RIGA TECHNICAL UNIVERSITY
Faculty of Computer Science, Information Technology and Energy

Report on the first practical assignment

Study course "Fundamentals of artificial intelligence"

Team number: 14

Teaching staff: Alla Anohina-Naumeca

Project/code link:
https://github.com/Borinskii/Practical_Assignment_Team14

2024/2025 academic year

Demonstration example of the software

<this section should include (1) a description of the game and the changes that were made to it, if applicable, and (2) screenshots showing the game course and the choices to be made, and explanations of them>

Game description

At the beginning of the game, the human-player indicates the length of the numerical string to be used in the game, which can be in the range of 15 to 25 numbers. The game software randomly generates a numerical string of the given length, including 1 and 0.

The initial state of the game is the generated numerical string. Each player has 0 points. Players perform moves sequentially, replacing a pair of two adjacent numbers based on the following conditions:

- the number pair 00 gives 1 and 1 point is added to the player's score;
- the pair of numbers 01 gives 0 and 1 point is added to the opponent's score;
- the pair of numbers 10 gives 1 and 1 points is subtracted from the opponent's score;
- the pair of numbers 11 gives 0 and 1 point for the player's score.

Only one pair of numbers can be substituted per move. The game ends when one number is left in the string. The player with the higher number of points wins. If the number of points is equal, then the result is a draw.

No changes were made to the initial game description.

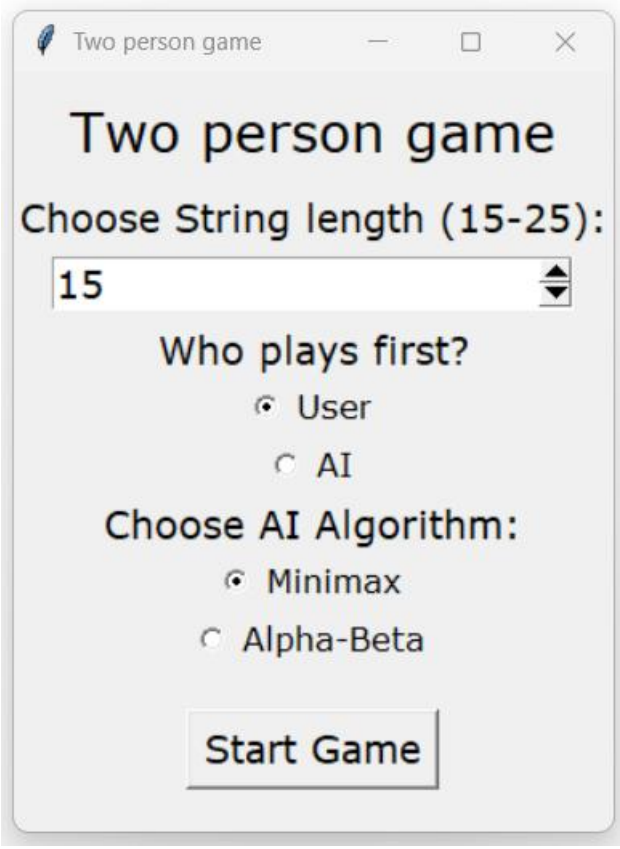
A demonstration example of the software:

1)Choosing the length of string between 15-25,

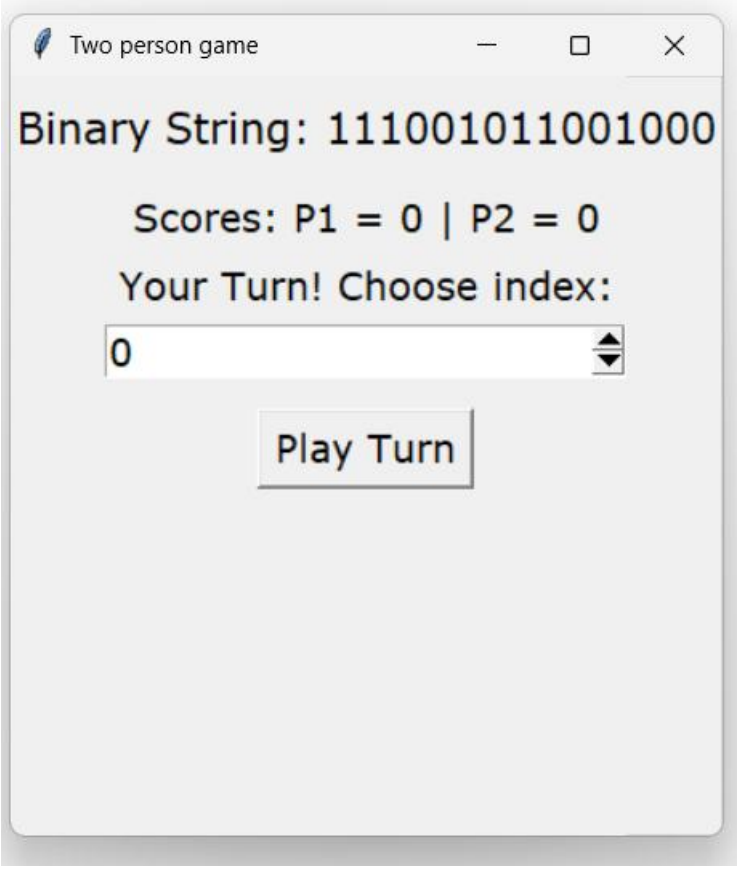
The user chooses whether they want to play first or second. In this case, the user starts the game.

