) with automated agents – for analysis.
- **Game** (*Class*) – represents a single game of tic-tac-toe between two players.
- **Board** (*Class*) – represents a tic-tac-toe game board with spaces indexed [0-8] in row-wise fashion.
- **Player** (*Abstract Class*) – represents an abstract tic-tac-toe player to be implemented by a subclass below.

2.3 Algorithm Descriptions of Agents

- **Human Player** (*Class*) – represents a human player who is prompted for input via the console
- **Sequence Player** (*Class*) – represents a player who plays a pre-defined sequence of moves (for testing)
- **Random Player** (*Class*) – represents an automated player that choses moves at random
- **Conditional Player** (*Class*) – represents an automated agent who uses conditional logic

Algorithm 1 Conditional Player

```
 1: class Conditional Player (Player)
 2: function GET-NEXT-MOVE returns int
 3:     move= getDecisiveMove()           ▷ Get a decisive (immediate win or loss) move if one exists
 4:     if move exists then
 5:         return move
 6:     else
 7:         return getNonDecisiveMove()
 8:     end if
 9: end function

10: function GET-DECISIVE-MOVE(board) returns int
11:     if space 0 has decisive move then   ▷ Check each diagonal/row/column for possible win/loss
12:         return 0
13:     end if
14:     repeat for spaces [1-8]
15:     if no decisive move exists then
16:         return 0
17:     end if
18: end function

19: function GET-NON-DECISIVE-MOVE(board) returns int
20:     Use simple heuristic logic to choose generally good positions
21: end function
```

- **Utility Player** (*Class*) – represents an automated agent who uses a utility function to evaluate moves. A modified version of the Utility-Based-Agent from [5] is shown in Algorithm 2. In this algorithm a recommended evaluation function $Eval = 3X_2 + X_1 - (3O_2 + O_1)$ to measure the performance of each location on a board at the current state of the board. The variable X_n is defined as the number of n $X's$ on a given row, column or diagonal. The variables O's are defined as the number of n $O's$ on a given row, column or diagonal. For this algorithm let $n = 1, 2$ since $n = 3$ will result in a win. Now, consider a unity function that assigns 1 to any position when the X_n is met and 1 to any position when the O_n is met. Any other row column or diagonal that does not meet the above definition are not evaluated or receive a 0.

Algorithm 2 Utility-Based-Agent from [5]

```
1: class Utility Player (Conditional Player)
2: function Utility-Based-Agent(percept) returns an action
3:     procedure Persistent:(descriptions)
4:         state, ...
5:         plan, ...
6:         action, ...
7:         replan, ...
8:         action, ...
9:     end procedure
10:    state ← Read in the current state of the board.
11:    plan, ... Evaluate each board location.
12:    action, ... If one location is identified with the highest evaluation returned value assign that
       location for the action to take. If more than one location has the highest returned value based on
       the evaluation function go to replan.
13:    replan, ... If more than one location exists, use a secondary evaluation to determine the best
       location form the subset with the highest returned values.
14:    actionbasedonreplan, ... If the replan is executed take the action from replan.
15:    return action
16: end function
```

Based the the game frame work the Algorithm 2 is updated as shown in Algorithm 3 to ensure compatibility with the code framework. Additionally, the proposed evaluation function is not used instead two functions are used Get-Utility-Of-Spaces and Get-Line-Utility to determine the best open space with the highest utility.

Algorithm 3 Utility Player

```
class Utility Player (Conditional Player)
function Get-Next-Move(board) returns int
    move= getDecisiveMove()                                   ▷ Method inherited from Conditional Player
    if move exists then
        return move
    else
        return getNonDecisiveMove()
    end if
    Lines = getLineUtility(board)                  ▷ utility of all horizontal, vertical, diagonal lines
    Spaces = getUtilityOfSpaces(board, Lines)            ▷ utility of each space as possible moves
    bestMove = max(Spaces)                          ▷ choose best move with space of highest utility
    return bestMove
end function

function Get-Utility-Of-Spaces(board) returns array
    LineUtilities = []
    for line in board do
        if line is empty then
            utility = 0
        else if line is full then
            utility = −10
        else
            utility = getLineUtility(board, line)
        end if
        LineUtilities.append(utility)
    end for
    return LineUtilities                                          ▷ array of utilities for each line
end function

function Get-Line-Utility(board) returns int
    agentMarks = 0
    opponentMarks = 0
    for space in line do
        if board[space] == selfMark then
            agentMarks += 1
        else if board[space] == opponentMark then
            opponentMarks += 1
        end if
    end for
    lineUtility = 3*agentMarks - opponentMarks      ▷ utility based on agents & opponents marks
    return lineUtility
end function
```

- **Min-Max Player** (*Class*) – represents an automated agent who uses the brute-force minimax algorithm. The algorithm shown in Algorithm 4 provides the initial requirements for developing the minimax search algorithm into the provided framework. This is an algorithm for calculating the optimal move using MiniMax - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions Max-Value and Min-Value go through the entire game tree, all the way to the leaf nodes, to determine the backed-up value of a state and the move to get there.

