School of Computing
FACULTY OF ENGINEERING

UNIVERSITY OF LEEDS

Develop an AI algorithm to play the board game Blokus

Sharon Mathew

Submitted in accordance with the requirements for the degree of
MSc Mobile Computing and Communication Networks

2016/2017

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Dissertation	*Report x2*	*SSO (23/08/17)*
Dissertation	*PDF*	*VLE (23/08/17)*
Blokus Code	*Software codes*	*Supervisor, assessor (23/08/17)*

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

Signature of Student: _____

Abstract

The project investigates the development of a high-quality artificial player that can defeat human players at the board game *Blokus*. Various *artificial intelligence* techniques have been investigated, and the *minimax algorithm* and *heuristic state evaluation* were chosen to implement the artificial player.

Acknowledgement

Firstly, I would like to thank Jesus Christ for guiding me throughout this dissertation according to Luke 12:12. You have really proved Luke 1:37.

Thanks to my family and friends for the support and encouragement.

Finally, I would like to thank my supervisor Dr Brandon Bennett for pointing me the right direction and always supporting me throughout my project.

Table of Contents

1 Introduction

In the early days of *Artificial Intelligence* (AI) [5] in game playing, the easiest path to achieving high performance was believed to be imitating the human approach. This approach was difficult due to problems such as apprehending and encoding human knowledge. Human strategies are not essentially the best computational strategy. Hence, a search-intensive approach came about which produced high-quality performance; a large influence in the field of AI in game playing. Since then, various search techniques were developed and applied to problems such as optimisation, planning, and bioinformatics.

There has been numerous research conducted on AI programs that play adversarial games using search-intensive AI techniques. *Deterministic* [5] board games have been researched deeply, and very efficient implementations of AI players in games have been implemented for games such as *Chess* and *Checkers*. AI players in these *perfect information* games have been developed to such high-quality using search-intensive techniques, that it has been able to defeat the best human players in the world. However, many games are yet to be studied to implement an AI player that can defeat humans. Therefore, there is a need to investigate on more board games to find a solution that can effectively beat human players. Different games require different aspects of intelligence.

In this project, the board game called Blokus has been chosen for study. Various AI techniques can be applied for the program to play effectively against humans. However, after some research, it has been decided that AI techniques including *minimax algorithm* and *heuristic state evaluation* will be used to develop the AI player of the board game.

1.1 Dissertation Outline

Chapter 1 – Introduction sets the scene as well as introduces and motivates the project.

Chapter 2 – Aim and Objectives state the aim and objectives of the project. The project plan is also discussed with a timescale provided.

Chapter 3 – Background describes the project area in detail and explains decisions taken within the project and why.

Chapter 4 – Design discusses the plan and workflow of the implementation that was carried out.

Chapter 5 – Implementation presents how the program was developed and why certain decisions within the task were made.

Chapter 6 – Testing is a description of the tests carried out and how it was done.

Chapter 7 – Evaluation analyses the test results and provides a discussion on whether the project objectives were met and analyses the project itself by discussing what went well and what could have improved.

Chapter 8 – Conclusion discusses the project also, generalises and states possible future work for the project.

2 Aim and objectives

The aim and objectives will be used within the project as checklists to make sure that the project is going the right way. The objectives were decided after some amount of background research was conducted.

2.1 Aim

To investigate and implement the use of AI techniques to produce a program that can ‘intelligently’ play the game Blokus.

2.2 Objectives

1. Develop a software capable of correctly playing the game of Blokus per its rules.
2. Develop a simple rule-based AI algorithm to play the game of Blokus.
3. Develop a GUI interface to show the game states.
4. Investigate the use of AI techniques such as Minimax, Heuristic state evaluation, Machine learning to enable the algorithm to play ‘intelligently’.

2.3 Project Plan

The chart shown in Figure 2.1 describes how time will be split to meet the aim and objectives. The schedule for the project starts from the beginning of March up to the end of August. March has mainly been used to think about the implementation of the board game and to do some background research on the topic to understand the context of the project properly. April was spent on doing the deliverables for this module while also thinking about how the implementation of the core game will be made. May to start of June has been left blank due to exams and revision as this is also quite important. Once exams finish at the start of June, the design and the implementation stage of the project will begin. The testing stage will be in mid-August which is also near the end of time for the implementation of the project. The results will be gathered and analysed and also the bugs found will be fixed, and the program will then be re-tested. The write up of the report will be a continuing process, and this will be written while the design and the implementation stage is being carried on. As per the project plan, it is the aim to finish the project by the end of August to be able to error check, format and submit the report by the start of September.

The methodology chosen for this project is an iterative waterfall model. This model was selected to meet each objective one by one by iterating through the design and implementation stage depending on whether there is enough time left on each objective. I have left extra time for each task than required if I come across any unexpected problem. If I do not come across any problem, I should have enough time to implement the code to allow a human to play against the program.

There will be weekly meetings arranged with my supervisor every week to ensure that I am going on the right track with my project and solving any issues or doubts that I may have regarding the project.

Task Name	Start	Finish	Duration
			Mar | Apr | May | Jun | Jul | Aug
Background	01/03/17	30/04/17	
Design	05/06/17	05/08/17	
Implementation	10/06/17	15/08/17	
Testing	13/08/17	18/08/17	
Write up	15/07/16	30/08/17	

Figure 2.1: Project Plan

3 Background

Background research of a project is necessary to gain an understanding of the project itself and information on how to carry forward the project to meet the aim. There may already be various research and projects conducted that are related to this project. This knowledge can be used to aid and implement within my project. Therefore, background research provides an excellent base and allows decisions to be made on what to do regarding the project.

3.1 Game Theory

Modern *game theory* [6] began with the work of Zermelo in 1913, Borel in 1921, von Neumann in 1928 and the seminal book of von Neumann and Morgenstern in 1944. Game theory is the study of models of conflict and cooperation between intelligent decision makers within a competitive situation. It provides general techniques to analyse the situation. Optimal decisions are made by two or more individuals strategically where every decision made influences the welfare of every individual in the situation. This theory is used within several areas including psychology, evolutionary biology, economics, and business.

3.1.1 Game

Game theory does not just apply to recreational games but the term “game” refers to a situation in which individuals or independent actors share formal rules and consequences [7]. To be able to understand and investigate different classifications of games, and game strategies, it is important to know the basics of a game. The components of a game consist of [8]:

- **Rules**: A game has strict rules which must be followed as this lays down the boundaries of what is allowed and what isn’t. These boundaries allow the game to be analysed to have a set of possible moves which may be already known in advance.
- **Outcomes**: Each game can have various possible outcomes. Each outcome is a value of one or more decisions made.
- **Payoffs**: Each outcome produces payoffs for the players. Every player wants to win the game.
- **Uncertainty of the Outcome**: The outcome of a game is unpredictable because if it is a one player game, then it will have some chance element and if it is two or more player game, a player cannot know in advance the move of another player; causing uncertainty.

- **Decision-making**: A decision must be made in a game to continue the game forward. These decisions allow the ability to analyse the game using game theory.
- **No cheating**: Cheating is when the game is not played as per the rules. Game theory always follows the rules.

3.1.2 Terminology in a game

Game: Described by a complete set of rules

Play: Instance of a game

State: A specific arrangement within the game using the existing components

Move: A decision made at a state

Strategy: A plan to aid the player to choose a move at every possible state

Rational behaviour: Each player has different behaviour and will try to optimise their payoffs while being aware that the other players are trying to optimise their payoffs.

3.1.3 Game classifications

One, Two, or N-player

Games that have a finite number of players are referred to as an n-player game [9]. The number of players in a game effect the strategy of each player to win the game; the higher the number of players, the higher the difficulty in the assessment of the next possible move chosen by each player [6]. This difficulty is also influenced by the frequency of the decisions made by each player. In one player games, such as roulette, it has uncertain elements and is not influenced by any players. There are no decisions made in games such as these since the number is usually chosen randomly by the player; therefore, it is not possible to create a winning strategy for such games. In one-player games that do not have uncertain elements, the winning strategy will be straightforward. One-player games are usually not considered game theoretical.

In two or more player games, each player will try to maximise it's expected payoff. This can be either done by only increasing one's payoff, by decreasing the other player's payoff, or by doing both in a move (further explained in Section 3.1.4). When a player chooses a move, this is influenced by what move the next player may choose next.

An example of a two-player game [6]: For Player 1 to select a move in a play, Player 1 will need to assess each of Player 2's possible choices. To do this, Player 1 must be in the shoes of Player 2. At this moment, Player 2 will be trying to solve its problem by considering all the possible moves of Player 1. Here, Player 2 may be putting itself into the shoes of Player 1. So, the solution to solving each player's problem depends on the solution to the other player's problem. Therefore, to maximise the payoff of the current player, each move of both player's must be analysed together, like a system of equations.

Simultaneous or Sequential

In a *simultaneous* game, each player has one move each play and every player may make a move concurrently in a play; e.g. Rock-Paper-Scissors. If the players do not move concurrently, then the players playing after a player will not have knowledge of the earlier moves made by the player which effectually makes the game simultaneous.

In a *sequential* game, only one player moves in a play. In such games, it is possible for a player to move several times in a play; e.g. Monopoly. Unlike simultaneous games, a player will have some knowledge about all the previous moves made by the other players. This knowledge may not be perfect information but can be some or minimal amount of knowledge.

A simultaneous game and a sequential game are both represented differently within a game. Simultaneous games are denoted by payoff matrices, and sequential games are denoted by game trees. The representations are further discussed in detail below

There also exist games that are not simultaneous or sequential [8].

Deterministic or Stochastic

At a specific state of the game, when a specific move is played, the resulting state will always be the same regardless of how many times the same move is played in the same state; this is a *deterministic* game. There are no other influences on the game, and therefore, the resulting state cannot be changed. In a deterministic game, a state can be recreated from the same order of moves that were played.

A game is *stochastic* when there is some element of *randomness*. Some one player games are stochastic because there is a stand-in player who makes random moves. This player is not considered as a second player but is there to provide the element of randomness. Element of randomness means when there are moves made due to "chance of nature"; this includes the rolling of a dice, shuffling of cards, etc. In stochastic games, when a specific

move is played at a specific state the resulting state may change each time it is repeated; causing uncertainty. In the existence of uncertainty, payoffs are calculated by estimation using the estimated average of the probabilities of the next possible states [10].

Perfect or Imperfect Information

Perfect information is when a player has full knowledge about the current state. In perfect information games, each player has full knowledge of the previous moves made by every player. The player can know everything about the current state by knowing the initial state and the previous moves made by each player. This makes every player fully aware of the current state which also involves the knowledge of the pieces/cards each player has left, the possible moves that can be made, the payoffs of each possible move at a state, etc. Perfect information is not the same as *complete information* where each player knows the strategies and payoffs that are available to the other players. Instead, perfect information allows every player to have the same amount of knowledge as each other [11]. Examples include chess, tic-tac-toe, checkers, etc.

Imperfect Information is the opposite of perfect information where the player can have no knowledge or minimal knowledge about a specific state. Simultaneous games are an example of such games where a player may not have any knowledge about other player's previous moves or pieces/cards they have left either at the start of the game or through the entire game. Examples include UNO, Poker, Scrabble, etc.

Zero Sum or Non-Zero Sum

Zero sum games [8] are when players cannot modify the resources within the game by adding or taking away. In this case, the total payoffs in the game for all the players, add to zero. In such games, a gain for a player is an equal amount of loss for another player. A good example of such a game is chess where the gain of a player is by attacking and capturing a piece of another player which in turn is a loss for the other player. In a zero-sum game, a win can be defined as +1, a loss can be defined as -1, and a draw can be defined as 0. Such games are called *strictly competitive* games.

Non-zero sum games are where all the players can gain together and lose together. So, the total payoffs in a game for all the players can add up to be less than or higher than zero. A gain to one player does not necessarily result in the loss to another player. Games that can have more than one winner naturally comes under this type of games.

Symmetric or Asymmetric

A *symmetric* game [12] is when a resulting game for a specific player does not change depending on the identity of the player. Even if the identity of the player changes, the resulting payoff to the strategies of each player stays the same. Two player games are usually symmetric, and this is if both the players use the same strategy space and if they swap the strategies then they swap the payoffs [13]. Figure 3.1a shows an example of a symmetric game which lists the payoffs for Player 1 and Player 2; a player's payoff can be expressed as a transpose of the other player's payoff [12]. Examples of such games include prisoner's dilemma, chicken, and battle of the sexes.

Asymmetric games are the opposite of symmetric games where the identity of the player affects the resulting game. Such games usually do not have same strategies used by every player. However, it is still possible to use same strategies and for the game to still be asymmetric; Figure 3.1b shows an example of an asymmetric game.

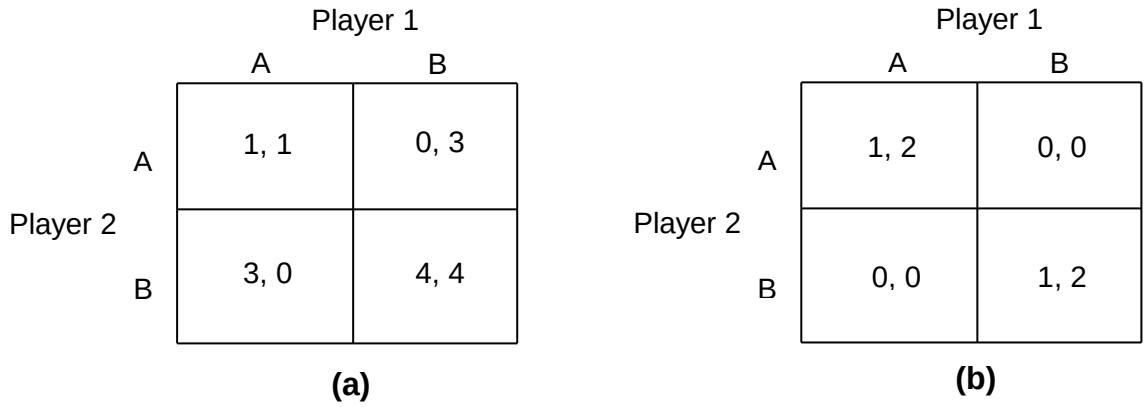


Figure 3.1: (a) a symmetric game (b) an asymmetric game

Cooperative or Non-Cooperative

If a game [6] has players who can form a commitment or a contract that is externally enforced, this is called a *cooperative* game. The competition is between groups of players rather than between each player. The coalition of groups of players is due to external enforcement. If there is a contract or a negotiation made between two players, then other players can also contribute to this. Cooperative games [14] are analysed by predicting which coalitions will form, the combined moves that the groups make, and the consequential summed payoffs. Cooperative games have three or more players because the objective is to win the game, so in a two-player game it would defy the purpose to play the game if a player

is aiding another player in winning the game. However, there are two player cooperative games where the players play together to win against the game.

Non-cooperative games are where players may not form a cooperation or an alliance with each other. If they do form a commitment, it is a self-enforced agreement.

Cooperative games only provide high-level approach which describes the structure, strategies and the payoffs of coalitions. However, non-cooperative games look at these as well as the result of bargaining procedures on payoffs with each alliance. Therefore, cooperative games are analysed through the approach of non-cooperative games given the adequate assumptions to cover all the possible strategies that are available to all the players due to the external enforcement of coalition between players.

3.1.4 Game Strategies

There are two main types of strategies: pure and mixed strategy. A pure strategy specifies a complete description of how a player will play the game. In each state, the player will select the move to play using the description to determine the move. The strategy set of a player is the combination of the pure strategies available to that player.

A mixed strategy [10, 15] involves randomisation, using positive probabilities that summate to 1, to determine the player's move to play. Probability is given to a minimum of two moves in distinct pure strategies. Players can then randomly choose from one of the pure strategies that were given a specific probability. Probabilities are continuous; therefore, players have an infinite number of mixed strategies available to choose from. In rare cases, it is also possible for a mixed strategy to be a choice of one of the pure strategies.

Mixed strategy [15] may be much more efficient in comparison to pure strategy when there is a finite and known or predictable pure strategy to the opponent. This is a disadvantage to the current player. In the example of Rock-Paper-Scissors, if the pure strategy always chooses one move (e.g. Paper) every time, then the opponent will be able to predict the move and can play a move to give them a maximum payoff (e.g. Scissors). However, if the mixed strategy equally distributes the probabilities between all moves, then the player can choose a move at random which makes it unpredictable for the opponent about the next move.

Strategy sets are combination of strategies for all players which specifies all possible moves in a game. This strategy set may be finite in games such as Rock-Paper-Scissors where the strategy set would be {Rock, Paper, Scissors}. A strategy set may also be infinite if there are an infinite number of discrete strategies available; for example, the strategy set for an auction can be infinite as it could be {£10, £20, £30, …}.