Choosing An algorithm of AI (Minimax or Alphbeta, in this case - Minimax).

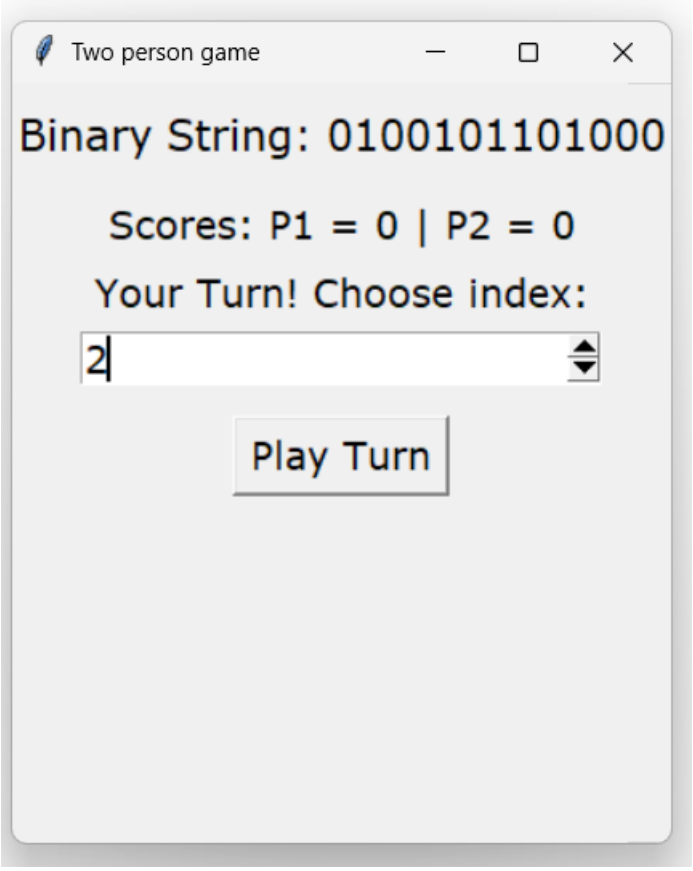
Then start the game:



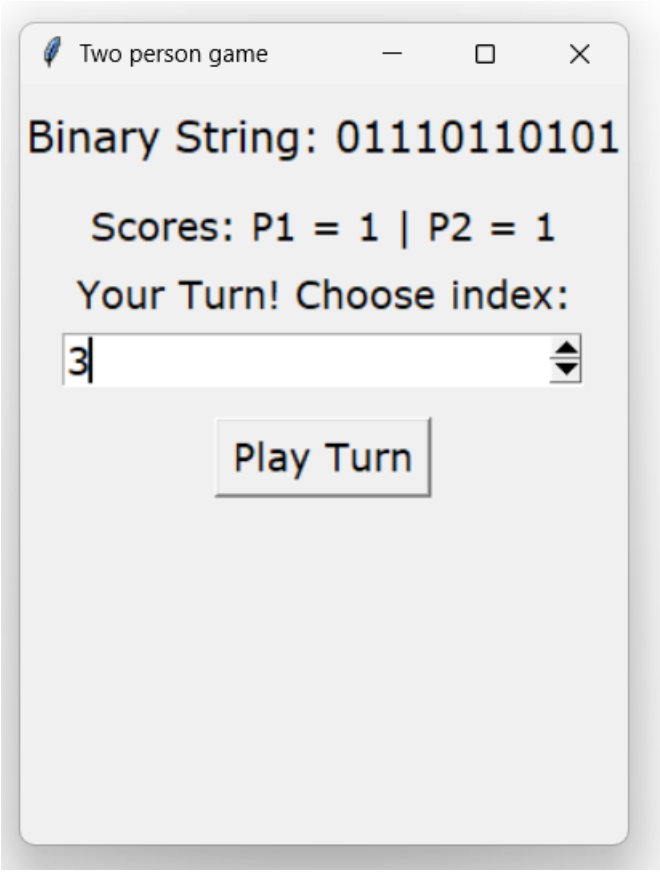
2)String generated in specific length(15) which was chosen by user before and user selects index of string(0) for playing:



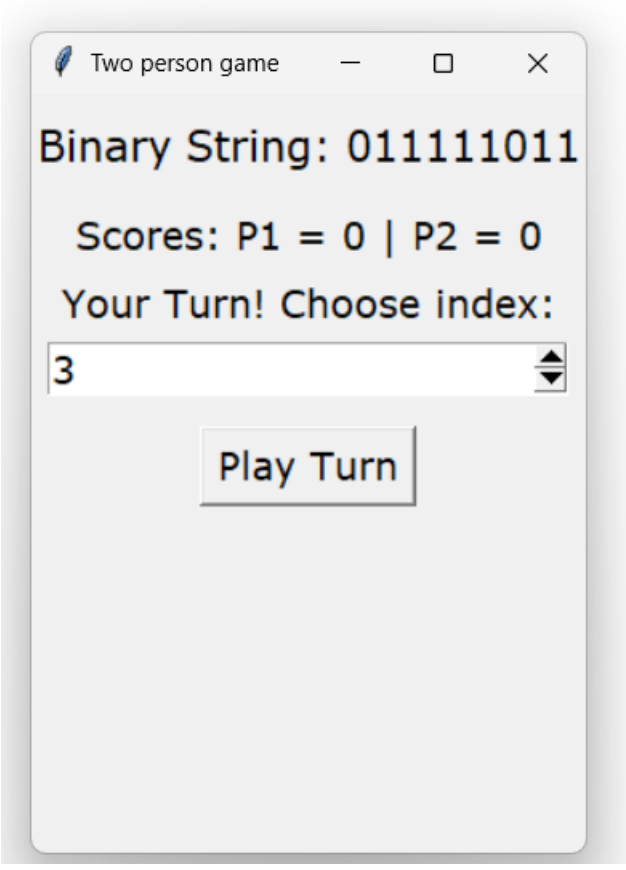
3) User selects index(2) of string after AI makes its move:



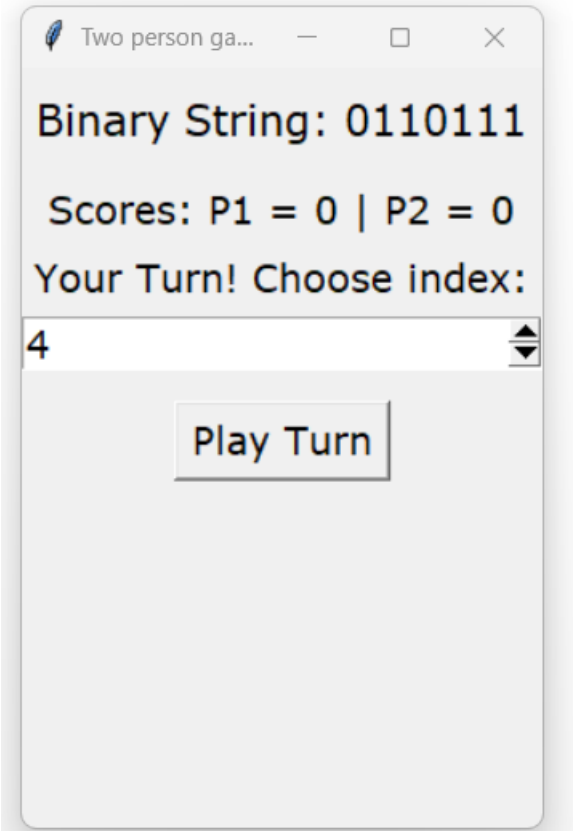
4) User selects index(3) of string after AI makes its move:



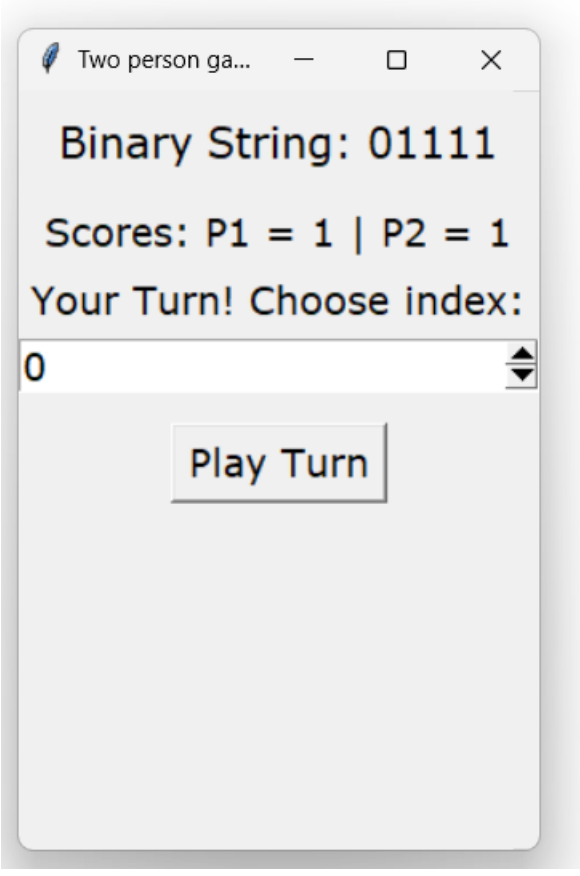
5) User selects index(3) of string after AI makes its move:



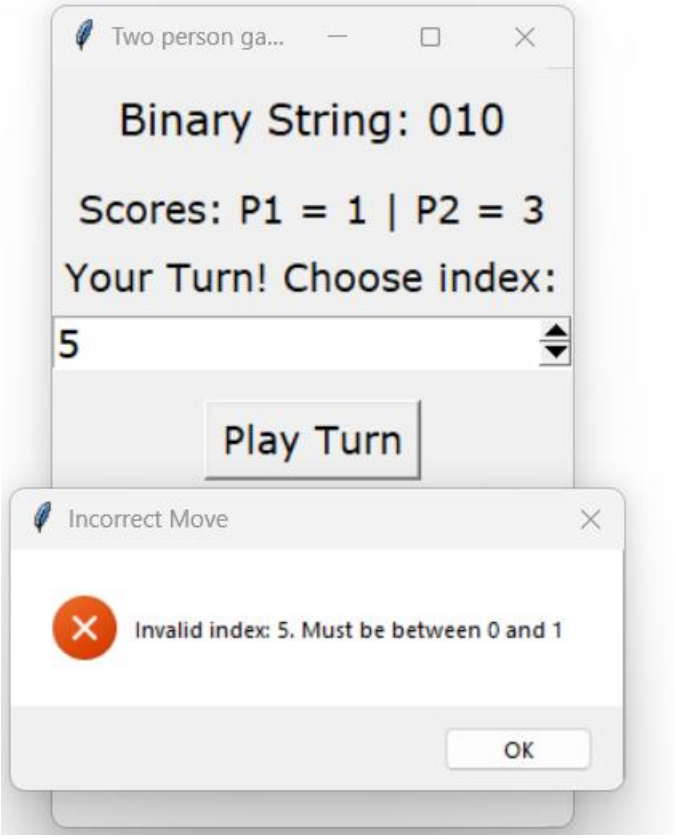
6) User selects index(4) of string after AI makes its move:



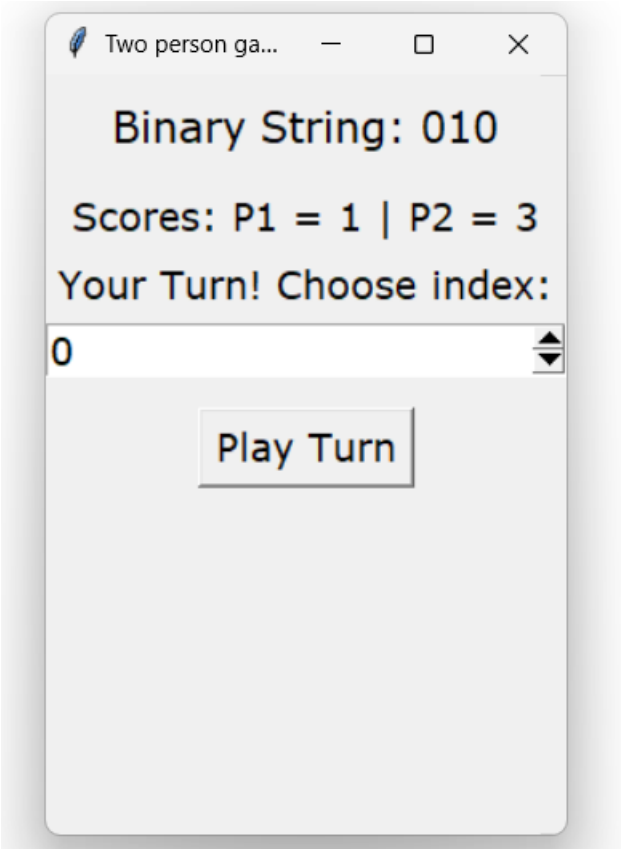
7) User selects index(0) of string after AI makes its move:



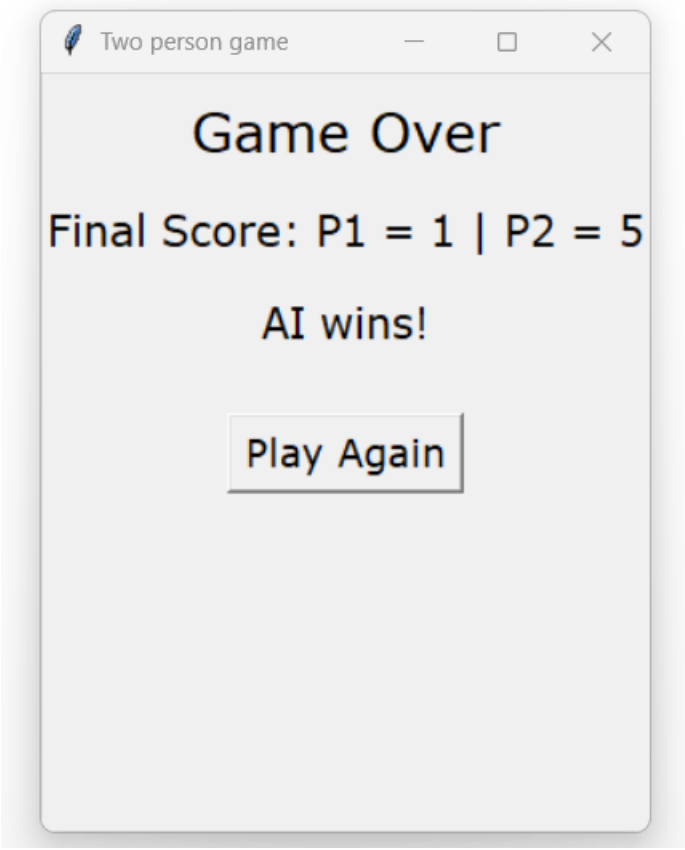
8)User selects index(5) of string for playing, but it can be 1 or 0. So the error message displays on the screen:



9)User chooses correct index(0) for playing:



10) The game ends, and the Player can see who wins the game If player wants, he can play the game again:



Description of data structures and algorithms

Description of data structures:

<a description of the data structures used to store the game tree, with detailed comments on what is stored in each data structure>