Algorithm 4 Original MiniMax Search Algorithm from [5]

```
class Min-Max-Search (Player)
function Min-max-Search(game, board) returns an action
    player ← game.To-Move(state)
    value, move ← Max-Value(game, state)
    return move
end function

function Max-Value(game, state) returns a (utility, move) pair
    if game.Is-Terminal(state) then return (board.Utility(state, player), null)
        v ← $-\infty$
    end if
    for a in game.Actions(state) do
        v2, a2 ← MinValue(game, game.RESULT(state, a))
        if $v2 > v$ then
            v, move ← v2, a
        end if
    end for
    return v, move
end function

function Min-Value(game, state) returns a (utility, move) pair
    if game.IsTerminal(state) then return (game.Utility(state, player), null)
        v ← $+\infty$
    end if
    for a in game.Actions(state) do
        v2, a2 ← Max-Value(game, game.RESULT(state, a))
        if $v2 < v$ then
            v, move ← v2, a
        end if
    end for
    return v, move
end function
```

Based the the game frame work the Algorithm 4 is updated as shown in Algorithm 5 to ensure compatibility with the code framework.

Algorithm 5 MiniMax Player

```
class MiniMax Player ($Player$)
function GET-NEXT-MOVE($board$) returns $int$
    $player$ = $board$.ToMove($selfMark$)
    $value$, $move$ = MaxValue($board$, $selfMark$)
    return $move$
end function

function MAX-VALUE($board$, $selfMark$) returns a (utility, move) pair
    if $board$.IsTerminal($selfMark$) then return  ($board$.Utility($selfMark$, $player$), null)
        $v$ = -$\infty$
    end if
    for $a$ in $board$.Actions($selfMark$) do
        $v2$, $a2$ = MinValue($board$, $board$.RESULT($selfMark$, $a$))
        if $v2 > v$ then
            $v$, $move$ = $v2$, $a$
        end if
    end for
    return $v$, $move$
end function

function MIN-VALUE($board$, $selfMark$) returns a (utility, move) pair
    if $board$.IsTerminal($selfMark$) then return  ($board$.Utility($selfMark$, $player$), null)
        $v = +\infty$
    end if
    for $a$ in $board$.Actions($selfMark$) do
        $v2$, $a2$ = MaxValue($board$, $board$.RESULT($selfMark$, $a$))
        if $v2 < v$ then
            $v$, $move$ = $v2$, $a$
        end if
    end for
    return $v$, $move$
end function
```

- **Alpha-Beta Player** (*Class*)– represents an automated agent who uses Min-Max with Alpha-Beta pruning. In the Alpha-Beta search algorithm, it should be noticed that these functions are the same as the Min-Max-Search functions in Algorithm 4, except that we maintain bounds in the variables α and β, and use then to cut off search when a value is outside the bounds. The algorithm shown in Algorithm 6 provides the initial requirements for developing the Alpha-Beta search algorithm into the provided framework.

Algorithm 6 Original Alpha-Beta Search Algorithm from [5]

```
class Alpha-Beta-Decision (Player)
function ALPHA-BETA-DECISION(game, board) returns an action
    player ← game.To-Move(state)
    value, move ← Max-Value(game, state, $-\infty$, $+\infty$)
    return move
end function

function MAX-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
    if game.Is-Terminal(state) then return (board.Utility(state, player), null)
        v ← $-\infty$
    end if
    for a in game.Actions(state) do
        v2, a2 ← Min-Value(game, game.RESULT(state, a), $\alpha$, $\beta$)
        if $v2 > v$ then
            v, move ← v2, a
            $\alpha$ ← Max($\alpha$, v)
        end if
        if $v2 \geq \beta$ then
            return v, move
        end if
    end for
    return v, move
end function

function MIN-VALUE(game, state, $\alpha$, $\beta$) returns a (utility, move) pair
    if game.IsTerminal(state) then return (game.Utility(state, player), null)
        v ← $+\infty$
    end if
    for a in game.Actions(state) do
        v2, a2 ← Max-Value(game, game.RESULT(state, a))
        if $v2 < v$ then
            v, move ← v2, a
            $\beta$ ← Min($\beta$, v), $\alpha$, $\beta$
        end if
        if $v2 \leq \alpha$ then
            return v, move
        end if
    end for
    return v, move
end function
```

Based the the game frame work the Algorithm 6 is updated as shown in Algorithm 7 to ensure compatibility with the code framework.