3.1.5 Game Representations

As mentioned before, a game is a model. To define a game, it must specify elements such as the players in the game, the amount of information available to each player at a specific state, the moves available to each player at a state, and the payoffs at each outcome [16].

Coalitional Form

Most cooperative games are represented in the *Coalitional Form* (sometimes called the *Characteristic Function Form*). As mentioned before, there are no restrictions on the contract that can occur among players. Also, [17] an assumption is made that there is a *transferable utility* which allows *side payments* to be made between the players under the contract. This side payment may be used to encourage players to use specific mutually beneficial strategies. Hence, this encourages the players with similar objectives in the game to form alliances. In games that possess a transferable utility, the payoffs are not given separately, but the coalitional form determines the payoffs of each group of players. Formally, the coalitional form is seen as (N,v) where N is the set of players and v is the characteristic function of the game; v: 2^N-> R is a normal utility.

Extensive Form

Extensive Form is very effective to be used to represent games with time sequencing of moves such as sequential games. As shown in Figure 3.2, the form is represented as a decision tree (game tree) where each node represents a state, and each branch represents a choice of move for a player; the player is specified by the number denoted next to each node. The numbers specified at the bottom of tree are the payoffs. The root of the tree could represent the initial state of the game or the current state if the game has already started.

To solve this type of games, *backward induction* is used which works up the game tree to determine what a rational player at each node would do and continue to work up until the root of the tree is reached. This type of form can be used to find an optimal move for a player giving the player the maximum payoff.

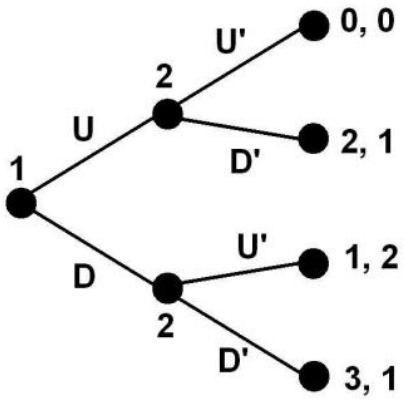


Figure 3.2: Extensive Form [3]

Normal Form

Normal Form, as shown in Figure 3.1 and 3.3, is represented using a matrix which shows the players, payoffs, and a set of possible moves for each player. This form can be formally represented by any function that associates a payoff to each player with all possible combinations of moves. It is assumed that the players play a move at the same time and that they do not have any knowledge of the moves of the other; hence, it is very effective to be used for simultaneous games. If the players may have some information about the moves of other players, then extensive form is used to represent such games.

Player1\Player2	L	R
U	3 , 3	0 , 5
D	5 , 0	1 , 1

Figure 3.3: Normal Form [3]

3.1.6 Solved Games

A *Solved* game is a game where at any state, the final decision of the game (win, draw, or lose) can be correctly predicted depending on the move that the player has chosen; in assumption that both the players are playing *perfectly*.

Perfect play is an optimal strategy for a player that leads them to gain the maximum payoff or best outcome in a state regardless of the move played by the opponent. A player who is using the optimal strategy is said to be playing perfectly. A perfect player in a drawn position will always have the outcome of the game as a draw or a win, never a loss.

Two-player [18, 19] games can be solved on several levels such as *ultra-weak*, *weak*, *strong*. Ultra-weak solved games determine whether the player will win, draw, or lose the game from the current state, in the assumption that both players play perfectly. Weak solved games provide an algorithm, with proof that each move is optimal in an ideal game produced by the algorithm, for the current player to win or draw the game from the start of the game, against any probable moves of the opponent. Strong solved games provide an algorithm to provide perfect moves from any state, regardless of any wrong moves played so far in the game.

3.1.7 Game Tree

Game Tree is a directed graph where each node represents a state, and each level represents a player's move. Game trees are used for games that are represented in the Extensive Form. Game trees are very important in AI because they allow algorithms to search the game tree to find the best move. Game trees for games such as Tic-Tac-Toe are easily searchable, but trees for games such as Chess are too large to search through. For games, such as these, an algorithm generates a specific number of levels to reduce the complexity and computation. Figure 4 shows an example of a partial game tree.

In a game of Tic-Tac-Toe [2], there are approximately five legal moves per state on average, and a total of 9 levels in a game. Therefore, there are there are 5^9 = 1,953,125 nodes which is reasonable to compute. In a game of Chess, there are approximately 35 average branching factor and approximately 100 levels per game. Therefore, there are approximately $35^{100} = 10^{154}$ nodes which are completely infeasible to compute.

3.2 Artificial Intelligence Techniques

3.2.1 Heuristic State Evaluation

Heuristic State Evaluation techniques are usually used to find a method that is not optimal or perfect but an estimate to a satisfactory solution. In the situation of when an optimal solution is not possible to be determined due to the lack of time, a heuristic technique is used to find a solution that is satisfactory. The trade-off measures used to decide whether a heuristic is required to find a solution to a problem includes Optimality, Completeness, Accuracy and Precision, and Execution time.

A *heuristic function*, also called a *heuristic*, is used to rank each branch at each branching step to decide which branch to follow; it may approximate the exact solution [20]. Heuristics

are the main part of AI and the computer simulation of thinking since they can be used even in a situation where there are no algorithms available to find a solution [21].

Within a game, a heuristic may be used to evaluate a state to give an estimated payoff of that state. A heuristic can be used in such a way to compare each state to find the best next state that will give the maximum payoff. Each move has a cost and a gain [2], evaluating a state gives a payoff of playing a specific move from the current state, which produces the next state that is the resulting state. Typically, a state can be evaluated by calculating how good the state is for a player and the opponent and subtracting the opponent's score from player. In a chess game, this may be by subtracting the value of the white pieces on the board from the black pieces, if the player is playing white and the opponent is playing black.

A heuristic can be designed for a game by taking into consideration its features and properties. Each feature can be taken together into categories; for example, in a chess game, the number of queens of the white and the number of queens of the black can be combined into a category. The features and properties of each game may vary, however, an example of these may include the current state, the number of players, position of each piece in the game, number of moves available, each piece available, etc. Each property can be assigned a value which is used by the heuristic function to evaluate all the next possible moves. The more the information given to the function, the more precise the heuristic will be to the actual result.

State evaluation heuristics [2] often require much less computational speed in comparison to *heuristic search* since they only require the information regarding the current state to be able to compute the heuristic. However, the computation of a state evaluation heuristic may also be too complex in the case when it is required to compute the expected value for each feature separately for the player and the opponent. Therefore, the evaluation function uses a simpler version of a heuristic function, *weighted linear function*:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

Each function f_x where x is within 1 to n, represents a feature or property such as the pieces in the state, and each weight w_x where x is within 1 to n, is a parameter that can be altered as required by the heuristic designer, since this represents the value of the component.

For example, $w_1 = 9$ with $f_1(s) = (number\ of\ white\ queens) - (number\ of\ black\ queens)$

3.2.2 Minimax and Maximin

Minimax is a decision rule used in game theory to minimise the maximum loss. When working with gains, the decision rule *Maximin* is used to maximise the minimum gain.

Minimax is often used in zero-sum games to minimise the opponent's maximum payoff. Therefore, since in a zero-sum game the gain of the player results in a loss of the opponent, this results in maximising the player's minimum payoff. Maximin is frequently used for non-zero sum games to maximise the player's minimum payoff.

Both these algorithms were originally used for two-player games to cover all the possible moves of both players. The players [2] will take alternative turns, and it is assumed that each player plays to their best ability to maximise the loss in minimax or to minimise the gain in maximin for the opponent. Each game is represented in Extensive Form, and this is shown in Figure 3.4 which describes a minimax partial game tree for Tic-Tac-Toe in the perspective of player X, as it is trying to maximise the payoff for player X. This game tree shows all the possible moves by both players. Each level in the tree is a ply, and it represents a turn of a specific player. The branch follows the move that each player chooses. A terminal test is when there is either no more moves available for any player, resulting in a draw, or when either player wins the game. The utility function is applied to the state once the game has terminated to generate either a -1 to represent a loss, 0 to represents a draw or +1 to represent a win for Player X. The utility values are then carried up recursively where player O tries to minimise the payoff of player X, and player X tries to maximise its payoff. When the values reach the root of the tree, the branch with the maximum payoff is selected.

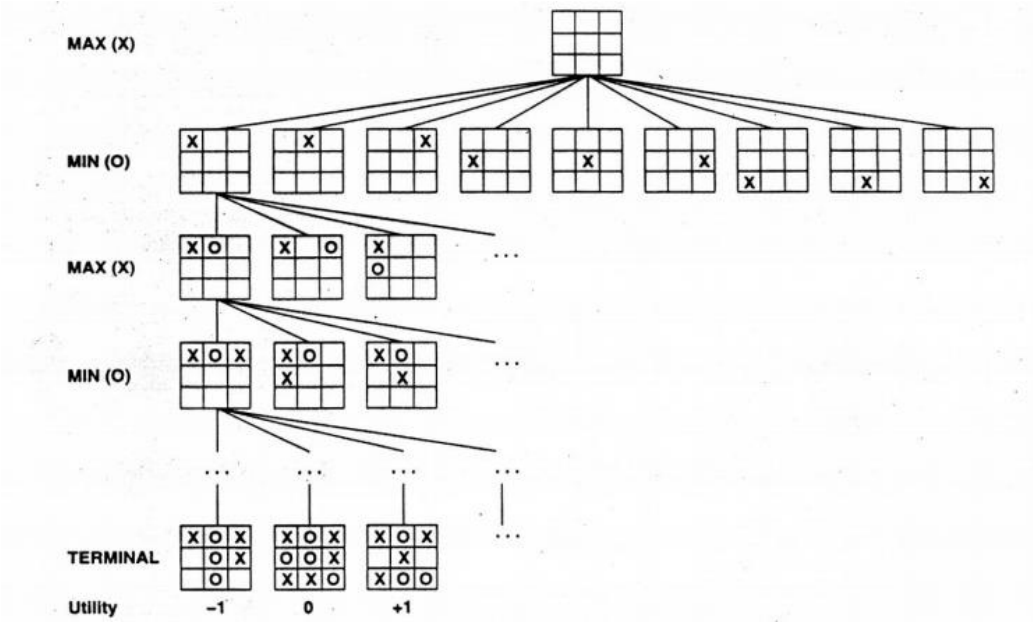


Figure 3.4: A minimax partial game tree for Tic-Tac-Toe [2]

The Minimax pseudo-code algorithm is as follows:

```
function MINIMAX_DECISION (state) returns an ACTION
      MAX_V ← MAX(MIN_VALUE(state, a ∈ ACTIONS))
      return action in ACTIONS with value MAX_V

function MAX_VALUE(state, game) returns a utility value
      if TERMINAL_TEST(state) then
            return UTILITY(state)
      v ← -∞
      for each a in ACTIONS(state) do
            v ← MAX(v, MIN_VALUE(state, APPLY(state, a)))
      end
      return v

function MIN_VALUE(state, game) returns a utility value
      if TERMINAL_TEST(state) then
            return UTILITY(state)
      v ← ∞
      for each a in ACTIONS(state) do
            v ← MIN(v, MAX_VALUE(state, APPLY(state, a)))
      end
      return v
```

3.2.3 α-β pruning

Usually, evaluating every node within a game tree using the minimax or the maximin algorithms can be very expensive. Therefore, *α-β pruning* is a search algorithm that decreases the number of nodes that is evaluated using the fact that some branches can be eliminated from the game tree (*pruning*). A game tree has a time complexity of $O(b^m)$ where b denotes the average number of branches on each level and m denotes the maximum depth of the tree. With perfect ordering, α-β pruning can decrease this complexity by up to $O(b^{m/2})$ [22].

The idea of this algorithm [2, 23] is to do a *depth-first* search to generate a partial game tree and then the heuristic state evaluation function is applied to all the leaf nodes of the tree. The α and the β bounds are then computed on the internal nodes which cut down on subtrees. The α bound is value for the best alternative for MAX along the path to state. The β bound is value for the best alternative for MIN along the path to state. The pseudo-code algorithm for α-β pruning is [23]:

```
function ALPHA_BETA_SEARCH(state) returns an ACTION
      v ← MAX_VALUE(state, -∞, ∞)
      return action in ACTIONS(state) with value v

function MAX_VALUE(state, α, β) returns a utility value
      if TERMINAL_TEST(state) then
            return UTILITY(state)
      v ← -∞
      for each a in ACTIONS(state) do
            v ← MAX(v, MIN_VALUE(state, α, β))
            if v ≥ β then
                  return v
            α ← MAX(α, v)
      return v

function MIN_VALUE(state, α, β) returns a utility value
      if TERMINAL_TEST(state) then
            return UTILITY(state)
      v ← ∞
      for each a in ACTIONS(state) do
            v ← MIN(v, MAX_VALUE(state, α, β))
            if v ≤ α then
                  return v
            β ← MIN(β, v)
      return v
```

3.2.4 Expectiminimax

In stochastic games, it is not possible to use minimax algorithm because the chance elements in the game do not guarantee the payoffs that were discovered. This is where *expectiminimax* comes in. Expectiminimax is very similar to minimax; however, it takes into account the chance elements and uses this within the algorithm to maximise the payoff. An

Expectiminimax tree has min and max nodes, and in addition to this, it also has chance nodes which take an expected value of a random event occurring [24]. The minimax tree alternates between the max and min nodes, which is same in the expectiminimax tree; however, it has the addition of chance nodes being interweaved with these nodes. The pseudo-code algorithm of expectiminimax is very much like minimax. It can be altered by making a few changes such as [2]:

```
…
if state is a MAX node then
      return the highest EXPECTIMINIMAX-VALUE of ACTIONS(state)
if state is a MIN node then
      return the lowest EXPECTIMINIMAX-VALUE of ACTIONS(state)
if state is a chance node then
      return average of EXPECTIMINIMAX-VALUE of ACTIONS(state)
…
```

3.2.5 Cutting off search

As described before, alpha-beta pruning is a method used to decrease the time complexity of the game. *Cutting off search* is another method that can be used. It limits the depth of the tree by a specific amount. Each node is explored only up to the depth specified. In cutting off search, the utility function is replaced with the heuristic state evaluation function to get an estimated payoff where this is used to select which branch to follow up on [25].

This type of search is very effective when used for stochastic games which may be using expectiminimax algorithm. Since stochastic games have chance elements, as the depth of the game tree increases, the more ambiguous it can become. This is because there are factors of probability involved. Hence, cutting off the search by limiting the depth of the tree results in choosing a branch that can most probably prove to be advantageous. Even though this method is effective with stochastic games, it is also still effective for deterministic games. Games such as Deep Blue has a depth limit of 12 [25].

3.2.6 Machine learning

Machine learning allows computers to learn from situations, preparing them to act without someone telling them how to act (by programming them). Machine learning evolved from AI [26]. These algorithms overcome following the strict program instructions by taking decision based on data.

Machine learning algorithms can be classified into three categories of learning including *Supervised learning*, *Unsupervised learning*, and *Reinforcement learning*. Supervised learning is trained using training data sets. This training data consists of example pairs which are the input and the output. The supervised learning algorithm analyses this training data to produce a function which can be used to produce outputs depending on the inputs of the algorithm.

Unsupervised learning is the opposite of supervised learning where no training data is given. This algorithm tries to define hidden structures into a classification or a categorisation from unlabelled input data [27]. There also exists a type of learning that lies in between supervised and unsupervised learning called *semi-supervised learning*. Semi-supervised learning uses unlabelled training data sets where some data are labelled, however, mostly consists of unlabelled data.

Reinforcement learning is concerned with how the algorithm must act in an environment. It is informed when the action is wrong; however, the wrong actions are never explicitly corrected. This results in the algorithm trying all possibilities to find the correct output. To find a possibility, reinforcement learning uses exploration of new areas and exploitation of the current knowledge [28]. This makes this learning very general and hence, is often used within many other disciples including game theory.

3.2.7 Adaptive Game AI

Adaptive Game AI [29] is the adaptation to the behaviour and the patterns of the opponent by analysing their decision strategies to be able to predict their behaviour within a game. This ability to predict the next move of the opponent is especially advantageous within a simultaneous game because the next move of the player can be selected to gain the maximum payoff. For example, in a Rock-Paper-Scissors game, if the opponent is predicted to play Rock, the player can play Paper. The AI won't win all the time, however, by learning and adapting to the opponent's decision strategies, the AI can perform better than chance which increases the probability of the AI player winning the game.

This type of AI can be used to predict the behaviour of the opponent and can be used to modify the difficulty of the game to match the player's skill level which makes the game more interesting for the player.

3.3 Implementation Language

An important decision that must be made before the implementation of the project is the decision regarding the implementation language that is to be used. Many languages are good for this task; hence, well-known languages must be compared with each other.