1. Class Node:
   Class Node represents a single Node in the game tree. Each Node stores Node id (unique number for each Node), string (a game string at the current moment), points of the first player (p1, at the current moment), points of the second player (p2, at the current moment), level at which Node is located in Game Tree and heuristic function value (by default it is None, but some Nodes that are evaluated by Computer get such values).
   A Node object has function get_node that returns all stored value in this object to the user in readable form.
2. Class Game_Tree:
   Class Game_Tree can represent a full game tree in the game. It stores a set of Nodes that could appear in the game and a set of arcs that connects Nodes from different levels (so that we could represent possible game paths).
   A Game_Tree object has functions adding_node and adding_arcs. Adding_node function takes a Node Class object as input and adds it to the set of Nodes. Adding_arcs function takes ids of 2 nodes: the initial node and following node. Using them, it adds a new arc to the set of arcs between corresponding Nodes.
   Short after implementing this class we understood that in our game it is impossible to create a full game tree due to limited computational resources. That's why we have decided to use another class to represent game states.
3. Class Game_States:
   Class Game_States is used to track and record states of the game. The main difference between this Class and Game_Tree Class is that Game_States object stores game states (as a list of class Node objects) that have already appeared in the game. Also Game_States object stores current game state as a Node object.
   Game_States has several functions: add_node, get_last_node, show_last_node and show_states. Add_node function takes a Node object and adds it to the game states list. Get_last_node function returns a current game state as a Node object. Show_last_node function returns a current game state to the user in readable form. Show_states function returns a list of nodes that have appeared in the game to the user in readable form.

Description of a heuristic evaluation function:

<a detailed description and justification of the heuristic evaluation function>

Our heuristic evaluation function for the Node takes into account 4 things:

1) Node string score (eval_score);
2) Who starts the game (Player or Computer);
3) Score difference;
4) Whose turn is it

Node string score (eval_score) is calculated relatively easy. Since different pairs in the string add or subtract points from players scores, we can make a table that shows how score difference will be

altered by choosing some pair. For example, ‘00’ gives +1 point to the player’s score => current player wins 1 point (+1). ‘01’ gives +1 point to the opponent’s score => current player loses 1 point (-1). ‘10’ subtracts 1 point from the opponent's score => current player wins 1 point (+1). ‘11’ gives +1 point to the player’s score => current player wins 1 point (+1). We can do this for each pair of adjacent numbers in the current string and sum these values up. It will be our Node string score. The higher this value is, the easier for the player it would be to make a move that will be ‘good’ for them.

To calculate Score difference correctly we need to know who is p1 and who is p2. In our implementation p1 is the player who starts the game (it can be Player or Computer). We call them a maximizer. So, p1 is always a maximizer and p2 is always a minimiser. If a player is p1, they want the difference between p1 and p2 to be maximized. If a player is p2, they want the difference between p1 and p2 be minimized, or in another words difference between p2 and p1 be maximized. As we want to make a heuristic function that has higher value for better moves, we should take this into account. In our case a Computer adds to the heuristic function (p1 – p2) if player is not a maximizer and (p2 – p1) if player is maximizer.

Turn also plays a big role in our heuristic evaluation function. In our implementation if a maximizer makes a move, the turn is 1. If a minimizer makes a move, the turn is -1. For example, if Computer is a maximizer, and a turn during the evaluation is 1 (maximizer’s), a Computer should add Node string score(eval_score) to the evaluation function. On the other hand, if a turn is -1 (minimiser’s), a Computer should subtract Node string score from the evaluation function. That’s why we multiply Node string score by Turn.
Our final heuristic function: if player_is_maximizer: (node.p2 - node.p1) + turn * eval_score
If not player_is_maximizer: (node.p1 - node.p2) + turn * eval_score

A computer will use this heuristic function to make moves.

Description of algorithms:

<code of the main algorithms implemented in the software (generating a game tree, assigning heuristic values to graph nodes, applying a game algorithm, finding winning paths) together with explanations. The code must be added to the report in text form only. It is not allowed to add it as a set of images>

```
1. def move_checking(nodes_generated, current_node):
       global j
       string = list(current_node[1])
       for i in range(len(string) - 1):
           # Extracting attributes string, p1 and p2 from current node to alter
   them later to create a new node
           p1_new = current_node[2]
           p2_new = current_node[3]
           pair = string[i] + string[i + 1]
           # removing the pair from the initial string to make a transformation
   according to game rules
           new_string = string[:i] + string[i + 2:]
           # "switching" our pair with some value and changing players points
   according to game rules
           if pair == "00":
               p1_new += 1
               new_string.insert(i, "1")
           elif pair == "01":
               p2_new += 1
```

```
                new_string.insert(i, "0")
            elif pair == "10":
                p2_new -= 1
                new_string.insert(i, "1")
            elif pair == "11":
                p1_new += 1
                new_string.insert(i, "0")
            # our new node is 1 level deeper than current one => increasing level
    by 1
            level_new = current_node[4] + 1
            # increasing id by 1 as well
            id_new = 'A' + str(j)
            j += 1
            # creating new Node
            new_node = Node(id_new, "".join(new_string), p1_new, p2_new, level_new)
            identical, check_id = is_identical(new_node)
            if identical:
                j -= 1
                gt.adding_arcs(current_node[0], check_id)
            else:
                nodes_generated.append([id_new, "".join(new_string), p1_new,
    p2_new, level_new])
                gt.adding_node(new_node)
                gt.adding_arcs(current_node[0], new_node.id)
```