Algorithm 7 AlphaBeta Player

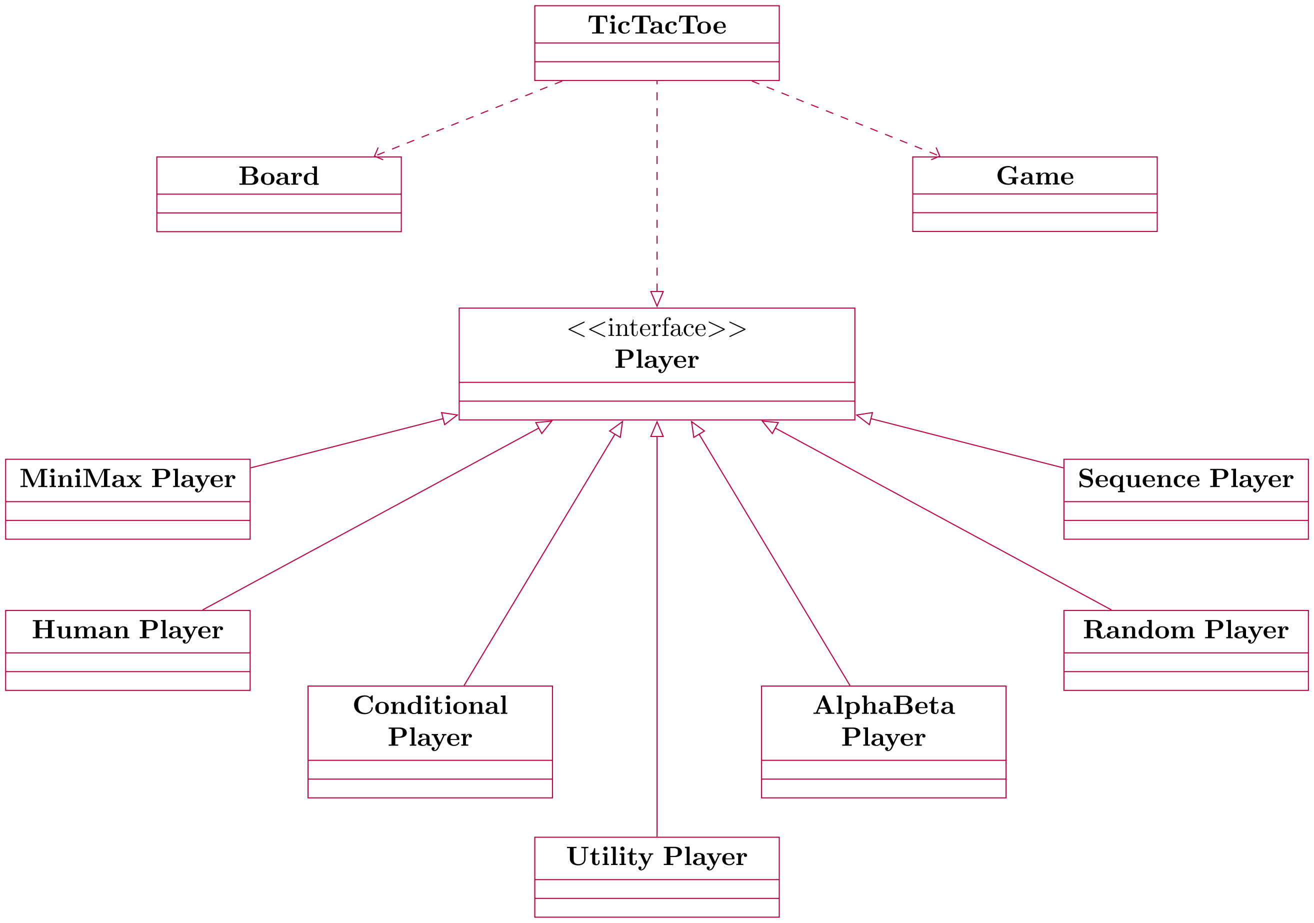
```
class AalphaBeta Player ($Player$)
function GET-NEXT-MOVE($board$) returns $int$
    $player$ = $board$.ToMove($selfMark$)
    $value$, $move$ = MaxValue($board$, $selfMark$, $-\infty$, $+\infty$)
    return $move$
end function

function MAX-VALUE($board$, $selfMark$, $\alpha$, $\beta$) returns a (utility, move) pair
    if $board$.IsTerminal($selfMark$) then return ($board$.Utility($selfMark$, $player$), null)
        $v$ = -$\infty$
    end if
    for $a$ in $board$.Actions($selfMark$) do
        $v2$, $a2$ = MinValue($board$, $board$.RESULT($selfMark$, $a$), $\alpha$, $\beta$)
        if $v2 > v$ then
            $v$, $move$ = $v2$, $a$
        end if
        if $v2 \geq \beta$ then
            return $v$, $move$
        end if
    end for
    return $v$, $move$
end function

function MIN-VALUE($board$, $selfMark$, $\alpha$, $\beta$) returns a (utility, move) pair
    if $board$.IsTerminal($selfMark$) then return ($board$.Utility($selfMark$, $player$), null)
        $v$ = $+\infty$
    end if
    for $a$ in $board$.Actions($selfMark$) do
        $v2$, $a2$ = MaxValue($board$, $board$.RESULT($selfMark$, $a$), $\alpha$, $\beta$)
        if $v2 < v$ then
            $v$, $move$ = $v2$, $a$
        end if
        if $v2 \leq \alpha$ then
            return $v$, $move$
        end if
    end for
    return $v$, $move$
end function
```