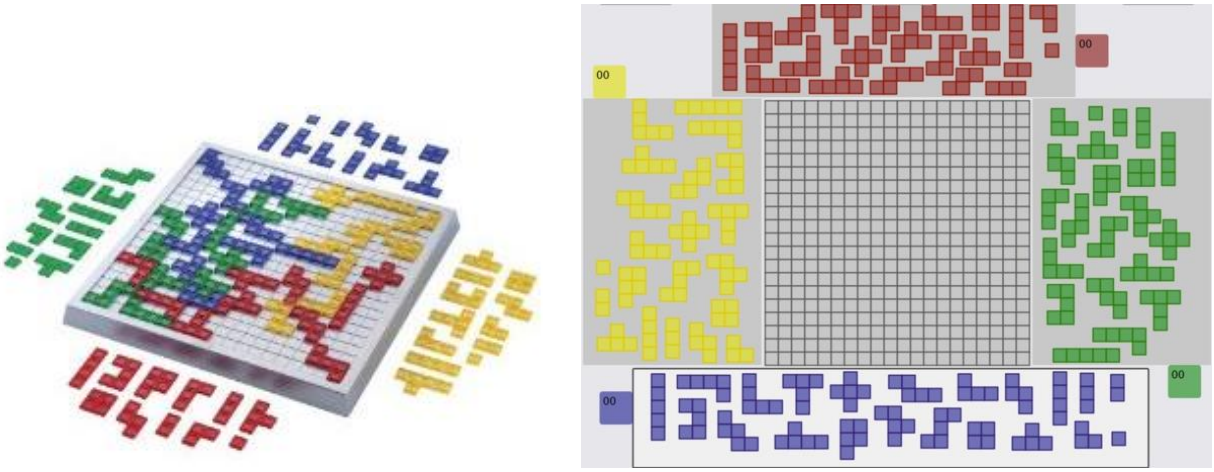
The language that I have chosen for this problem is Python. This is because it is simple, powerful, and effective. This project will also require an interface that is more advanced than text interface to meet Objective 3. Python provides a nice and easy to use Graphical User Interface (GUI) called *Tkinter*.

Low-level languages such as C/C++ were not chosen for this project because they do not offer garbage collection. For C/C++, it is required that memory must be managed within the code and this could be error prone; hence, automatic memory management is important since the game tree may be large. The disadvantages of C/C++ include error prone, slow to write, and frequently unreadable; the advantages of Python are easy to write, readable, and error reduction [30]. These advantages of Python are very important in this project.

Both Python and Java are object orientated, and they both have automatic garbage collection. However, I felt that Python is better to be used for this project because it is very concise and easy to write and read.

3.4 Blokus Duo

The game that I have chosen for my project is called *Blokus [31]* which is a two to four player game, shown in Figure 3.5. However, for the implementation of this project, I have chosen the game called *Blokus Duo* which was chosen for the simplicity of this project. This game is two player version of Blokus that is played on a 14x14 board which has 196 squares. Each player begins the game with 21 pieces. The size of the pieces' range from 1 to 5 tiles in a piece where each piece has a unique shape, as shown in Figure 3.6. A tile is a square within a piece, and a piece consists of a total number of tiles that range from 1 to 5. Each player is distinguished by the colour of their tiles which is unique to each player. Each set of pieces may be blue or red. A piece can be placed on the board in any orientation by flipping or rotating the piece, as shown in Figure 3.7. The game objective is to place as many tiles on the board as possible; therefore, it is considered better to start off with the 5 sized pieces. The ending condition of the game is that neither player can make a legal move. If one player out of the two cannot make anymore moves, however, the other player still can, then the game can continue until both the players are unable to play anymore moves.



(a) (b)

Figure 3.5: Showing the Blokus board game. a) shows a game of Blokus. b) shows an empty board with the pieces of all player

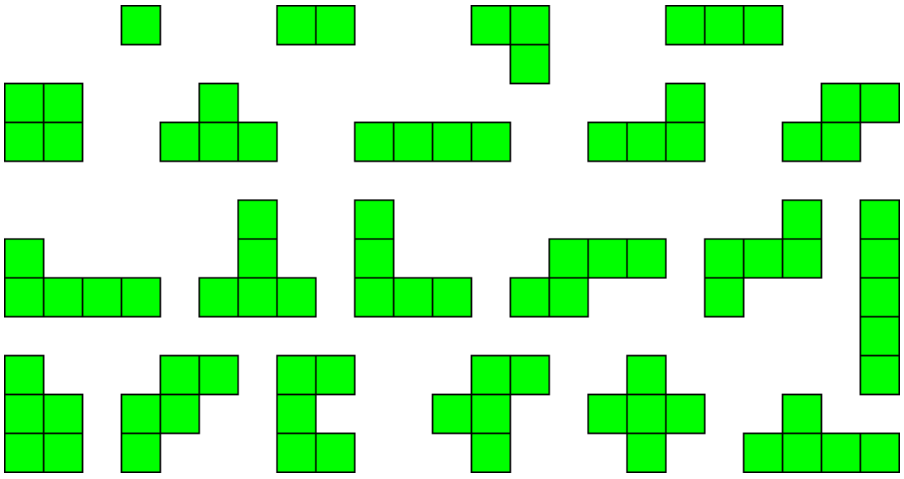


Figure 3.6: Pieces in the board game Blokus [1]

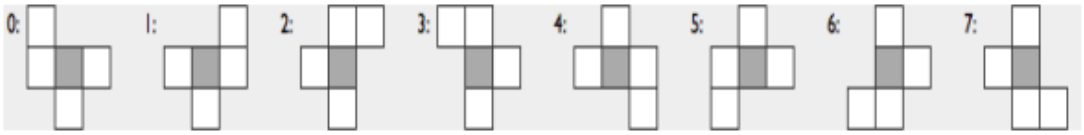


Figure 3.7: 8 different rotations of a piece in the board game Blokus [4]

3.4.1 Rules

The order of the game is follows: red, blue. A move is legal, if:

1. The first move played by each player covers a corner square on the board. Either the top-left corner or the bottom-right corner of the board.

2. The new piece placed on the board by the player must be touching a corner of at least one of the other pieces of the same colour.
3. Pieces of same colour cannot touch along the edges.
4. The new piece being placed does not overlap another piece that is already on the board.

3.4.2 Scoring

The scoring of the game is given by computing the total number of tiles of the unplaced pieces of each player. This is done because it's easier to count these tiles than to count each tile placed on the board. The player with the lowest calculated score wins because this means that they have more tiles placed on the board. A player may get a bonus of 15 points to take away from the overall score if he/she has played all 21 pieces.

3.4.3 Winning Strategies

Here are a few commonly played winning strategies that could be used within the game:

1. At the start of the game, place tiles to move to the middle of the board to be able to take up as much space on the board as possible later.
2. Place the largest tiles on the board first as it might be harder to place big tiles on the board later due to less space.
3. Keep one or more means of escape on each side of the area containing the specific colour.
4. Try and block the opponent by covering their most advanced corners to prevent them from moving forward.
5. Keep a note of the squares where no other player can play and keep these spaces in reserve while playing in a more exposed area.
6. Always keep an eye on your remaining pieces and the remaining pieces of your opponent.

3.4.4 Classification

Game classifications were described in Section 3.1.3. These are the classifications of the board game Blokus Duo:

- *Two-player*: This game can be played with two to four people. However, for simplicity, two player game was chosen.
- *Sequential*: Each player takes turns to play where each player waits on the other player to play their move.

- *Deterministic*: There are no chance elements in this game. Every possible move of a player is affected only by the previous moves made by the player or the opponent.
- *Perfect Information*: Each player has full knowledge about the current and the previous states of board. They also have knowledge about which pieces each player has left to play.
- *Non-Zero Sum*: The gain of a player does not affect a loss for the other unless the game is played in such a way that a corner of the opponent is blocked. Also, it can be made possible to place all the pieces of both the players on the board if the board size is increased.
- *Non-Cooperative*: If the game was cooperative then this defies the purpose of the game to try and defeat the other player.

3.4.5 Approach

The technique that will be used for this game implementation will be minimax applied to a heuristic state evaluation at a specific depth limit of the game tree. This will be used by the AI player to decide which move to play next which will increase the player's points and also may decrease the opponent's possibilities of gaining points. Due to computational limits, a depth limit will be chosen after testing to find an efficient limit that doesn't take too much time and also, gives a good search. Since a depth limit is being used, a heuristic function will also be used to assess states at the limit of the game tree that will be the input for the minimax algorithm. The minimax algorithm was chosen because this allows the AI player to look at the opponent's best move and tries to find the best and maximum payoff possible by minimising the maximum loss.

Initially, the approach of the program will be to use a basic approach which uses random decision-making rules. Random decision making is very easy to implement, which involves choosing any of all the possible moves. A simple heuristic approach will then be taken to select a move which uses a greedy decision making rule which may be based on a few things such as: choosing a move that places a piece to add the maximum number of tiles, or choosing a move that places a piece to increase the maximum number of open corners.

Later, the approach will be focussed on involving minimax to create the game tree recursively until the specified depth limit and then apply the heuristic state evaluation function to the leaves of this partial game tree. Different heuristic state evaluation functions can be designed and combined to give importance to different features of the game. Functions can be created to select a move that decreases the corners of the opponent,

increases the corners of the player, or include both; also, can increase the number of tiles on the board, etc.

The greedy approach and the different heuristic state evaluation functions will all be tested against each other by creating two AI players where both use different heuristics. This allows the comparison of these approaches to understanding which strategy or heuristic wins the most. This test will find the most effective heuristic function and this will be chosen for the AI player to use to play against human players.

4 Design

This chapter will describe the details of the design stage of the project, which defines the workflow for the implementation of the project and continues to describe each step required to meet the objectives of the project. This section will use the research from Chapter 3 and will form a base for the implementation stage.

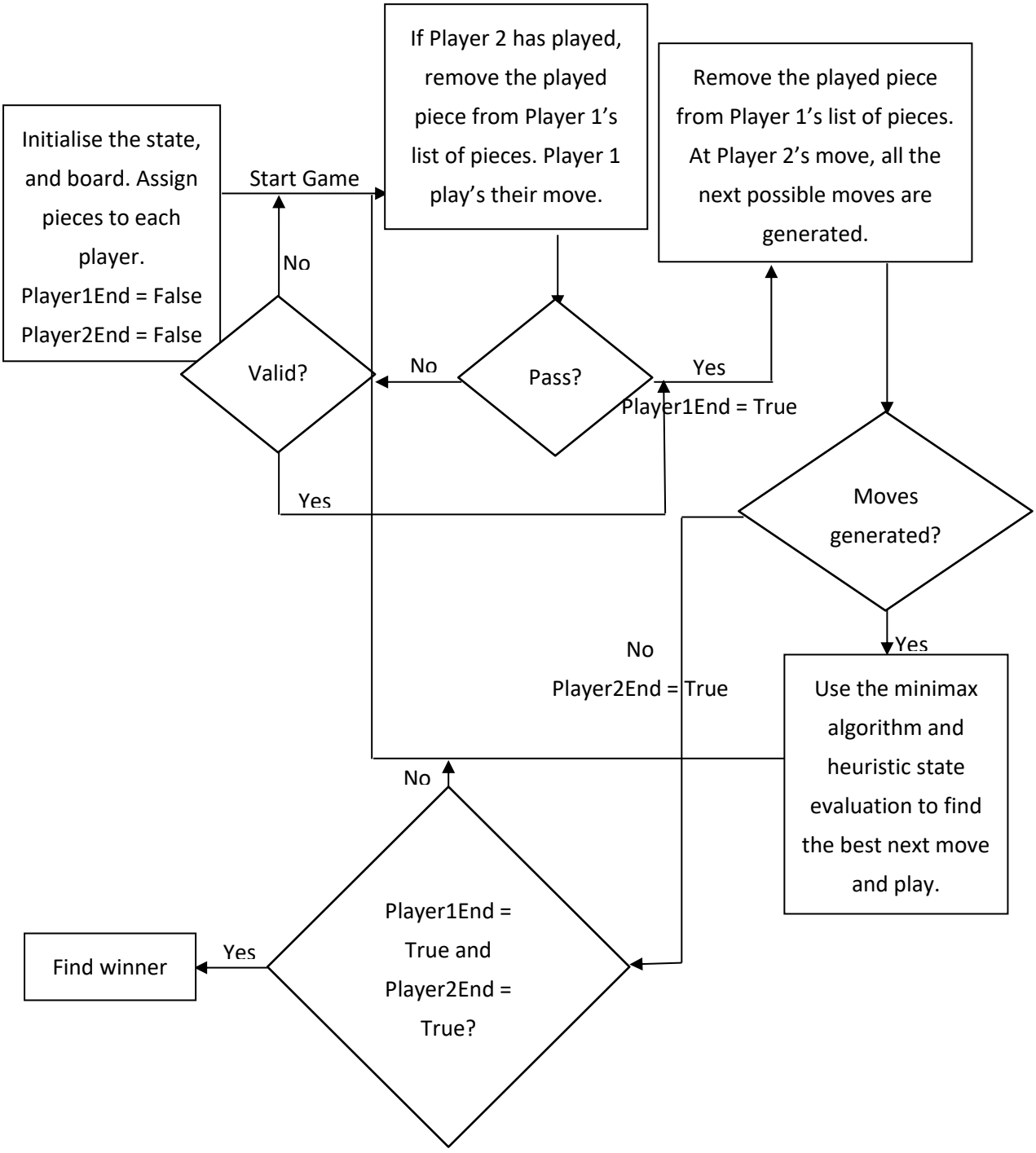
4.1 Workflow

It is important that the main workflow of the program be designed before the implementation. The AI techniques to meet the objectives of the project were researched and described in the background research conducted. In the background research, it was decided that the AI technique that is going to be used will be minimax and heuristic state evaluation. The workflow of the program will include these techniques chosen. It will describe the logic of the game program.

Player 1 will be playing red coloured pieces, and Player 2 will be playing blue coloured pieces. It is assumed that Player 1 is a human player and Player 2 is an AI player. The AI techniques are only used by the AI player to play a move at each turn. Initially, all players have all the pieces, and the board will have no pieces placed on it. The stopping condition of the game is when both Player 1 and Player 2 have no more moves to play. If a player has moves to play and the other doesn't, the game still goes on. Therefore, there are two Boolean values, both initialised to False, used to end the game: Player1End and Player2End; where each of them indicates whether Player 1 has played all their possible moves and whether Player 2 has played all their possible moves, respectively. As per the order of the turns, Player 1 will play first. Once Player 1 has made their move or has passed the move, Player 2, the AI player, will analyse the board to find all the possible moves. These possible moves will all be analysed using the minimax algorithm recursively, and the heuristic state evaluation function will be applied to determine the next best move at the depth limit. If Player 2 cannot find any possible moves, then the Boolean value Player2End is assigned to True. When Player 1 passes a turn, the Player1End Boolean value is assigned to True because it is assumed the Player 1 will be playing perfectly. Therefore, if Player 1 finds any possible moves to place, then they would have played it; hence, it is assumed by the program that there are no more possible moves that can be made by Player 1. After Player 1 passes a turn, and if Player 2 finds possible moves, then once Player 2 has played their turn, Player 1 will again get a chance to play their turn, and they can pass the

move again if no moves can be played. Also, after a piece is placed on the board by a player, the piece is deleted from the player's list of available pieces.

The workflow of the program is represented in the form of a *Unified Modelling Language* (UML):



4.2 Game Design

This will be the main function which runs the whole game. As specified in the workflow, the game stops only when both players cannot place anymore pieces on the board. Therefore, an infinite loop will be used and two Booleans values to denote whether each player has finished playing their turns. Once the game has ended, the scoring will be applied to determine the winner of the game. The design for the implementation of the game will be explained in detail.

Human Player

When a human player plays the game, he/she must be able to understand from the state what pieces are available to play, what pieces the opponent has, and what pieces are placed on the board currently.

The human player must be able to specify the coordinates of where the tile must be put on the board and also specify the piece and the rotation of the piece. This information will be retrieved from the player using keyboard inputs. The player will be prompted to enter the piece and its rotation, the y coordinate and the x coordinate of where to place the piece. If the chosen piece does not exist in the player's piece list, or if placing the move at the given coordinates is invalid, then the error will be made known to the player and the player will be asked to play again. If the move he/she tried to play is successful, or if the player enters 'pass' then the chance will be passed to the next player.

Initialisation of the properties

The properties of the game include the board and the pieces placed on it, player to play the next move, pieces of Player 1, pieces of Player 2, and the move number.

The board will be initialised with the specified number of squares and will be initialised to be empty. These squares will be denoted as 0 when it is empty when Player 1 has placed a tile on it, it will be denoted as 1 and for Player 2 will be denoted as 2. Each square on the board will have a unique value, and this will be denoted using a y coordinate and an x coordinate in the form, *(y,x)*. The y coordinate denotes the row on the board, and the x coordinate denotes the column on the board. Both these coordinates start from 0.