This function is creating a full game tree. It uses nodes_generated list to expand nodes and modifies gt (an object of the class Game_Tree), avoiding duplicates. We don't use this function in our game because we can't generate a full game tree due to lack of computational resources.

```
2. # a function to check if a Node is identical to some other Node in GameTree (in
   our case, it is gt)
   def is_identical(current_node):
       # taking values of id, string, p1, p2 and level from current node
       cur_id = current_node.id
       cur_string = current_node.string
       cur_p1 = current_node.p1
       cur_p2 = current_node.p2
       cur_level = current_node.level
       # iterating through set of existing nodes
       for el in gt.set_of_nodes:
           # checking if some node in set of nodes is identical to "current node"
           # Note: we don't compare cur_id with el.id, as in our game tree there
   can't be 2 identical IDs
           if cur_string == el.string and cur_p1 == el.p1 and cur_p2 == el.p2 and
   cur_level == el.level:
               # If we've found a match, return True and id of a Node identical to
   current node to use it for new arc creation
               return True, el.id
       # If we haven't found a match, return False and current id to use it for
   new arc creation
       return False, cur_id
```

This function is checking if the Node we are trying to add unique or not. If it is not unique, we create an arc from current node to the new node instead of adding a new node to the Game Tree. This function is used in **move_checking** function.

```
3. def heuristic(node, turn, player_is_maximizer):
       s = node.string
       pair_values = {
           '00': 1,
           '01': -1,
```

```
            '10': 1,
            '11': 1
        }
        eval_score = 0
        for i in range(len(s) - 1):
            pair = s[i:i+2]
            eval_score += pair_values.get(pair, 0)
        if player_is_maximizer:
            return (node.p2 - node.p1) + turn * eval_score
        if not player_is_maximizer:
            return (node.p1 - node.p2) + turn * eval_score
```

This function was described in detail in the description of heuristic evaluation function. In short words, it assigns a heuristic value to some Node based on player's scores, turns and evaluation score (calculated using simple algorithm).

```
4. def generate_children(node, is_maximizing):
        children = []
        string = list(node.string)

        for i in range(len(string) - 1):
            p1_new = node.p1
            p2_new = node.p2
            pair = string[i] + string[i + 1]
            new_string = string[:i] + string[i + 2:]

            if pair == "00":
                if is_maximizing:
                    p1_new += 1
                else:
                    p2_new += 1
                new_string.insert(i, "1")

            elif pair == "01":
                if is_maximizing:
                    p2_new += 1
                else:
                    p1_new += 1
                new_string.insert(i, "0")

            elif pair == "10":
                if is_maximizing:
                    p2_new -= 1
                else:
                    p1_new -= 1
                new_string.insert(i, "1")

            elif pair == "11":
                if is_maximizing:
                    p1_new += 1
                else:
                    p2_new += 1
                new_string.insert(i, "0")

            child = Node("temp", "".join(new_string), p1_new, p2_new, node.level +
    1)
            children.append(child)

        return children
```

This function takes a Node object and returns a list of Nodes that can be the result of the next move. Also it takes is_maximizing parameter, which allows a computer to calculate scores properly. As in our implementation p1 is a maximizer and p2 is a minimizer, is_maximizing parameter can show, whose move it is right now – of p1 or p2.

```
5. def minimax(node, depth, is_maximizing, max_depth, player_is_maximizer):
       global nodes_visited
       nodes_visited += 1
       if depth == max_depth or len(node.string) == 1:
           turn = 1 if is_maximizing else -1
           return heuristic(node, turn, player_is_maximizer)

       children = generate_children(node, is_maximizing)
       if not children:
           return heuristic(node, 1 if is_maximizing else -1, player_is_maximizer)

       if is_maximizing:
           max_eval = float('-inf')
           for child in children:
               eval = minimax(child, depth + 1, False, max_depth,
   player_is_maximizer)
               max_eval = max(max_eval, eval)
           return max_eval
       else:
           min_eval = float('inf')
           for child in children:
               eval = minimax(child, depth + 1, True, max_depth,
   player_is_maximizer)
               min_eval = min(min_eval, eval)
           return min_eval
```

This function takes a Node, its depth, is_maximizing parameter (for **generate_children** function), max_depth and player_is_maximizer (for **heuristic** function). It performs a recursive minimax search which is limited by depth(max_depth parameter). It goes through possible moves, minimizing and maximizing moves based on is_maximizing parameter. It returns a score (min_eval or max_eval) that represents the best move based on heuristic evaluation function.

```
6. # possible_moves is a list
   def ai_move_choosing_minimax(possible_moves, max_depth=4,
   player_is_maximizer=True):
       best_score = float('-inf')
       best_nodes = []
       for node in possible_moves:
           player_next_move_is_maximizing = player_is_maximizer
           score = minimax(node, depth=1,
   is_maximizing=player_next_move_is_maximizing, max_depth=max_depth,
                           player_is_maximizer=player_is_maximizer)
           node.heuristic_value = score
           if score > best_score:
               best_score = score
               best_nodes = [node]
           elif score == best_score:
               best_nodes.append(node)
       return random.choice(best_nodes)
```