2.4 Algorithm Output(s)

Each implemented Player Class returns a move based on the game's state (X or O) and game board. The returned move is in the format of an index, corresponding to a spot on the board. The returned move should be returned for a valid open position assuming the game has not ended.

2.5 Algorithm Implementation

Implement each agent by completing their GET-NEXT-MOVE methods, inherited from the abstract Player Class. Following, is a chart illustrating the relations between different agents and environments:



3 Unit Testing

In his code framework a series of unit tests to verify the correct behavior of each class has been set up. These unit tests covered a large % of the lines of code contained in files that were testable. The files `tic_tac_toe.py`, `experiment.py`, and `human_player.py` are not testable because they are interactive user interfaces. For each of the player classes a set of unit tests for the corresponding class `[class]_tests.py`. A `board_test.py` is provided to ensure the functionality of the board is correct, such as, an empty board, full board, open spaces, placing the correct piece on the board, a win state, copy of the board, and a grid of the board. The `game_test.py` test if a win for X, O or a tie has been reached. These unit tests can be expanded to ensure various cases are accounted for in other framework development. The `argmax_test.py` tests the return of the `argmax.py` returns the correct values from the current game board, the `argmax.py` is used by the `minimax_player.py` and others.

The `experiment.py` is a program to run a specified number of games with automated players for analysis. This file is also used to output the run time of a specified number of games that two agents play against each other. Caution should be taken in that this function is dependent on the specific machine the code is being run on.

4 Conclusion

In this algorithm description document the game of Tic-Tac-Toe has been described from an algorithms perspective. The description document covers the process flow to ensure a clear understanding of the framework has been presented. The software development requirements is give for proper coding standards. The software development challenges are provided and will continue to be updated as new functionality is provided is this game framework. The bulk of this document is give for the algorithm descriptions for the various files within this framework. The descriptions were provided for initial algorithms (pseudocode) and in certain algorithms some modifications were presented to ensure the algorithms conformed with provided framework. This description documentation ends with the unit test provided in this framework and uses.

References

[1] Ken Jensen, Donna McNamara, and Thomas King, *ALGORITHM THEORETICAL BASIS DOCUMENT TEMPLATE VERSION 2.2*, NOAA, 2012, Retrieved June 12, 2019, https://www.star.nesdis.noaa.gov/jpss/documents/Templates/SPSRB_ATBD_Template.docx

[2] Python enhancement proposals, *PEP 8 – Style Guide for Python Code*, (n.d.), Retrieved August 17, 2022, from https://peps.python.org/pep-0008

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Fourth Edition, The MIT Press., 2022

[4] Thomas H. Cormen, The clrscode and clrscode3e packages for LaTeX2e, Retrieved Jan 2010, http://www.cs.dartmouth.edu/ thc/clrscode/

[5] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th Edittion, Pearson Series in Artificial Intelligence, 2022

[6] Toni Santo-Regis, *LaTeX A document preparation system*, Retrived Jan 2010, http://www.latex-project.org/