From the background research, it was understood that the state of the game is what is updated and passed to the minimax functions to apply the algorithm recursively. Therefore, the state will specify the properties of the game: current player, current board, Player 1's

pieces, Player 2's pieces, and the move number. The initial state of the game will be set to Player 1 to play first, empty board, all pieces assigned to Player 1, all pieces assigned to Player 2, and the move number to be 0.

Representation and orientation of pieces

A piece can be placed on the board by rotating it, or by flipping (reflecting) and then rotate it again. An orientation of a piece is the position of the tiles in the piece at a specific rotation of the piece. A piece can have eight possible orientations. However, some pieces are symmetrical, hence, there will be duplicate orientations for such pieces. Some symmetrical pieces do not need to be rotated or flipped, and some do not need to be flipped but must be rotated. Therefore, these features of a piece will be specified using a Boolean, so there is no unnecessary computation made. Also, it is important to ensure that there are no duplicates because this could cause duplicate possible orientations of a piece which may increase computation time.

Each tile in a piece must be uniquely identifiable, and therefore, the tiles will be defined using a pair of a y coordinate and an x coordinate in the form *(y,x)*. The coordinates start from 0; for the y coordinate, as the tiles go down, the y coordinate increments by 1; for the x coordinate, as the tiles go to the right, the x coordinate increments by 1. Figure 4.1a shows an example of a 5-tile piece for which the coordinates are: (0,1), (0,2), (1,0), (1,1), (2,1).

Hard-coding the coordinates of all eight orientations of an un-symmetrical piece is not the best solution. To present these pieces, they will be represented in a matrix form as shown in Figure 4.1b. The 1 in the matrix represents a tile and 0 represents no tile in the piece. The coordinates of the piece will be generated by iterating through the matrix of the specific piece, and when it finds a 1, it stores the coordinates of that position. To compute the rotations of a piece, iterating through the matrix with different orientation of the matrix gives the rotations of the piece. To compute the coordinates in the current orientation, the iteration through the matrix can be done from left-to-right; to compute the three other orientations of the piece, the iteration through the matrix will be done top-to-bottom, right-to-left, and bottom-to-top. This gives all four rotations of the piece because the direction of the iteration is taken to be a row.

To flip the un-symmetrical pieces and find the rotation, it is best to hard-code the reflected matrix and then calculate the rotations as specified before. Because computation of the reflected piece is only required for some pieces and the computational time will also be reduced. This reflection matrix is presented in Figure 4.1c. Only un-symmetrical pieces

require this computation. As specified before, there will be a Boolean value to specify whether a piece needs to be reflected or not.

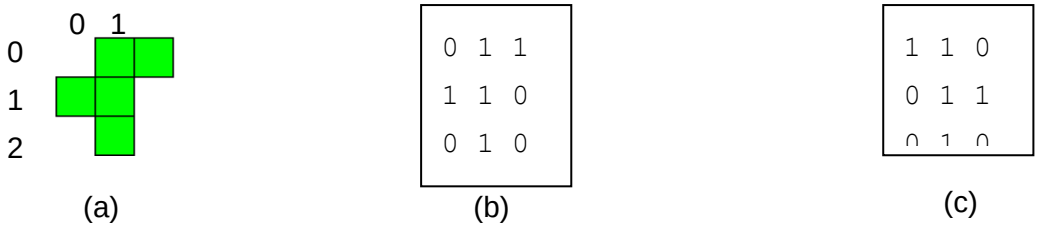


Figure 4.1: a) a 5-tile piece [1] b) the matrix representation of the piece c) the reflection matrix of the piece

Corners and Edges

For the player to place a piece, they must place it to touch a corner of another piece of same colour. The piece cannot be placed on the edge of another piece of same colour. Therefore, all the *open corners* and all the edges must be computed. An open corner is a corner where a piece can place a tile with no edges of another piece with the same colour around it.

To compute the open corners and the edges of a specific player, each square of the board must be looped through. If a square is taken on its own, it has four corners: top-left corner, top-right corner, bottom-left corner, and bottom-right corner. While looping through the board, these corners will be checked for on the board of each square that is populated by the player. If a potential open corner is found (the square that is a corner of a piece is empty), it will be checked whether there is a tile of the same colour on the square above, below, right, left of this found corner. If there isn't then this corner will be added onto the list of all the open corners.

The edges will be found by searching each square in the same loop that is used to find the corners. While looking through each square on the board, if the square above, below, on the right, or on the left of the current square, which is populated by a tile by the same player, is empty, then this current square is added to the list of edges.

Placing the piece

As mentioned before, a piece can have up to eight orientations. Therefore, the inputs of this function would require the piece, orientation number, the coordinates to place the tile, and the move number. It uses the move number to confirm whether the move being placed is the first move of either player because they must place the piece to populate either the top-left

or the bottom-right corner square of the board. While placing a piece, it must be checked that these rules are being met; making it a valid move. Therefore, each tile of the new piece will be tested to see if it keeps these rules and if it is placed on an open corner, whether any tiles of the piece are touching any edges, and if it is *over placing* by placing a tile in a square that is already populated. The corners and the edges generating function will be used to get all the open corners and the free edges of the board for the specific player. These details will be used to check whether it is valid to place the current piece on the board at the given coordinates.

Displaying the state

The state of the game cannot be displayed as a text interface since it would be hard for the human player to analyse the board and to understand which squares are free and the pieces left. Therefore, the state must be displayed on a canvas using squares to represent the board where each square is coloured to the tile placed on it. The canvas must also display the pieces available for both the players. Since it is a perfect information game, both the players may see each other's pieces to think about the opponent's possible moves.

Since the player needs to understand the different orientations of a piece, each different orientation will be displayed on the canvas. They will also be given a specific number to help the player choose the piece and the orientation they would like to play. As specified before the specific numbers will be decimal numbers.

The board will also specify the coordinates of each row and the column starting from 0 to the selected board size. This is so that the player may easily understand the coordinates of the board to place the piece.

4.3 AI Player Design

The AI player is the computer player which uses AI techniques to play the game. The AI player will generate the possible moves that can be made from the current state, and these moves will be analysed by the minimax algorithm using the heuristic state evaluation function. The best move is then selected by the algorithm and placed on the board.

Finding the possible moves

The possible moves are the moves that can be successfully made in a specific state by following all the rules of the game. In the list of possible moves, the same piece can be placed in different parts of the board as there may be more than one open corner. Therefore, this is a possible move. The possible move function will only be used by the AI player to find the best next move.

A piece may have more than one orientation, and the board has many corners. A piece can be placed if it inhibits any of the open corners and is not on the edge of another piece of same colour. To find a possible move, all the corners and the edges must be computed, and then all the orientations of all the current pieces of the player must be computed. If this is the first move of either player, then the list of corners will only contain either the top-left corner of the board or the bottom-right corner of the board and there won't be any edges. Using these it can be checked whether an orientation of a piece can be placed. The algorithm will loop through each orientation of a piece. For each orientation, it will calculate the corner tiles of the piece and place it on a corner on the board with the rest of the tiles from the piece being checked whether it is over placing, being placed within the limits of the board, and if the tile is not being placed on an edge. Once it has passed these checks, the move is then stored. The algorithm takes the state as the input which specifies the current player so using this, the possible moves for the player can be calculated. A possible move will be stored as, the piece name, orientation number, and the y and x coordinates.

The pseudo-code of the algorithm is described as:

```
function find_moves(state) returns all possible moves
      corners ← corners on the board
      edges ← edges on the board
      pieces ← pieces of the player from state
      possible_moves ← initialise to empty

      for each p in pieces do
            piece_orientations ← all orientations of piece p
            for each a in piece_orientations do
                  tile_corners ← all corners of piece a
                        for each c in tile_corner do
                              for each corner in board open corners do
                                    place c in corner
                                    check whether the rest of the tiles
                                    in the piece are placed validly

                                    if valid then
                                          add (y, x, move_name,
                                                orientation_no) to
                                          possible_moves
      return possible_moves
```

Minimax algorithm

As decided before, the minimax algorithm will be used with a depth limit because the number of nodes created for the whole game will be too large. The depth limit is the limit given to tell the algorithm the maximum number of levels that the game tree may search until. Once the depth limit has been reached and the game tree generated, the heuristic state evaluation function will be applied to the leaves of the tree. The pseudo-code for the minimax algorithm is given in Section 3.2.2. A few changes to the algorithm will need to be made to adapt it to the game. The algorithm will make sure that even if there are no possible moves for the next player, and there are more levels allowed to be searched as per the depth limit, then the algorithm will skip the search of next player and continue the search for current player to find the best move. Figure 4.2 shows an example of when a search is skipped for Player 2. The dotted arrow represents a skipped search. The depth limit given for this game tree was 3,

and minimax algorithm was initiated to maximise the payoff for Player 1. After Player 1's search, Player 2 has no more moves, and because the depth limit allows the search of Player 1 again, it skips Player 2's search and searches for Player 1 to maximise it's payoff. Figure 4.3 is like Figure 4.2; however, it shows how Player 1 has no more moves to play. Even though there are more depth limit's, the search is terminated after two searches. This is because Player 2 cannot do another search since the number of levels left to search is too low.

Various heuristic state evaluation functions will be compared and analysed in the testing phase to find the best heuristic.

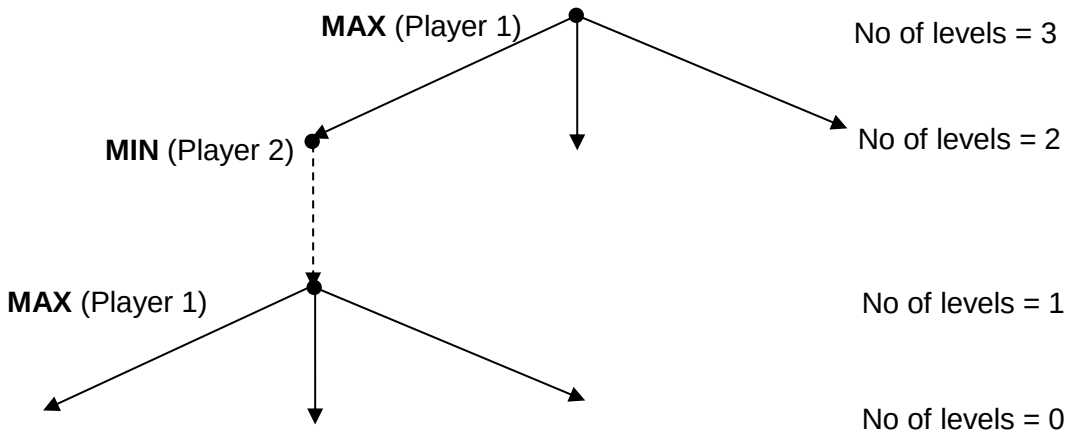


Figure 4.2: A partial game tree example of when a level is skipped in minimax

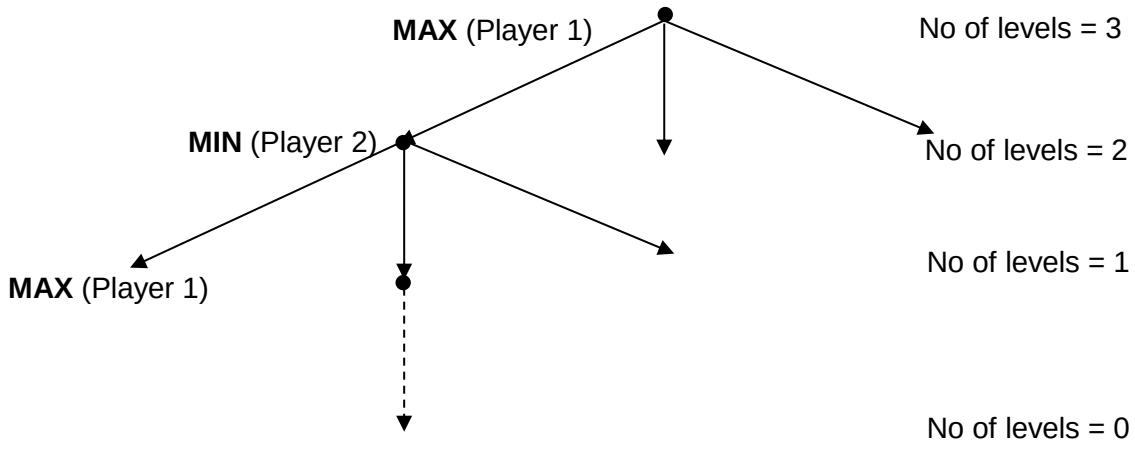


Figure 4.3: A partial game tree example of when a level is skipped in minimax algorithm.

5 Implementation

The implementation is the main stage of the project where the research conducted and all the information collected is applied and developed. This stage is followed by the design stage where the implementation was designed. It will be based off the pseudo-codes and the design from the design stage and the background stage. This chapter will explain and describe how the implementation of the game was done on the design specified in Chapter 4 and how each part was integrated together to create the final program to meet the objectives of the project.

As discussed and decided in Section 3.3, the programming language that will be used to implement the program is Python. This chapter will follow the workflow from the design chapter. The representation of the properties of the game will be discussed first and then, how the game itself was implemented to allow two human players to play against each other. Then, it will describe how the functionality for an AI player to play the game was implemented.

5.1 Game Implementation

As described in the design stage, this function is the main function that runs the whole game. Two Booleans are initialised to `False` which indicates whether Player 1 and Player 2 has finished playing their moves. An infinite loop is done where each player takes turns to play. The first move is played by Player 1, and if a move was made, then the piece is deleted from its piece list. If a move was not made and if Player 2 already finished playing its moves, then it breaks out of the infinite loop. If a move was not made and Player 2 has not finished playing, then Player 1's Boolean that indicates whether it has finished playing is assigned to True. The same logic is applied when it is Player 2's turn. After the infinite loop is terminated, the scoring function is applied to calculate the winner of the game.

When the human player (Player 1) enters 'pass', the function returns the old board and hence, the computer assumes that the human player does not have any more moves to play. However, until the computer player (Player 2) has finished with all its moves, it asks the human to play at each turn for which the human can enter 'pass' every time to miss a turn. This is the same for the computer player. If it has finished with its moves, the human player can still play until it has also finished playing all its moves. The implementation of this

algorithm is given in `blokus.py` in the Appendix. How the functions were developed to implement the game will be explained further.

Initialisation of the properties

The properties of the game were specified in the Design chapter. These properties define the state of the game. The full implementation of initialisation of the properties of the game is given in `settings.py` in the Appendix.

The initial board is set to be empty by every square on the board is set to 0. As specified in the design, 0 in a square represents that the square is empty, 1 represents that Player 1 has a tile placed in the square, and 2 represents that Player 2 has placed a tile in the square. Variables such as the `BOARD_SIZE` was initialised to be 14 as that is the typical board size of the Blokus Duo game. A two-dimensional list of `BOARD_SIZE` is used to represent the board. The y coordinate specifies the position of the list for the row and the x coordinate specifies the position of the square within the board in the column.

As was specified, the state of the board is a 5-tuple where the initial state was given arguments such as Player 1 to play first, the initial board that was created, the piece list of Player 1 and 2, and finally the move number is set to 1 as the initial state will have the details to play the first move of the game.

There are 21 pieces for each player to play. Each piece is given a unique name, and the list of the names of pieces are assigned to each player's piece list; shown in Figure 5.1.

```
piece_list = [ 'unit', 'pair',
               '3line', '3v',
               '4line', '4el', '4tee', '4sq', '4z',
               '5line', '5el', '5z', '5l', '5u', '5t', '5T',
               '5v',    '5w',  '5s', '5?', '5cross' ]

piece_list1 = deepcopy( piece_list )
piece_list2 = deepcopy( piece_list )
```

Figure 5.1: `piece_list` lists the unique name of each piece. `piece_list1` is the piece list for Player 1 and `piece_list2` is the piece list for Player 2.

Each piece is described using a two-dimensional list which describes each row of the piece. The tiles are represented as (y,x), so when using a `list`, the y coordinate of the tile specifies the row. Each row specifies whether a tile is present or not, denoted using 1 or 0 respectively. The x coordinate specifies the value of the column number which is the position

of the tile within the row. Figure 5.2 shows the specification of each piece and how each piece is described. The matrix called cover is the list of rows to specify the position of the tiles in the piece. The `reflec_cover` specifies the reflected position of the tiles. The Boolean values reflection and rotation specifies whether the piece requires the computation to find all the reflected orientations of the piece and whether the piece requires the computation to find all the rotated orientations of the piece, respectively. The `size` specifies the number of tiles in the piece.

```
piece_spec = {
          ....
           '4el': {   'cover': [[1,1,1],[1,0,0]],
                   'reflec_cover': [[1,1,1],[0,0,1]],
                    'reflection': 'yes',
                     'rotation': 'yes',
                     'size': 4
                    },
          ....
          }
```

Figure 5.2: a snippet of `piece_spec` which describes a piece

Orientations of a piece

Once the representation of the properties of the game was implemented, the computation of the different orientations of a piece could be implemented. The algorithm as specified in the design will loop through the matrix of the piece from different directions to compute the rotations a piece. The full implementation of this algorithm is given in `library.py` in the Appendix.