This function takes possible_moves list that contains Node objects and max_depth. It uses minimax to evaluate each node from possible moves and returns the best Node. If 2 Nodes have the same heuristic function value (and they are best candidates), it chooses randomly between these 2 Nodes.

```
7. def ai_node_creation_minimax(game_states, player_is_maximizer):
       global l
       possible_moves = []
       current_node = game_states.get_last_node()
       string = list(current_node.string)
       for i in range(len(string) - 1):
           p1_new = current_node.p1
           p2_new = current_node.p2
```

```
            pair = string[i] + string[i + 1]
            # removing the pair from the initial string to make a transformation
    according to game rules
            new_string = string[:i] + string[i + 2:]
            if pair == "00":
                if player_is_maximizer:
                    p2_new += 1
                    new_string.insert(i, "1")
                else:
                    p1_new += 1
                    new_string.insert(i, "1")
            elif pair == "01":
                if player_is_maximizer:
                    p1_new += 1
                    new_string.insert(i, "0")
                else:
                    p2_new += 1
                    new_string.insert(i, "0")
            elif pair == "10":
                if player_is_maximizer:
                    p1_new -= 1
                    new_string.insert(i, "1")
                else:
                    p2_new -= 1
                    new_string.insert(i, "1")
            elif pair == "11":
                if player_is_maximizer:
                    p2_new += 1
                    new_string.insert(i, "0")
                else:
                    p1_new += 1
                    new_string.insert(i, "0")
            id_new = 'A' + str(l)
            l += 1
            level_new = current_node.level + 1
            new_node = Node(id_new, "".join(new_string), p1_new, p2_new, level_new)
            possible_moves.append(new_node)
        chosen_node = ai_move_choosing_minimax(possible_moves,
    player_is_maximizer=player_is_maximizer)
        game_states.add_node(chosen_node)
        return possible_moves, chosen_node
```

This function takes an object of Game_States class and player_is_maximizer parameter (it is used for **ai_move_choosing_minimax** function). At first it generates children of the current node and handles them in possible_moves list. After it selects the best node from possible_moves (using **ai_move_choosing_minimax** function) list and adds it to the game_states.

```
8. # this function adds ready-made node after player's move to the game states
    def player_node_creation(game_states, i, player_is_maximizer):
        global l
        current_node = game_states.get_last_node()
        string = list(current_node.string)
        while i < 0 or i > len(string) - 2:
            print(f"Invalid index: {i}, valid range is 0 to {len(string) - 2}")
            i = int(input("Enter a valid index: "))
        p1_new = current_node.p1
        p2_new = current_node.p2
        pair = string[i] + string[i + 1]
        # removing the pair from the initial string to make a transformation
    according to game rules
        new_string = string[:i] + string[i + 2:]
        if pair == "00":
            #checking if p1 was starting
            if player_is_maximizer:
```

```
                p1_new += 1
                new_string.insert(i, "1")
            else:
                p2_new += 1
                new_string.insert(i, "1")
        elif pair == "01":
            if player_is_maximizer:
                p2_new += 1
                new_string.insert(i, "0")
            else:
                p1_new += 1
                new_string.insert(i, "0")
        elif pair == "10":
            if player_is_maximizer:
                p2_new -= 1
                new_string.insert(i, "1")
            else:
                p1_new -= 1
                new_string.insert(i, "1")
        elif pair == "11":
            if player_is_maximizer:
                p1_new += 1
                new_string.insert(i, "0")
            else:
                p2_new += 1
                new_string.insert(i, "0")
        id_new = 'A' + str(l)
        l += 1
        level_new = current_node.level + 1
        new_node = Node(id_new, "".join(new_string), p1_new, p2_new, level_new)
        game_states.add_node(new_node)

        return new_node
```

This function adds ready-made node after player's move (not Computer's) to the game_states.

```
9. def alpha_beta(node, depth, alpha, beta, is_maximizing, max_depth,
   player_is_maximizer):
        # Terminal condition: maximum depth reached or game over (string of length
   1)
        global nodes_visited
        nodes_visited += 1
        if depth == max_depth or len(node.string) == 1:
            turn = 1 if is_maximizing else -1
            return heuristic(node, turn, player_is_maximizer)

        # Generate children based on the current move's turn.
        children = generate_children(node, is_maximizing)
        if not children:
            turn = 1 if is_maximizing else -1
            return heuristic(node, turn, player_is_maximizer)

        if is_maximizing:
            value = float('-inf')
            for child in children:
                value = max(value, alpha_beta(child, depth + 1, alpha, beta, False,
   max_depth, player_is_maximizer))
                alpha = max(alpha, value)
                if alpha >= beta:
                    break  # Beta cutoff
            return value
        else:
            value = float('inf')
            for child in children:
                value = min(value, alpha_beta(child, depth + 1, alpha, beta, True,
```

```
    max_depth, player_is_maximizer))
                beta = min(beta, value)
                if alpha >= beta:
                    break  # Alpha cutoff
            return value
```

