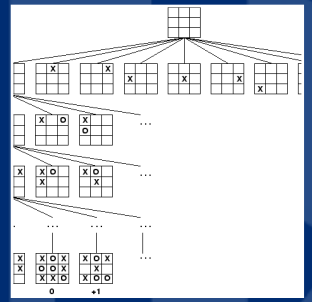
JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

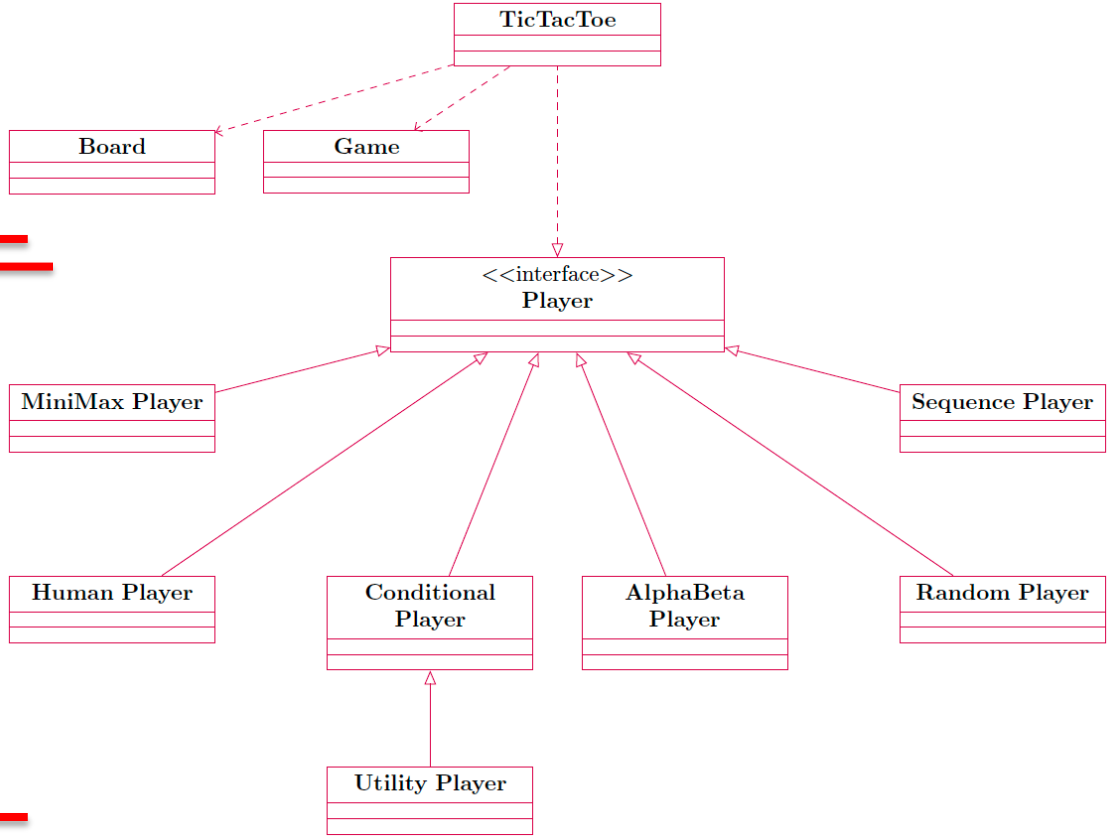
Tic-Tac-Toe an Introduction

Ben Rodriguez
Shruti Shah



Outline for Algorithm Implementation

- Environments
  - Classes
  - Run File
- Agents



Environments: *Tic-Tac-Toe (Run File)*

User Interface:

1

```
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]:
```

Prompts user to place token on board

2

```
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]:
```

Once user inputs position, the new board is displayed and the opposite agent makes their move.

Environments: *Tic-Tac-Toe (Run File) cont.*

User Interface:

3

```
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:
```

Once game ends, user is able to clear the board and begin a new game with the same agent.

Environments: *Tic-Tac-Toe (Run File) cont.*

Testing Different Agents:

```
# Set the players for the
game
# Note: Change these players
to test different agents
player1 = HumanPlayer(1)
player2 = AlphaBetaPlayer(2)
```

Reassign players here to have other agents compete against each other. Note: If HumanPlayer() is not one of the agents, there is no user input required.

Environments: *Board (Class)*

Representation:

```
def __init__(self, state="---------"):
        self.empty = "-"
        self.spaces = list(state)
```

The board is a nine character string populated by the tokens “X”, “O”, or “-”.

```
self.lines = (
            (0, 1, 2),
            (3, 4, 5),
            (6, 7, 8),
            (0, 3, 6),
            (1, 4, 7),
            (2, 5, 8),
            (0, 4, 8),
            (2, 4, 6))
```