The function takes the name of the piece as an input and retrieves the details of the piece from the `piece_spec`. The Boolean values are used to check whether the piece requires a rotation or a reflection. A `for` loop within a `for` loop was used to iterate through the two-dimensional list to find the coordinates of one orientation. In total, 4 iterations were implemented to compute the orientations by iterating from different directions. If a piece must be reflected, then this function would be used twice as much, and hence there would be 8 iterations in total. The variety in directions to calculate all 4 orientations of each side of the piece does not affect the number of times a tile is looped through, which is always once. If the matrix is a 3 x 3 matrix, then the number of times it visits a position is always 1 and the number of times it iterates is 9.

Some pieces that are symmetrical may require rotation but rather than having 4 different orientations for a face, it may only have 2 and this will create duplicates. To avoid duplicates, a `set` was used to store the coordinates of a piece which will only store unique orientations of a piece.

Displaying the state

As specified in the design chapter, the state of the game is displayed to the players using Python's GUI interface called Tkinter. The state of game displays the board and the pieces available for each player. Figure 5.3 shows how the initial state is displayed within the game. The size of the canvas was chosen to fit the 14 x 14 board and all the 21 pieces of both players with its different orientations; `CANVAS_WIDTH = 1500`, `CANVAS_HEIGHT = 800`. The size of each square on the board was chosen to be 25 pixels, and the size of each tile on a piece was chosen to be 10 pixels. Each square on the board displays a colour of grey shade if the square has value of 0, red if the square has a value of 1, and blue if the square has a value of 2. The y coordinates and the x coordinates of the board are displayed on the left side and the top of the board respectively.

Player 1's pieces are displayed in red colour, and Player 2's pieces are displayed in blue colour. Each piece and its orientations are given a decimal value. The human player will want to specify which orientation of a piece they want to place. Each orientation of a piece is given a specific number. For example, for the piece shown in the Figure 4.1a, an orientation of that piece may be called 1.1. The number before the decimal point represents the piece number and the number after represents the orientation number. The piece number will go maximum up to 21, and the orientation number will only go maximum up to 8. While displaying the pieces, they are displayed one by one on the canvas. When it reaches close to the canvas window, to avoid the pieces being displayed outside of the canvas window, it starts to display the pieces on the next line. As each piece is placed on the board, the placed piece and all its orientations are removed from the player's pieces displayed on the canvas. Even when pieces are removed from the list, it is ensured that the decimal number that is unique to each orientation of every piece, stays the same throughout the game; as shown in Figure 5.4.

The full implementation of the functions used to display the state of the game is given in `library.py` in the Appendix.

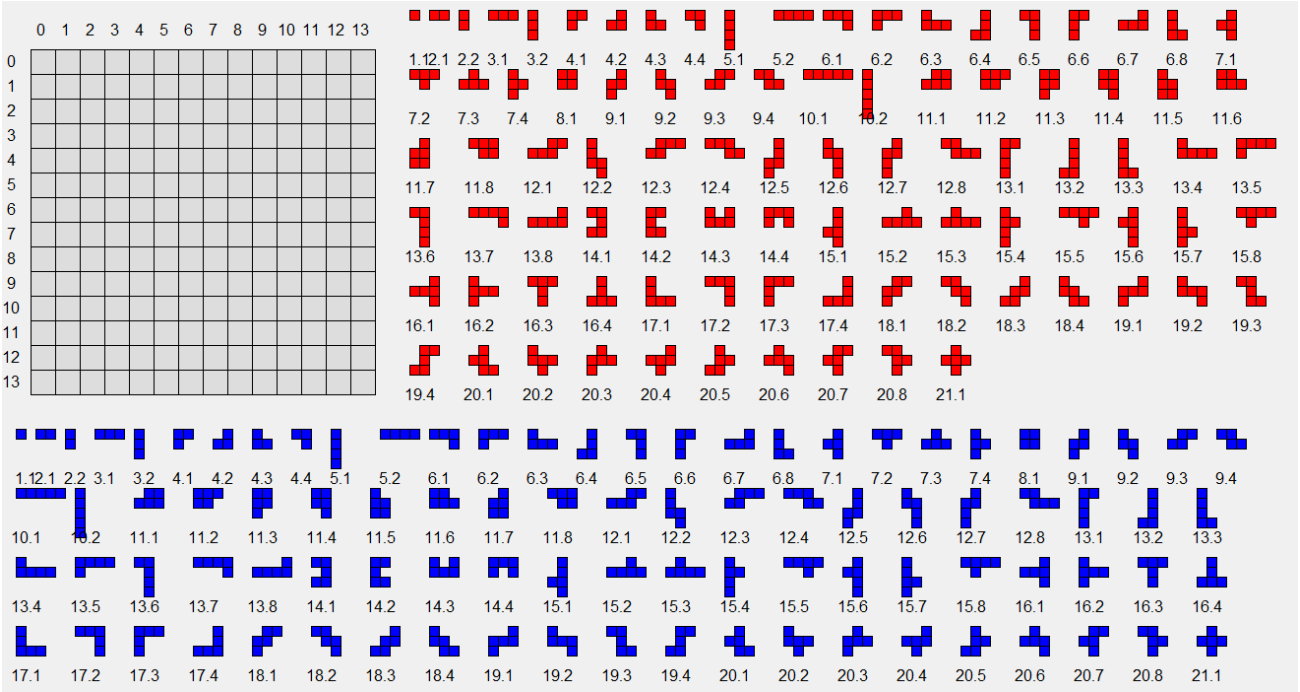


Figure 5.3: the initial state display before the game has started

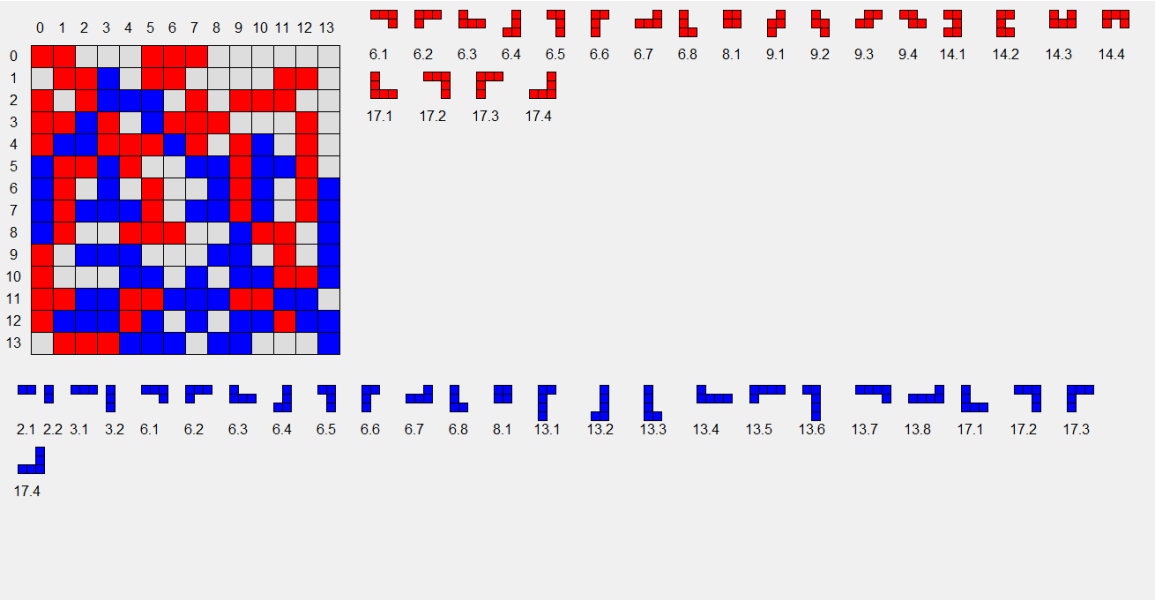


Figure 5.4: the display of the state at the end of a game

Corners and edges

The corners and the edges of the pieces on the board are computed as was described in the design. The corners and edges of the specified player are computed by looping through each square on the board. This function takes the board and the player to find the corners and the edges for and returns a 2-tuple which contains a list of corners and a list of edges. The full implementation of this algorithm is given in library.py in the Appendix.

While iterating through, at each square, if it contains the tile from the specified player, the squares on its 4 corners are then checked whether they are empty. If a corner is empty, then the edges of that corner square are tested to confirm that they do not contain a tile from the same player. If this test is passed, then this is an open corner. The edges are computed in the same iteration that is used to find the corners. At each square, if it contains a tile from the given player, then its edge squares are tested whether they are empty. If they are, then the edge square is stored because another tile from the same player cannot be placed here.

The corners and the edges are added to a `set` to avoid duplicates. A corner or an edge is only added if it is empty because if it is not empty, then a piece cannot be placed there and hence, do not have to check whether another tile may be placed there. The functions that use this function already check if a tile is being over placed. Avoiding duplication and the addition of corners and edges that are not empty decreases computation time.

Placing a piece

This function takes the piece, orientation number, y coordinate, x coordinate, and the state as the input and returns the board after placing the piece if the piece is placed successfully if not, the old board is returned which doesn't have the piece placed on it. As was specified in the design, when placing a piece, the function confirms that the piece exists in the piece list of the player that wants to place the piece. If it does, then it gets all the corners and edges on the board of the specified player and tries to place the piece on the board. It places the piece on the board by placing each tile individually after checking whether it is violating the rules of the game. It checks whether the rules 1 and 2 from Section 3.4.1, is kept. This is implemented using 2 Boolean functions which are then checked at the end of placing the whole piece if were set to `True`; if it was, then the new board is returned and if not, the old board is returned, after printing the error. It also makes sure while placing each tile that it is not being placed on an edge, if it is, then it returns the old board. If a tile is violating, then it prints out the appropriate error and returns the old board. If it is a valid move, then it places the piece on the board and returns the new board. The implementation of this algorithm is given in `library.py` in the Appendix.

The function prints out the error so that the human player may understand the error with the move that was made. The AI player does not require an error message because it is played by the computer. Also, when the AI player generates moves, it only generates moves that comply with the rules of the game.

The coordinates will be specified by the player so that the player is specifying where the top left tile (0,0) of the piece must be placed on the board. The coordinates of where the rest of the pieces will be placed can be worked out by the algorithm from these coordinates. An example of how this is done can be shown using Figure 5.5 below. Currently, it is Player 2's move to play. If player 2 wants to play piece 21.2 to cover the coordinates (8,10), (9,9), (9,10), (9,11), (10,10) then the (y, x) coordinate specified by the player for this function will be (8,9).

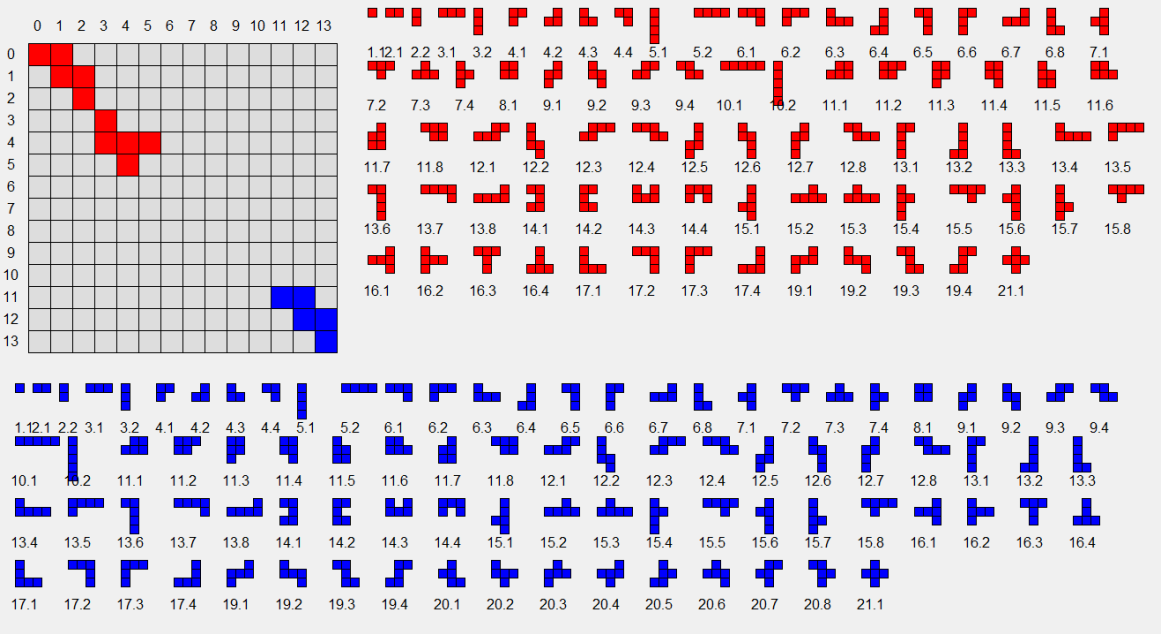


Figure 5.5: state of the game after 3rd move

Human turn

When it is the turn of the human to play, he/she is prompted by the game for the piece to play for which he/she would enter the decimal number. It then asks for the y coordinate and x coordinate separately. These are the only 3 inputs required by the human player to be able to play a piece. The decimal number entered by the player is then split by the decimal point. The piece number is used to find the piece name the user wants to place. This piece name and the orientation number along with the y and x coordinates and the state of the game is passed to the placing piece algorithm. It then tries to place this move and if successful, returns the new board. This new board is then returned by the algorithm for the human

player. If the move is not valid, the placing piece algorithm returns the old board. The algorithm of the human player then asks the human player to try again so that he/she can correct their error; this is done by checking whether the board returned by the placing piece function is same as the old board. If it is, then the player is asked to play again. If not, the new board is returned with the name of the piece that was placed.

If the human player decides to pass the game, then when asked for the piece to play, he/she may enter '`pass`' to pass the move. The function then returns the old board and an empty string as the name of the placed piece.

The implementation of this algorithm is given in `blokus.py` in the Appendix.

Scoring

The score of each player is calculated at the end of the game when there are no more possible moves by both players. Each player's un-played pieces are looped through to sum the size of each piece that is specified in the piece specification because each tile is worth 1 point; therefore, the size is the total points gained by that specific piece. If a player has played all their pieces, the sum value of the piece sizes, which would be 0 in this case, will be subtracted 15 points. This value is then subtracted from the total points a player can get by playing all their pieces, which is 89 points. So, when a player has played all their pieces, they would get a total of 89 + 15 points. If a player has played only a 5 piece, then their total will only be 5 because the total of the un-played pieces' size would be 84. The winner and each player's scores are then printed. The full implementation of this algorithm is given in `library.py` in the Appendix.

5.2 AI Player Implementation

For the AI player to make a move, it uses the minimax technique and the heuristic state evaluation. To find all the possible moves and the best next move using the minimax, there are a few steps that must be made beforehand. The minimax technique looks at all the possible moves that can be made by the player. Therefore, a function was created to generate all the possible moves of the player from the current state. After this, the minimax algorithm was implemented which uses the function to generate the possible moves of the player. The heuristic state evaluation function was also implemented to find the best move. Few heuristic functions were developed to compare them to each other to find the best

heuristic. This comparison will be explained and discussed within the Testing stage of the project. All implementations of this algorithm are given in `blokus.py` in the Appendix.

Possible moves

The possible moves are computed using the un-played pieces, and the corners of the player on the board. The input to this function is the state of the game. As described in the design, to find the possible moves of an un-played piece, all the orientations of each piece are generated, and the corner tiles of each orientation are computed. These corner tiles of a piece are placed on an open corner on the board and checked whether placing the piece there complies with the rules of the game given in Section 3.4.1. If it is valid, then the move is added to the list where the y and x coordinates are the top-left corner tile (0,0) of the piece. The pseudo-code is provided in the design section.

A possible move is represented in the form: `(4, 5, 1, '4el')`. This specifies the y coordinate, x coordinate, orientation number and the piece name.

Minimax

The minimax function is implemented from the pseudo code from the background section. However, as specified in the design section, it was modified to be adapted to the game. The minimax algorithm takes the state, the player that is trying to maximise their move, specification of whether a max or a min is required, the number of levels left of the game tree, and the move to place. The minimax algorithm places the move of the player and creates a temporary piece list for the player and removes the placed piece from this. If the current level is the depth specified, then the heuristic state evaluation is done, and the value is returned. If not, it then creates a new state with the next player to play the move and increments the move number and assigns the new piece list to it depending on the player. Then the moves of the next player are found and if the next player does not have any moves and yet still there are more than 2 levels left that can be searched, then the search of next player is skipped. When the search is skipped, it means that the current player gets to go onto its next level and the number of levels left is decremented. It then calculates all the possible moves of the current player and calls the minimax algorithm for each move which then goes through it recursively. If a search is not skipped meaning that the next player has possible moves, then the next player's moves are looped through, and the minimax algorithm is called again which will go through each move recursively. This is done until the specified depth limit at which the heuristic is returned. Details about skipping searches are further explained in the design section.