This function takes a Node, its depth, is_maximizing parameter (for **generate_children** function), max_depth and player_is_maximizer (for **heuristic** function). It also takes alpha and beta values. It performs a recursive alphabeta search which is limited by depth(max_depth parameter). It goes through possible moves, minimizing and maximizing moves based on is_maximizing parameter. Also it checks if it can skip some paths (alpha and beta cutoffs). It returns a score (value) that represents the best move based on heuristic evaluation function.

```
10.  def ai_node_creation_alphabeta(game_states, player_is_maximizer,
    max_depth=4):
        global l
        possible_moves = []
        current_node = game_states.get_last_node()
        string = list(current_node.string)

        # Generate possible moves (child nodes) from current state
        for i in range(len(string) - 1):
            p1_new = current_node.p1
            p2_new = current_node.p2
            pair = string[i] + string[i + 1]
            new_string = string[:i] + string[i + 2:]

            # Apply the game rules (same as in your ai_node_creation)
            if pair == "00":
                if player_is_maximizer:
                    p2_new += 1
                    new_string.insert(i, "1")
                else:
                    p1_new += 1
                    new_string.insert(i, "1")
            elif pair == "01":
                if player_is_maximizer:
                    p1_new += 1
                    new_string.insert(i, "0")
                else:
                    p2_new += 1
                    new_string.insert(i, "0")
            elif pair == "10":
                if player_is_maximizer:
                    p1_new -= 1
                    new_string.insert(i, "1")
                else:
                    p2_new -= 1
                    new_string.insert(i, "1")
            elif pair == "11":
                if player_is_maximizer:
                    p2_new += 1
                    new_string.insert(i, "0")
                else:
                    p1_new += 1
                    new_string.insert(i, "0")

            id_new = 'A' + str(l)
            l += 1
            level_new = current_node.level + 1
            new_node = Node(id_new, "".join(new_string), p1_new, p2_new, level_new)
            possible_moves.append(new_node)
```

```
    # Evaluate each move using alpha-beta pruning.
    best_score = float('-inf')
    best_nodes = []
    # The next move will be by the player. So if the player is maximizer,
    # then for that move is_maximizing should be True.
    player_next_move_is_maximizing = player_is_maximizer
    for node in possible_moves:
        score = alpha_beta(node, depth=1, alpha=float('-inf'),
beta=float('inf'),
                           is_maximizing=player_next_move_is_maximizing,
                           max_depth=max_depth,
                           player_is_maximizer=player_is_maximizer)
        node.heuristic_value = score
        if score > best_score:
            best_score = score
            best_nodes = [node]
        elif score == best_score:
            best_nodes.append(node)
    chosen_node = random.choice(best_nodes)
    game_states.add_node(chosen_node)
    return possible_moves, chosen_node
```

This function is very similar to **ai_node_creation_minimax** function. In fact, it stores alternative to **ai_node_creation_minimax** and **ai_move_choosing_minimax.** The mechanism of work is the same: At first it generates children of the current node and handles them in possible_moves list. After it selects the best node from possible_moves (using **alpha_beta** function) list and adds it to the game_states.

Comparison of algorithms:

We have conducted 10 experiments with Minimax algorithm and 10 experiments with Alphabeta algorithm. The results can be seen in the table below

Starting number	Who goes first	Algorithm chosen	Number of visited nodes	Average time (s)	Who won
17	Computer	Minimax	95568	0.0402	Computer
18	Player	Alpha-Beta	18448	0.0117	Computer
19	Computer	Alpha-Beta	22861	0.0142	Computer
16	Player	Minimax	48272	0.0261	Computer
21	Computer	Alpha-Beta	51616	0.0295	Computer
16	Computer	Minimax	381469	0.1228	Computer
24	Player	Alpha-Beta	43654	0.0303	Computer
17	Computer	Minimax	95568	0.0402	Computer
18	Player	Alpha-Beta	18448	0.0117	Computer
19	Computer	Alpha-Beta	22861	0.0142	Computer
25	Computer	Minimax	750776	0.2222	Computer
23	Computer	Alpha-Beta	70419	0.0298	Computer
21	Player	Minimax	229408	0.0725	Computer
19	Computer	Alpha-Beta	23702	0.0123	Tie
17	Player	Minimax	68720	0.0259	Player
15	Computer	Alpha-Beta	8567	0.0055	Computer
16	Player	Minimax	95568	0.0378	Computer
21	Computer	Alpha-Beta	49989	0.0242	Computer
24	Player	Minimax	483032	0.1572	Computer
24	Computer	Minimax	605144	0.1705	Computer

:

We can clearly see that using a **Minimax** algorithm Computer won 9/10 times, having a **win rate of 90%**. We managed to win only once => a player has a **win rate of 10%.** Depending on initial string and its length (from 16 to 25), **number of visited nodes** goes from 48272 to 750776, which indicates that with longer strings Minimax will struggle to work. Along with number of visited nodes, **average time per move** goes from 0.0261s to 0.2222s. Although this numbers are not very good looking, a Minimax algorithm with depth 4 (in our case) manages to make moves relatively fast and win games.

As for the second algorithm, using an **Alphabeta** algorithm Computer also won 9/10 times, having a **win rate of 90%**. We managed not to lose once (it was a draw) => a player has a **win rate of 0%**, and a **draw rate is 10%**. Depending on initial string and its length (from 15 to 24), **number of visited nodes** goes from 8567 to 43654. Along with number of visited nodes, **average time per move** goes from 0.0055s to 0.0303s. We can see that Alphabeta algorithm visits much less nodes (even though the depth was the same, 4)due to cutoffs. This allows Computer to calculate next move much faster, making Alphabeta algorithm suitable for longer strings without losing efficiency (win rate is the same for Minimax and Alphabeta).

In conclusion, deciding between Minimax and Alphabeta algorithms, it would be much more beneficial to **choose Alphabeta** due to its robustness and efficiency (however, both algorithms will have a high and stable win rate). Moreover, the implementation of Alphabeta is not much more complex than implementation of Minimax algorithm.

Information sources

- Minimax algorithm - https://en.wikipedia.org/wiki/Minimax
- Alphabeta algorithm - https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
- Additional practical (code samples) and theoretical information - Fundamentals of Artificial Intelligence ORTUS Course page