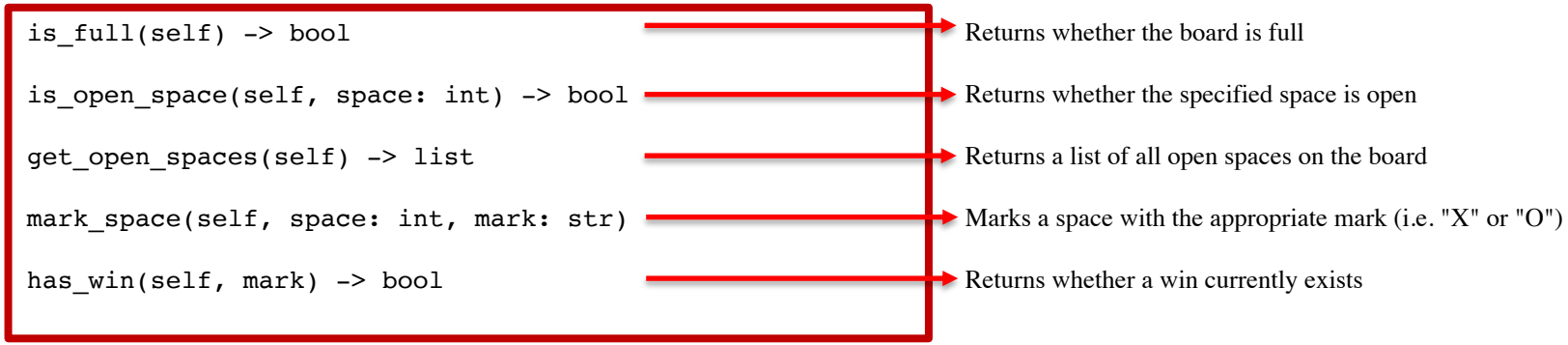
Each tuple represents a potential combination of indexes on the board which may result in a win.

Note: The board is numbered in this way

0 1 2
3 4 5
6 7 8

Environments: *Board (Class)*

Methods:



Environments: *Game (Class)*

Representation:

```
def __init__(self, player1: Player,
player2: Player):
        self.board = Board()
        self.player1 = player1
        self.player2 = player2
        self.players = [player1,
player2]
```

Initialize Game by instantiating both players and board.

Environments: *Game (Class)*

Playing a Game:

```
def play(self) -> int:
        print("Starting a new game of tic-tac-toe.")
        print("Spaces are numbered as follows:")
        print(self.board.get_space_indexes())
        turn = 1
        while True:
            for player in self.players:
                move = player.get_next_move(self.board)
                self.board.mark_space(move, player.mark)
                print(f"Player {player.number} chooses space {move}.")
                print(str(self.board))
                if self.board.has_win(player.mark):
                    print(f"Player {player.number} wins!\n")
                    return player.number
                if turn == 9:
                    print("Game is a draw.\n")
                    return None
```

Switch tokens every other player

Exit loop if any player has won

Exit loop if 9 turns are complete, and no one has won, indicating a draw

Agents: *Player (Interface)*

Methods:

```
from board import Board

class Player:
    def __init__(self, number):
        self.number = number
        self.mark = "X" if number == 1 else "O"
        self.opponent_mark = "O" if number == 1 else "X"

    def get_next_move(self, board: Board) -> int:
        pass
```

Instantiate passed token for player

This method belongs to each child class and it is implemented according to the purpose of the class

Agents: *MiniMaxPlayer (Class)*

represents an automated agent who uses the brute-force minimax algorithm

Methods:

```
# Import libraries
from player import Player
from board import Board
from argmax import argmax

class MinimaxPlayer(Player):

    def get_next_move(self, board: Board) -> int

    def get_minimax(self, board: Board, is_max: bool) -> int

    def get_score(self, board: Board) -> int
```

Test each possible move and return the move with the best MiniMax Score

Search through board states to find MiniMax of a hypothetical move

Returns the score for a win, loss, or draw given the board state. Positive indicates that the given token is winning and negative indicates losing

Agents: *ConditionalPlayer (Class)*

represents an automated agent who uses conditional logic

Methods:

```
from player import Player
from board import Board

class ConditionalPlayer(Player):

    def get_next_move(self, board: Board) -> int

    def get_decisive_move(self, board: Board) -> int

    def has_decisive_move(self, board: Board, space: int, neighbor1:
int, neighbor2: int)-> bool

    def get_non_decisive_move(self, board: Board)
```

If there is a move which will result in a victory for the current player, return that.

If there is a move which will result in a victory for the current player, return that.

If no decisive move exists, use a series of strategic moves which have the best chances of winning (these strategies can be from your own experience)

Agents: *UtilityPlayer (Class)*

represents an automated agent who uses a utility function to evaluate moves

Methods:

```
from board import Board
from conditional_player import ConditionalPlayer
from argmax import argmax
class UtilityPlayer(ConditionalPlayer):
      def get_next_move(self, board: Board) -> int
      def get_utility_of_lines(self, board: Board) -> list

      def get_line_utility(self, board: Board, line: list) -> int


      def get_utility_of_spaces(self, board: Board, utility_of_lines:
list) -> list:
```

- Inherits from the ConditionalPlayer Class
- Returns move with highest utility score
- Returns an array of utility scores for each line (diagonal, row, column)
- Determine utility for a single line, depending on the number of agent and opponent marks
- Based on line utilities, determine the possible moves which the player can make

Agents: *AlphaBetaPlayer (Class)*

represents an automated agent who uses minimax with alpha-beta pruning

Methods:

```
from player import Player
from board import Board
from argmax import argmax

class AlphaBetaPlayer(Player):
   def get_next_move(self, board: Board) -> int
   def get_minimax(self, board: Board, is_max: bool, alpha: int, beta:
int) -> int
   def get_score(self, board: Board) -> int
```

Note: This player is very similar to the MiniMax player, except that the searching stops after a certain condition is met with the alpha and beta variables

Other Agents

- ***HumanPlayer*** (Class) – represents a human player who is prompted for input via the console

- ***SequencePlayer*** (Class) – represents a player who plays a pre-defined sequence of moves (for testing)

- ***RandomPlayer*** (Class) – represents an automated player that choses moves at random