Computer Turn

The computer turn is the implementation of the MINIMAX_DECISION algorithm for which the pseudo-code was given in the background section. It generates all the possible moves of the AI player and initiates the minimax algorithm for each move. From these moves, the move with the maximum heuristic is chosen and placed on the board using the function to place the piece. The move must be a valid move; therefore, this function will not return an error. The new board that is returned by the placing piece function is then returned; and the placed piece name is also returned. If there are no moves, then the old board is returned with an empty string.

6 Testing

It is important that the best winning heuristic is chosen for the AI player of the game. Therefore, the testing phase was used to compare and determine the best winning strategy. Two AI players will play against each other where both use different heuristic for each strategy. These two players will play against each other and the most winning heuristic will be chosen to be the best heuristic. Both the AI players will be using minimax algorithm; therefore, it is important to choose a depth limit for the search of best move of each player.

6.1 Depth Limit

The minimax algorithm was executed using various depth limits to choose an efficient depth limit to use while testing each heuristic. Efficiency was measured using the time taken overall, and the total number of nodes. The higher the number of nodes, the higher the memory taken up on the machine. The results are given in Table 1 below. The depth limits were tested using a 14x14 board where the board was empty. From the results, it seems like depth 3 may be the best option to choose even though the time taken is rather large. However, it must also be considered that these tests were executed when there were no pieces on the board. Therefore, when there are more pieces, which means more corners, the time and the number of nodes will increase largely because there are more possible moves to make. When the board is empty, there is only one corner to test for possible moves. From this also, considering the computational limits of the machine, I concluded to use a depth limit of 2.

Depth Limit	No. of max nodes	No. of min nodes	Total no. of nodes	Time taken
1	0	58	58	29 ms
2	3364	58	3422	1736 ms
3	3364	577622	580986	1305771 ms
4	-	-	-	Terminated after 12 hrs

Table 1: Comparison of time taken and total no. of nodes for different depth limits

6.2 Heuristic

Each test was only executed once because it is pointless to redo a test as the result of a test will always be the same. When using a heuristic at a specific state, it will try to optimise the payoff and will end up placing the same piece. The winning heuristic of each comparison will be compared with other winning heuristics to find the best one from all. The heuristic comparisons are explained, and the results are shown below.

The heuristic tests were based on Strategy 1 and 4 from Section 3.4.3. The tests were then evolved from this to find the best solution.

Comparison 1

Player 1: Decrease corners of the opponent.

Player 2: Increase its corners.

The heuristic for Player 1 calculates all the corners of the opponent and the minimax decision of Player 1 minimises this. Hence, finding the best next move by minimising the number of corners of opponent.

The heuristic for Player 2 calculates all the corners of itself, and the minimax decision increases this.

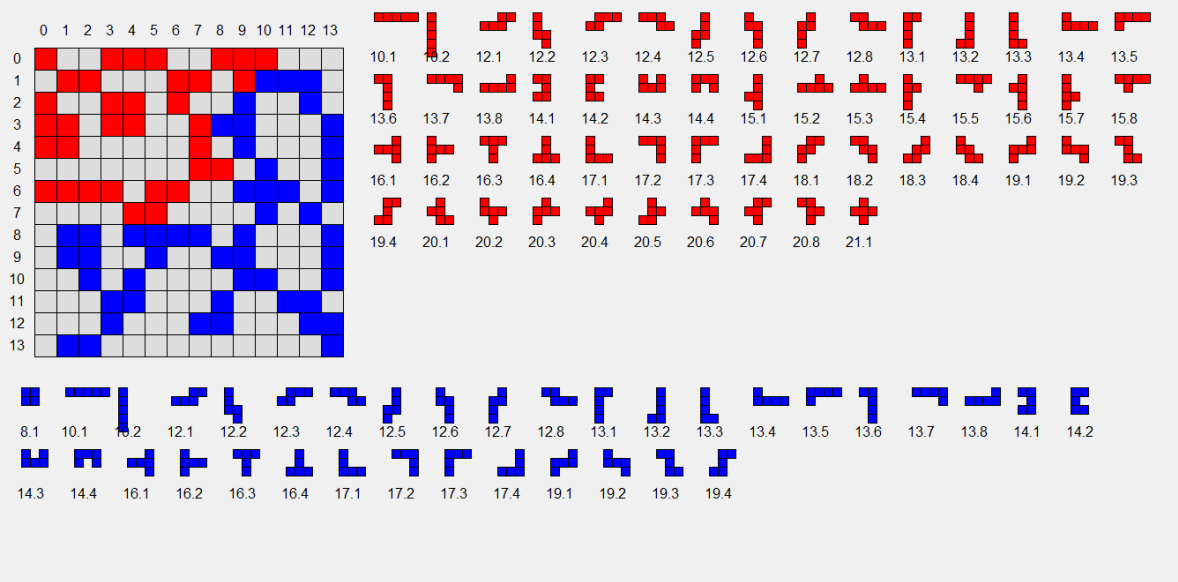


Figure 6.1: State at the end of game after comparison 1

Comparison 2

Player 1: Decrease corners of the opponent and place big tiles first.

Player 2: Increase its corners

The heuristic for Player 1 calculates all the corners of the opponent and the minimax decision of Player 1 minimises this. The function to find all the possible moves were limited to find possible moves with the biggest sized pieces available. If it cannot find any moves with biggest sized pieces, then it finds moves with lower sizes.

The heuristic for Player 2 calculates all the corners of itself, and the minimax decision increases this.

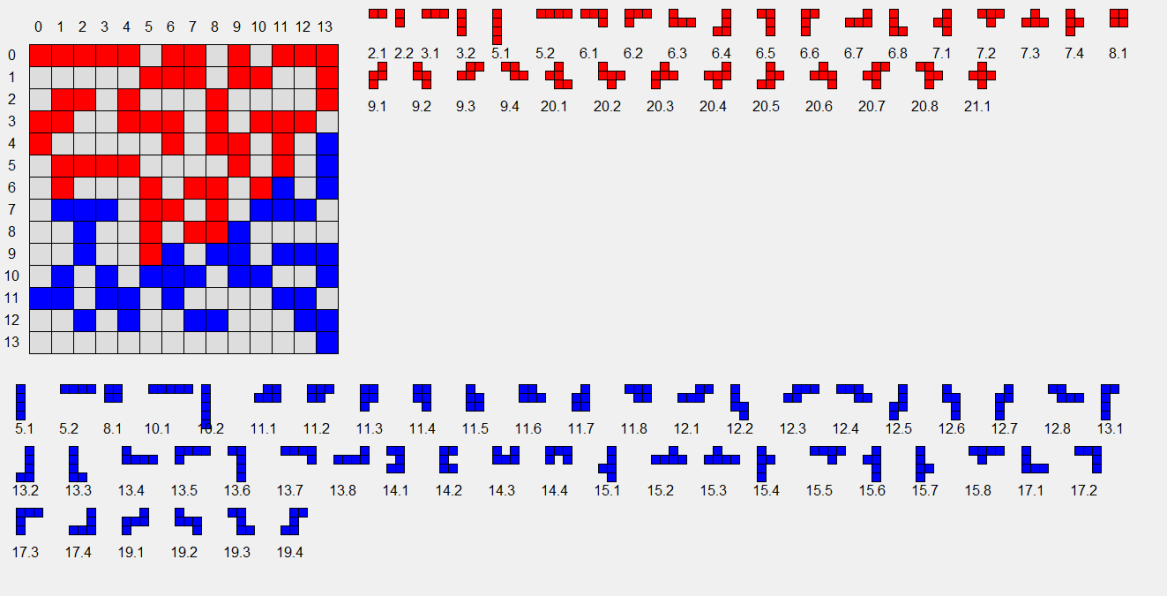


Figure 6.2: State at the end of game after comparison 2

Comparison 3

Player 1: Decrease corners of the opponent and increase its number of tiles placed

Player 2: Increase its corners

The heuristic for Player 1 calculates all the corners of the opponent and the number of its squares. The heuristic was as follows: Heuristic = (no. of tiles placed by Player 1 *2) – (corners of opponent). The minimax decision of Player 1 maximises this. This heuristic gives priority to increasing its number of tiles placed than decreasing the corners of the opponent.

The heuristic for Player 2 calculates all the corners of itself, and the minimax decision increases this.

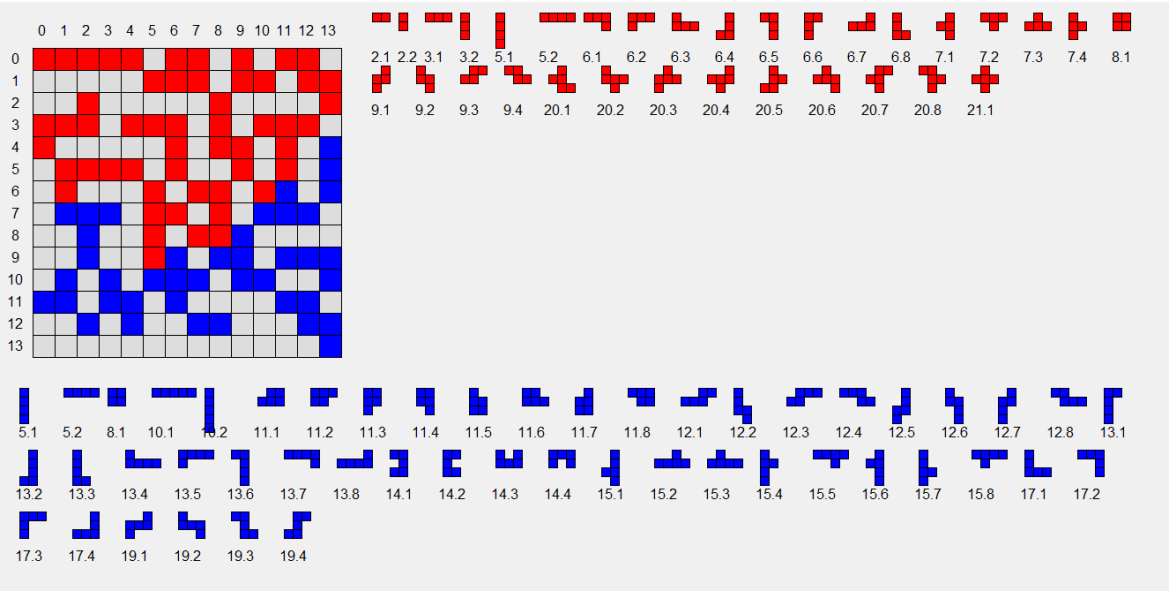


Figure 6.3: State at the end of game after comparison 3

Comparison 4

Player 1: increase its corners and place big tiles first.

Player 2: decrease opponent's the corners and place big tiles first

The heuristic for Player 1 calculates all the corners of itself, and the minimax decision of Player 1 maximises this.

The heuristic for Player 2 calculates all the corners of the opponent and the minimax decision of Player 2 minimises this.

The function to find all the possible moves for both players were limited to find possible moves with the biggest sized pieces available. If it cannot find any moves with biggest sized pieces, then it finds moves with lower sizes.

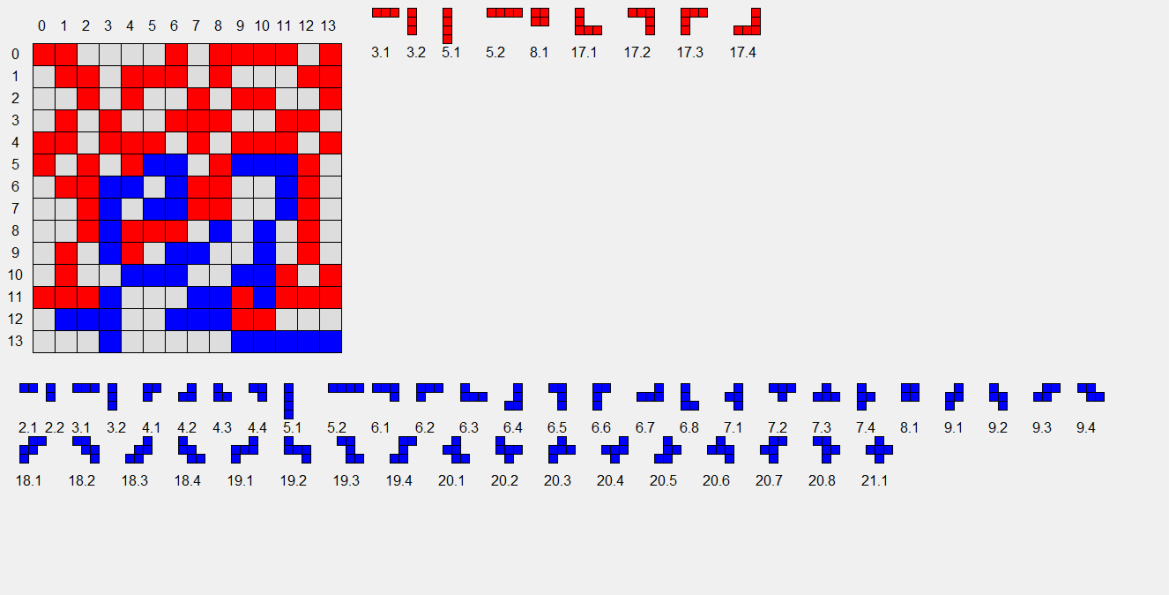


Figure 6.4: State at the end of game after comparison 4

Comparison 5

Player 1: place big tiles first and decrease the number of tiles placed by opponent.

Player 2: place big tiles first

The heuristic for Player 1 calculates all the tiles placed by both players. The heuristic is as follows: Heuristic = (no. of tiles placed of Player 1) – (no. of tiles placed by opponent *2). The minimax decision of Player 1 maximises this. This heuristic gives priority to decreasing the number of tiles placed by the opponent.

The heuristic for Player 2 calculates all the tiles placed Player 2 and the minimax decision function tries to maximise this.

The function to find all the possible moves for both players were limited to find possible moves with the biggest sized pieces available. If it cannot find any moves with biggest sized pieces, then it finds moves with lower sizes.

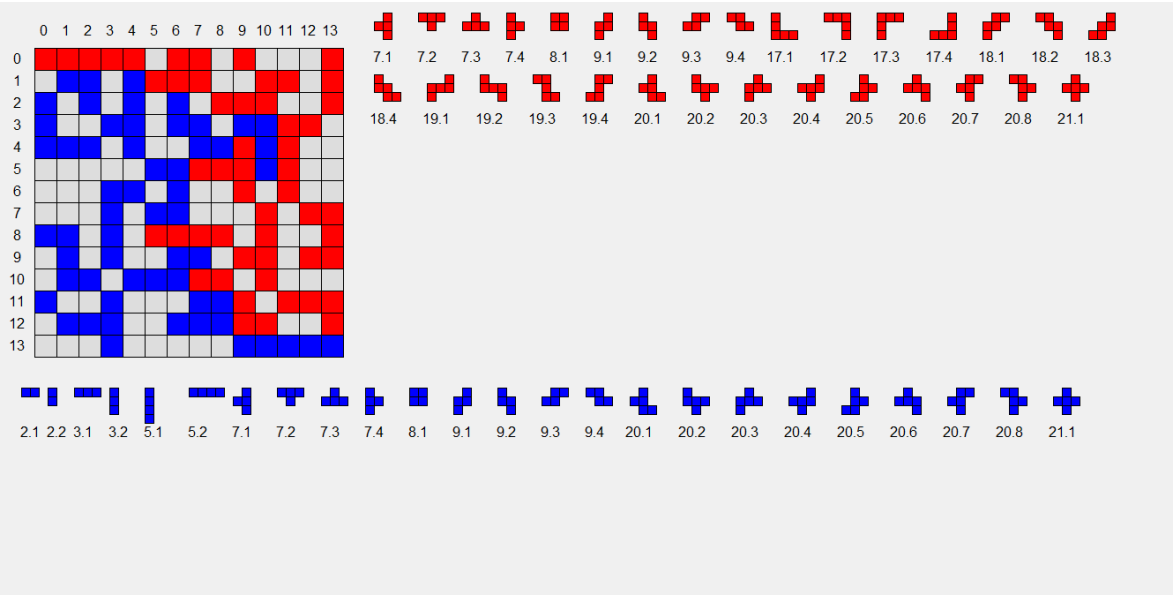


Figure 6.5: State at the end of game after comparison 5

Comparison 6

Player 1: increase its corners and place big tiles first.

Player 2: decrease the opponent's corners, increase its corners, and place big tiles first

The heuristic for Player 1 calculates all the corners of itself, and the minimax decision of Player 1 maximises this.

The heuristic for Player 2 calculates all the corners of the opponent and its corners. The heuristic is as follows: Heuristic = (no. of corners of Player 2 *2) – (no. of corners of opponent). The minimax decision of Player 2 maximises this. This heuristic gives priority to increasing its number of corners.

The function to find all the possible moves for both players were limited to find possible moves with the biggest sized pieces available. If it cannot find any moves with biggest sized pieces, then it finds moves with lower sizes.

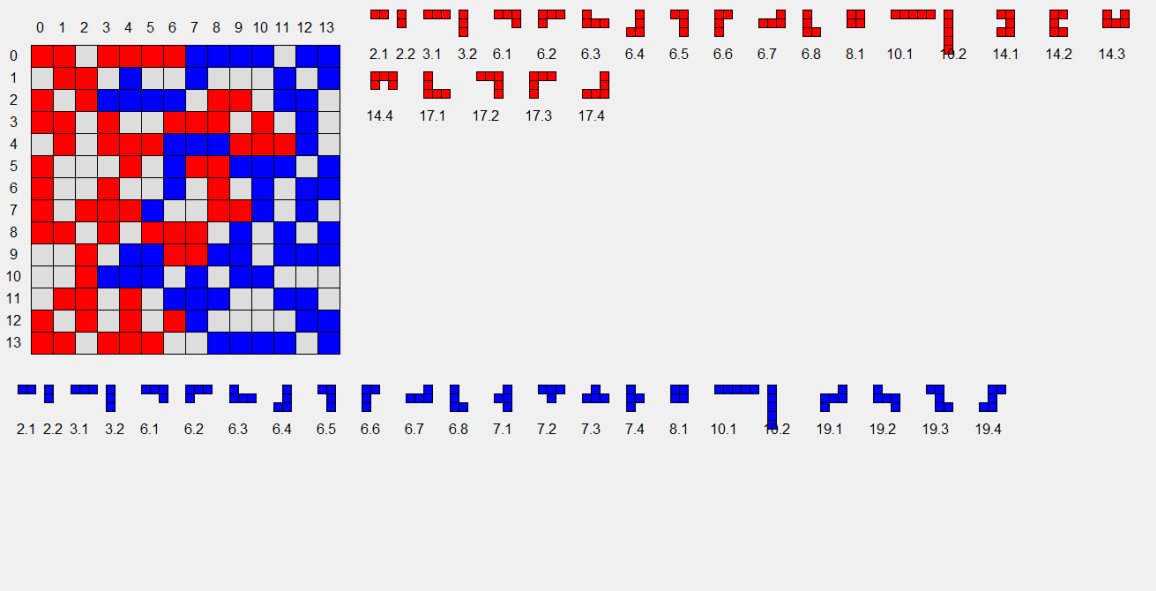


Figure 6.6: State at the end of game after comparison 6

Comparison 7

Player 1: decrease the opponent's corners, increase its corners, and place big tiles first

Player 2: increase its corners and place big tiles first.

The heuristic for Player 1 calculates all the corners of the opponent and its corners. The heuristic is as follows: Heuristic = (no. of corners of Player 1 *2) – (no. of corners of opponent). The minimax decision of Player 1 maximises this. This heuristic gives priority to increasing its number of corners.

The heuristic for Player 2 calculates all the corners of itself, and the minimax decision of Player 2 maximises this.

The function to find all the possible moves for both players were limited to find possible moves with the biggest sized pieces available. If it cannot find any moves with biggest sized pieces, then it finds moves with lower sizes.

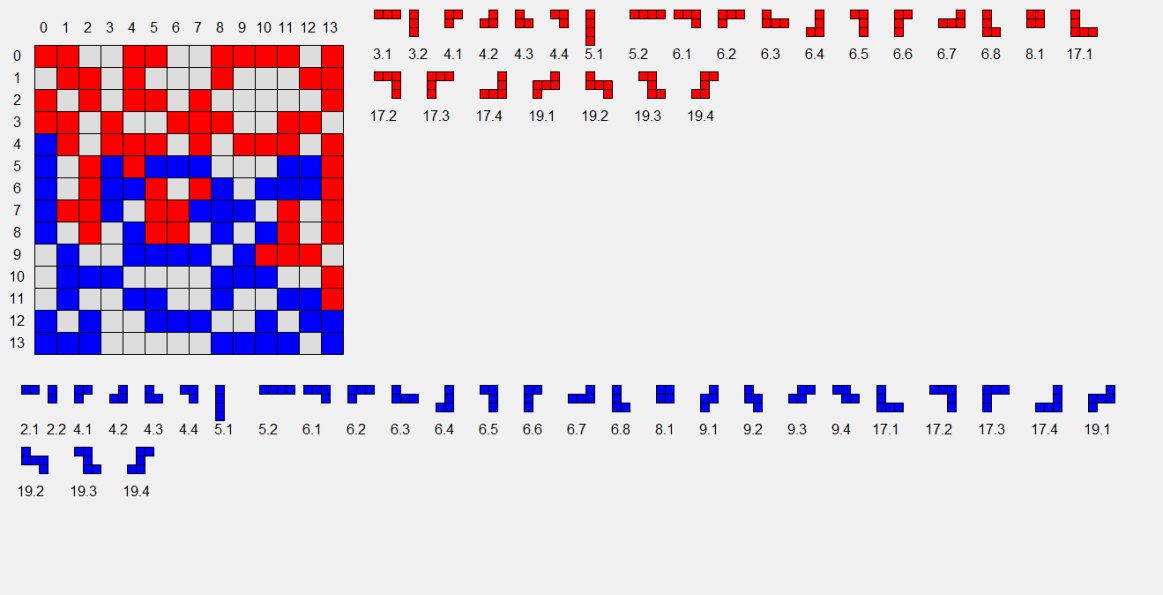


Figure 6.7: State at the end of game after comparison 7

6.4.1 Results

The results of each comparison are shown below in Table 2. The player that won each game is coloured in green.

Comparison	Player 1	Player 2	Total no. of nodes	Time (ms)	Scores
1	Decrease corners of opponent	Increase its own corners	1,648,437	933,704	Player 1 = 34 Player 2 = 50
2	Place big tiles first and decrease corners of opponent	Increase its own corners	783,135	425,638	Player 1 = 54 Player 2 = 41
3	Increase (no. of tiles placed * 2) and decrease corners of	Increase its own corners	1,170,894	1,470,371	Player 1 = 54 Player 2 = 41

	opponent				
4	Place big tiles first and increase corners	Place big tiles first and decrease corners of opponent	575,178	1,060,542	Player 1 = 73 Player 2 = 41
5	Place big tiles first and decrease (no. of tiles placed by opponent *2)	Place big tiles first	325,142	542,051	Player 1 = 52 Player 2 = 58
6	Place big tiles first and increase no. of corners	Place big tiles first and increase (no. of corners *2) and decrease no. of corners of opponent	755,392	1,697,035	Player 1 = 61 Player 2 = 62
7	Place big tiles first and increase (no. of corners *2) and decrease no. of corners of opponent	Place big tiles first and increase no. of corners	737,601	1,214,382	Player 1 = 61 Player 2 = 58

Table 2: Results of each comparison

7 Evaluation

The evaluation stage is mainly used to analyse the test results to determine the best heuristic. This section will consist of the discussion on the test results, and also on the project objectives to evaluate whether the project has correctly met the objectives.

7.1 Testing results

While testing, each test was evolved from the previous one to try and make it better or to test whether different conditions will change the results. After the first few tests, it was obvious that placing the big pieces first was a strategy that helps towards gaining points. Comparisons 3 and 4 are very similar where the only difference was that one limits the pieces when finding possible moves and the other does not. Both try to increase number of tiles placed on the board in a move. From Figure 6.3 and 6.4 it can be seen that the pieces used are the same but just have some variations on the positions they were placed. Therefore, due to the speed of the game, limiting the possible moves were chosen to be a better option from henceforth.

In comparison 6, the scores for both heuristics were very close. Therefore, the heuristics were swapped for comparison 7 regarding who starts first to see if this made a difference in the winner. From the results of comparison 7, the same heuristic has won the game with better results. Therefore, from the results, it concluded that the heuristic ‘Place big tiles first and increase (no. of corners *2) and decrease no. of corners of opponent’ will be used for the AI player. This AI player will play the first move of the game as this seems to give the best results.

7.2 Project Objectives

This section evaluates whether the project objectives specified in chapter 2 has been met. The list of the objectives and the discussion is as follows:

- Objective 1: *Develop a software capable of correctly playing the game of Blokus in accordance with its rules*

  During the general testing of the implementation phase, it had been obvious that the program is correct as per the rules of the game. This belief was further strengthened during the testing phase of the project when the moves were chosen by the program rather than a human where there is a higher chance of moves played by the program

that is against the rules coming to light. However, there didn't seem to be such things; therefore, this objective has been met.

- Objective 2: *Develop a simple rule-based AI algorithm to play the game of Blokus*
  The implemented program uses AI techniques such as minimax algorithm and heuristic state evaluation to play the game Blokus. Therefore, this objective has been met.

- Objective 3: *Develop a GUI interface to show the game states*
  The implemented program displays the state of game using a Python GUI interface called Tkinter. This is described and shown in Section 5.1. Therefore, this objective has been met.

- Objective 4: *Investigate the use of AI techniques such as Minimax, Heuristic state evaluation, Machine learning to enable the algorithm to play 'intelligently'*
  During the research phase, various AI techniques were investigated including Minimax, Heuristic state evaluation, and Machine learning; some of which has been used within the project. They were researched about, and the techniques that seemed best suitable for the project and for the given timescale of the project were chosen. Therefore, this objective has been met.

7.3 Project Evaluation

Overall, the project went very well since it has been able to meet the aim to investigate and implement the use of AI techniques and produce a program that 'intelligently' plays Blokus. Every stage of the project had no issue since any issues that arose were able to be sorted out with some further research into the matter. However, a challenge faced would be the computational time of the minimax algorithm and the time it took for a game to finish while testing, either in the implementation phase or in the testing phase. This led to some confusion whether the algorithm was correct or not. When it took more time to search for a move than I expected, I suspected a possible bug. There wasn't a bug, but it was just the computation limits of my machine. However, debugging was done to further investigate the program and to solve any bugs that were found. The results of the investigation were then added into the testing section of the project and was used to choose a good depth limit. If there was more time available, I would have been able to do more extensive tests of different heuristics to possibly find a better solution to the current one found.

Using AI techniques does meet the aim to play Blokus intelligently; however, I would have preferred to add some machine learning elements into the program to be able to make it more intelligent. If there was more time available for the project, I am sure this would have been possible. Also, in the time given, I was able to meet all the objectives and keep to the project plan; therefore, I can say that the time management of the project was very good.

8 Conclusion

The aim of the project was to investigate and implement AI techniques to play Blokus ‘intelligently’. Initially, a background research was conducted with the aim to give an idea about the project and to understand the problem and how this could possibly be tackled. Through this research, the aim was broken down into project objectives to make meeting the aim an easier task. There were four different stages to the development stage: design, implementation, testing, and evaluation. The design was based strongly upon the background research conducted and information gathered. It was used to shape the implementation stage and create an outline of how to implement the solution to meet the objective. The implementation stage was where the knowledge upon the project was applied. The testing and evaluation stage was used to investigate on the best solution to play the game as ‘intelligently’ as possible. Finally, each of these stages was a step towards meeting the aim of the project.

8.1 Future Work

There are many future improvements and developments that can be implemented for this tool. They were not attempted in this project due to time constraints. Therefore, these will be considered in the future work of this project.

8.1.1 Test more strategies to find a better heuristic

In Section 3.4.3, it specifies a few strategies that will help towards winning the game. Strategy 2 and 4 were used to explore a good heuristic within this project. Therefore, a possible improvement for this project would be to implement and test the strategies 1, 3, and 5. This includes implementing the knowledge into the program where the minimax algorithm it can determine where it the player has placed pieces on the board and from this work out the next best move.

8.1.2 Adaptive AI

Currently, the AI player only uses its ways and uses its own heuristics to try and predict what the opponent may play. However, the opponent may not be using the same heuristic as the player. Therefore, a possible improvement would be to understand how the opponent plays

it’s moves by determining their heuristic and to include this heuristic in the minimax algorithm when playing as the opponent in a specific level. This will give a better chance of winning.

8.1.3 Add more players

Currently, the game only allows 2 players in the game. The game Blokus can have from 2-4 players in a game. A possible improvement for the program would be to include up to 4 players and also giving the player a choice to include the number of players they want. Hence, implementing the real game of Blokus.

8.1.4 Include α-β pruning

Currently, the game takes a very long time to finish when two AI players play against each other. α-β pruning can be included to make the AI player choose a move quicker. This makes the game more efficient and allows the use of a higher depth limit which further helps to find the best move.

References

1. Derksen, B. *Blokus*. 2006 18/12/06 15/08/16]; Available from: https://en.wikipedia.org/wiki/Blokus.
2. Kask, K. *Set 4: Game-Playing*. 2016 12/08/17]; Available from: http://www.ics.uci.edu/~kkask/Fall-2016%20CS271/slides/04-games.pdf.
3. Hotz, H. *A Short Introduction to Game Theory*. 13/07/06; Available from: https://www.theorie.physik.uni-muenchen.de/lsfrey/teaching/archiv/sose_06/softmatter/talks/Heiko_Hotz-Spieltheorie-Vortrag.pdf.
4. Jahanshahi, A., M.K. Taram, and N. Eskandari. *Blokus Duo game on FPGA*. in *Computer Architecture and Digital Systems (CADS), 2013 17th CSI International Symposium on*. 2013. IEEE.
5. Schaeffer, J., et al., *Checkers is solved.* science, 2007. **317**(5844): p. 1518-1522.
6. Myerson, R.B., *Game theory*. 2013: Harvard university press.
7. *Game Theory*. [cited 2017 02/08/2017]; Available from: http://www.investopedia.com/terms/g/gametheory.asp.
8. Prisner, E., *Game theory: through examples*. 2014: Mathematical Association of America.
9. Luce, R.D. and H. Raiffa, *Games and decisions: Introduction and critical survey*. 2012: Courier Corporation.
10. Kockesen, L.a.O., E. , *An Introduction to Game Theory*. 1st Edition ed. 2007.
11. Shoham, Y. and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. 2008: Cambridge University Press.
12. Shor, M. *Symmetric Game*. 12/08/05 [cited 2017 02/08/17]; Available from: http://www.gametheory.net/dictionary/games/SymmetricGame.html.
13. Plan, A. *Symmetric n-player games*. 13/03/17; Available from: https://econ.arizona.edu/sites/econ/files/wp2017-17-08_symmetric_n-player_.pdf.
14. Chandrasekaran, R. *Cooperative Game Theory*. Available from: http://www.utdallas.edu/~chandra/documents/6311/coopgames.pdf.
15. Aumann, R.J. and A. Brandenburger, *Epistemic conditions for nash equilibrium*, in *Readings in Formal Epistemology*. 2016, Springer. p. 863-894.
16. Rasmusen, E. and B. Blackwell, *Games and information.* Cambridge, MA, 1994. **15**.
17. Ferguson, T.S. *Game Theory*. 08/08/17]; Available from: https://www.math.ucla.edu/~tom/Game_Theory/coal.pdf.
18. Allis, L.V., *Searching for solutions in games and artificial intelligence*. 1994: Ponsen & Looijen.
19. Van Den Herik, H.J., J.W. Uiterwijk, and J. Van Rijswijck, *Games solved: Now and in the future.* Artificial Intelligence, 2002. **134**(1-2): p. 277-311.
20. Pearl, J., *Heuristics: intelligent search strategies for computer problem solving.* 1984.
21. Apter, M.J., *The Computer Simulation of Behaviour*. 1970: London: Hutchinson & Co.
22. Zettlemoyer, L. *Adversarial Search*. CSE 473: Artificial Intelligence Autumn 2011 2011 10/08/17]; Available from: https://courses.cs.washington.edu/courses/cse473/11au/slides/cse473au11-adversarial-search.pdf.
23. *Alpha Beta Exercises*. 12/08/17]; Available from: http://classes.engr.oregonstate.edu/eecs/spring2012/cs331/lectures/AlphaBetaExercises.2pp.pdf.
24. Russell, S., P. Norvig, and A. Intelligence, *A modern approach.* Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs, 1995. **25**: p. 27.
25. Mausam. *Adversarial Search Chapter 5*. 10/08/17]; Available from: https://courses.cs.washington.edu/courses/csep573/11wi/lectures/05-games.pdf.

26. Hosch, W.L., *Machine Learning*.
27. Marsland, S., *Machine learning: an algorithmic perspective*. 2015: CRC press.
28. Kaelbling, L.P., M.L. Littman, and A.W. Moore, *Reinforcement learning: A survey.* Journal of artificial intelligence research, 1996. **4**: p. 237-285.
29. Ponsen, M., et al., *Knowledge acquisition for adaptive game AI.* Science of Computer Programming, 2007. **67**(1): p. 59-75.
30. Radcliffe, T., *Python vs. C/C++ in Embedded Systems*. 2016.
31. Cai, J.C., et al. *From C to Blokus Duo with LegUp high-level synthesis*. in *Field-Programmable Technology (FPT), 2013 International Conference on*. 2013. IEEE.

Appendix

settings.py

```
from copy import deepcopy


TOTAL_TILES = 89

BOARD_SIZE = 14

MIN_MAX_LEVELS = 2

CANVAS_WIDTH = 1500

CANVAS_HEIGHT = 800

INFINITY = 1.e400


INTRO = "Each piece placed must be touching a corner (but not on an
edge) of a piece of same colour. Place as many tiles as possible.
Most number of tiles on the board wins."

INTRO2 = "When asked for a move, type the decimal number of piece.
When prompted, enter y coordinate and x coordinates to place the
(0,0) tile of the piece."


piece_list = [ 'unit', 'pair',

               '3line', '3v',

               '4line', '4el', '4tee', '4sq', '4z',

               '5line', '5el', '5z', '5l', '5u', '5t', '5T',

               '5v',    '5w',  '5s', '5?', '5cross' ]

# cover, corner and edge squares are listed as (y,x) coords

# with (0,0) being the top leftmost cell of the piece.

piece_spec = { 'unit': {   'cover': [[1]],

                     'reflection': 'no',
```

```
                'rotation': 'no',
                  'size': 1
                 },
          'pair': {   'cover': [[1,1]],
                'reflection': 'no',
                  'rotation': 'yes',
                  'size': 2
                 },
        '3line': {   'cover': [[1,1,1]],
                'reflection': 'no',
                  'rotation': 'yes',
                  'size': 3
                 },
           '3v': {   'cover': [[1,1],[0,1]],
                'reflection': 'no',
                  'rotation': 'yes',
                  'size': 3
                 },
        '4line': {   'cover': [[1,1,1,1]],
                'reflection': 'no',
                  'rotation': 'yes',
                  'size': 4
                 },
          '4el': {   'cover': [[1,1,1],[1,0,0]],
              'reflec_cover': [[1,1,1],[0,0,1]],
                'reflection': 'yes',
                  'rotation': 'yes',
                  'size': 4
                 },
```

```
'4tee': {   'cover': [[1,1,1],[0,1,0]],
      'reflection': 'no',
        'rotation': 'yes',
        'size': 4
       },
 '4sq': {   'cover': [[1,1],[1,1]],
      'reflection': 'no',
        'rotation': 'no',
        'size': 4
       },
  '4z': {   'cover': [[1,1,0],[0,1,1]],
     'reflec_cover': [[0,1,1],[1,1,0]],
      'reflection': 'yes',
        'rotation': 'yes',
        'size': 4
       },
'5line': {   'cover': [[1,1,1,1,1]],
      'reflection': 'no',
        'rotation': 'yes',
        'size': 5
       },
  '5el': {   'cover': [[1,1,1],[1,1,0]],
     'reflec_cover': [[1,1,1],[0,1,1]],
      'reflection': 'yes',
        'rotation': 'yes',
        'size': 5
       },
   '5z': {   'cover': [[1,1,0,0],[0,1,1,1]],
     'reflec_cover': [[0,0,1,1],[1,1,1,0]],
```

```
                'reflection': 'yes',
                 'rotation': 'yes',
                 'size': 5
                },
           '5l': {   'cover': [[1,1,1,1],[1,0,0,0]],
              'reflec_cover': [[1,1,1,1],[0,0,0,1]],
                'reflection': 'yes',
                 'rotation': 'yes',
                 'size': 5
                },
           '5u': {   'cover': [[1,0,1],[1,1,1]],
                'reflection': 'no',
                 'rotation': 'yes',
                 'size': 5
                },
           '5t': {   'cover': [[1,1,1,1],[0,1,0,0]],
              'reflec_cover': [[1,1,1,1],[0,0,1,0]],
                'reflection': 'yes',
                 'rotation': 'yes',
                 'size': 5
                },
           '5T': {   'cover': [[1,1,1],[0,1,0],[0,1,0]],
                'reflection': 'no',
                 'rotation': 'yes',
                 'size': 5
                },
           '5v': {   'cover': [[1,0,0],[1,0,0],[1,1,1]],
                'reflection': 'no',
                 'rotation': 'yes',
```

```
                        'size': 5
                       },
                '5w': {   'cover': [[1,0,0],[1,1,0],[0,1,1]],
                     'reflection': 'no',
                      'rotation': 'yes',
                      'size': 5
                       },
                '5s': {   'cover': [[1,1,0],[0,1,0],[0,1,1]],
                   'reflec_cover': [[0,1,1],[0,1,0],[1,1,0]],
                     'reflection': 'yes',
                      'rotation': 'yes',
                      'size': 5
                       },
               '5?':  {   'cover': [[1,1,0],[0,1,1],[0,1,0]],
                   'reflec_cover': [[0,1,1],[1,1,0],[0,1,0]],
                     'reflection': 'yes',
                      'rotation': 'yes',
                      'size': 5
                       },
            '5cross': {   'cover': [[0,1,0],[1,1,1],[0,1,0]],
                     'reflection': 'no',
                      'rotation': 'no',
                      'size': 5
                       }
            }



initial_board = [[ 0 for x in range(BOARD_SIZE) ]
                        for y in range(BOARD_SIZE) ]
```

```
piece_list1 = deepcopy( piece_list )

piece_list2 = deepcopy( piece_list )


initial_state = ( 1, initial_board, piece_list1, piece_list2, 1 )
```

library.py

```
from tkinter import *
from settings import *


def setup():
    global w
    master = Tk()
    w = Canvas(master, width=CANVAS_WIDTH, height=CANVAS_HEIGHT)
    w.pack()


# get all the coordinates of the orientations of a piece
def all_orientations(p):
    coord_list = set()
    matrix = piece_spec[p]['cover']
    coord_list = coord_list.union(coordinates_matrix(matrix))


    rotate = piece_spec[p]['rotation']
    if(rotate == 'yes'):
        coord_list = coord_list.union(coordinates_rotate(matrix))


    reflec = piece_spec[p]['reflection']
    if(reflec == 'yes'):
        matrix = piece_spec[p]['reflec_cover']
        coord_list = coord_list.union(coordinates_matrix(matrix))
        coord_list = coord_list.union(coordinates_rotate(matrix))


    return list(coord_list)
```

```
# get the coordinates from a given matrix
def coordinates_matrix( matrix ):
    coord_set = set()
    l = []
    b = 0

    for x in range(len(matrix[0])):   #    left to right
        a = 0
        for m1 in matrix:
            if m1[x] == 1:
                l.append((a,b))
            a += 1
        b += 1
    tup1 = tuple(l)
    coord_set.add(tup1)
    return coord_set

# get the rotation coordinates given a matrix
def coordinates_rotate( matrix ):
    coord_set = set()
    l = []
    b = 0

    for x in range(len(matrix)-1,-1,-1):    #    bottom to up
        a = 0
        for g in matrix[x]:
            if g == 1:
                l.append((a,b))
```

```
            a += 1
        b += 1

    tup2 = tuple(l)
    coord_set.add(tup2)

    l = []
    b = 0

    for g in range(len(matrix[0])-1,-1,-1):
        a = 0
        for x in range(len(matrix)-1,-1,-1):    #    right to left
            if matrix[x][g] == 1:
                l.append((a,b))
            a += 1
        b += 1

    tup3 = tuple(l)
    coord_set.add(tup3)

    l = []
    b = 0

    for x in matrix:    #    top to bottom
        a = 0
        for g in range(len(x)-1,-1,-1):
            if x[g] == 1:
```

```
                l.append((a,b))
            a += 1
        b += 1

    tup4 = tuple(l)
    coord_set.add(tup4)

    return coord_set

# place a piece on the board given the orientation position and
coordinates
def place_piece( p, orientation_no, Y, X, state ):
    player = state[0]
    board = state[1]
    move_number = state[4]

    if not ((player == 1 and p in state[2]) or (player == 2 and p in
state[3])):
        print("FAIL. PLAYER DOES NOT HAVE THIS PIECE")
        return board

    newboard = deepcopy(board)
    initial = False
    valid = False

    plac = all_orientations(p)
    piece_to_place = plac[int(orientation_no)-1]
    (corners,edges) = get_all_corners_edges(board, player)
```

```
    for coord in piece_to_place:
        y = Y+coord[0]
        x = X+coord[1]

        if (newboard[y][x] != 0):
            print(( p, orientation_no, board, Y, X, player,
move_number ))
            print("FAIL. OVERPLACING")
            return board
        elif (((move_number == 1 or move_number == 2 ) and  # check
whether the first move fills in the tiles on each corner of the grid
               ((y,x) == (0,0) or (y,x) == (BOARD_SIZE -1,
BOARD_SIZE -1))) or move_number > 2):
            initial = True

        # check whether a valid move
        if ((move_number > 2 and (y,x) in corners) or move_number <=
2):
            valid = True
        elif ((y,x) in edges):
            print(( p, orientation_no, board, Y, X, player,
move_number ))
            print("FAIL. CAN'T PLACE ON EDGE")
            return board

        newboard[y][x] = player

    if (initial and valid):
        return newboard
```

```
   else:
        print("FAIL. PIECE NOT PLACED CORRECTLY ON CORNER")
        return board

# draw a square cell given the cell size and coordinates
def draw_cell(topleft_x,topleft_y,cell_size,y,x,col):
    w.create_rectangle(topleft_x + (x*cell_size),
                       topleft_y + (y*cell_size)+10,
                       topleft_x + ((x+1)*cell_size),
                       topleft_y + ((y+1)*cell_size)+10,
                       fill=col)
    w.update()

# display the tiles on the canvas
def display_tiles( tiles, topleft_x, topleft_y, col ):
    initial = topleft_x
    n = 1
    e = 1

    for p in piece_list:
        if (p in tiles):
            e = 1
            for piece in all_orientations(p):
                for coord in piece:

draw_cell(topleft_x,topleft_y,10,coord[0],coord[1],col)
                w.create_text(topleft_x + 10,topleft_y+60,
text=str(n)+"."+str(e),font="bold")
                w.update()
```

```
                topleft_x += (10 + (len(piece) * 10))

                if topleft_x > CANVAS_WIDTH-230:
                    topleft_x = initial
                    topleft_y += (20 + (len(piece) * 10))
                e += 1
        n = n + 1


# display the board and the pieces of each player
def display_state( state ):
    w.delete("all")

    board = state[1]

    for y in range(BOARD_SIZE):
        w.create_text(55 + (y*25),35, text=str(y), font="bold")
        w.create_text(25,67 + (y*25), text=str(y), font="bold")
        w.update()
        for x in range(BOARD_SIZE):
            draw_cell(45,45,25,y,x,'#dddddd' if board[y][x] == 0
else "red" if board[y][x] == 1 else "blue")

    display_tiles(state[2],430,5,'red')
    display_tiles(state[3],30,430,'blue')

def check_orientation_with_corners( board, board_open_corners,
board_edges, orientations, position, piece ):
```

```
    moves = []
    tilecorners = tile_corners(orientations[position])
    for corner in board_open_corners:
        Y = corner[0]
        X = corner[1]
        for t in tilecorners:
            c = [n for n in orientations[position] if not (n[0] ==
t[0] and n[1] == t[1])]    # get all coords of tile except the
corner tuple
            valid = True

            for r in c:
                diffY = Y+(r[0] - t[0])
                diffX = X+(r[1] - t[1])

                # check whether this orientation is a valid move on
the board
                if not ( diffY < BOARD_SIZE and diffY > -1 and diffX
> -1 and
                         diffX < BOARD_SIZE and board[diffY][diffX]
== 0 and
                         (diffY,diffX) not in board_edges):
                    valid = False
                    break

            # add to the list of possible moves
            if valid:
                move = (Y-t[0],X-t[1],position+1,piece)
                moves.append(move)
    return moves
```

```
# get all the corner for a specific orientation of a tile
def tile_corners( coords ):
    corners = []

    for i in coords:
        y = i[0]
        x = i[1]
        if (((y+1,x) not in coords or (y-1,x) not in coords) and
((y,x-1) not in coords or (y,x+1) not in coords)):
            corners.append(i)

    return corners

# get all edges and corners of the board for the specific player
def get_all_corners_edges( board, player ):
    corners = set()
    edges = set()

    for y in range(BOARD_SIZE):
        for x in range(BOARD_SIZE):
            if (board[y][x] == player):
                if (y > 0 and board[y-1][x] == 0):
                    edges.add((y-1,x))   # top edge
                    if (x < BOARD_SIZE-1 and board[y-1][x+1] == 0
and edges_of_corner(y-1,x+1,board,player)): # top right corner
                        corners.add((y-1,x+1))
                    if (x > 0 and board[y-1][x-1] == 0 and
edges_of_corner(y-1,x-1,board,player)): # top left corner
                        corners.add((y-1,x-1))
```

```
                if (y < BOARD_SIZE-1 and board[y+1][x] == 0):
                    edges.add((y+1,x))   # bottom edge
                    if(x < BOARD_SIZE-1 and board[y+1][x+1] == 0 and
edges_of_corner(y+1,x+1,board,player)): # bottom right corner
                        corners.add((y+1,x+1))
                    if (x > 0 and board[y+1][x-1] == 0 and
edges_of_corner(y+1,x-1,board,player)): # bottom left corner
                        corners.add((y+1,x-1))


                if (x > 0 and board[y][x-1] == 0): # left edge
                    edges.add((y,x-1))
                if (x < BOARD_SIZE-1 and board[y][x+1] == 0): #
right edge
                    edges.add((y,x+1))


    return (list(corners), list(edges))


def edges_of_corner(cy, cx, board, player):
    if(((cy > 0 and board[cy-1][cx] != player) or cy == 0) and # top
       ((cx < BOARD_SIZE-1 and board[cy][cx+1] != player) or cx ==
BOARD_SIZE-1) and # right
       ((cy < BOARD_SIZE-1 and board[cy+1][cx] != player) or cy ==
BOARD_SIZE-1) and # bottom
       ((cx > 0 and board[cy][cx-1] != player) or cx == 0)):    #
left
        return True


    return False


def calc_score_pieces( pieces ):
```

```
    n = 0
    for i in pieces:
        p = piece_spec[i]['size']
        n += p
    return n


def get_sized_pieces(pieces, size_pieces):
    new_pieces = []
    for i in pieces:
        n = piece_spec[i]['size']
        if n == size_pieces:
            new_pieces.append(i)


    return new_pieces
def win( state ):
    p1 = calc_score_pieces(state[2])
    p2 = calc_score_pieces(state[3])


    if p1 == 0:
        p1 -= 15


    if p2 == 0:
        p2 -= 15


    if ( p1 < p2 ):
        print ('Player 1 has won the game.')
    elif ( p2 < p1 ):
        print ('Player 2 has won the game.')
```

```
    else:
        print ('The game is a tie!')


    print()
    print ('Scores:')
    print('Player 1 = ', TOTAL_TILES - p1)
    print('Player 2 = ', TOTAL_TILES - p2)
```

blokus.py

```
from copy import deepcopy


import time
from settings import *
from library import *
maxcalls = 0
mincalls = 0


# get all the possible moves that the computer can do, given the
state of the board
def possible_moves( state ):
    player = state[0]
    board = state[1]
    move_number = state[4]


    if player == 1:
        player_pieces = state[2]
    else:
        player_pieces = state[3]


    moves = []
    sizes = [5,4,3,2,1]


    # if it's first move of the computer then must place on either
corner of the board
    if(move_number == 1 or move_number == 2):
        if board[0][0] == 0:
            board_open_corners = [(0,0)]
```

```
        else:
            board_open_corners = [(BOARD_SIZE-1, BOARD_SIZE-1)]
        board_edges = []
    else:
        (board_open_corners, board_edges) =
get_all_corners_edges(board, player)


    i = 0
    while moves == [] and not i == len(sizes):
        sized_pieces = get_sized_pieces(player_pieces, sizes[i])
        for x in sized_pieces:
            p = all_orientations(x)   # find all orientations of the
piece
            for u in range(len(p)):
                moves_each_orientation =
check_orientation_with_corners(board, board_open_corners,
board_edges, p, u, x)
                moves = moves + moves_each_orientation
        i += 1


    return moves


def max_min_value( state, player, max_or_min, no_of_levels, move ):
    global maxcalls, mincalls
    if (max_or_min == 'min'):
        mincalls += 1
    else:
        maxcalls += 1


    if (state[0] == 1):
```

```
        current_player = 1
        next_player = 2
        player_pieces = state[2]
        other_player_pieces = state[3]
    else:
        current_player = 2
        next_player = 1
        player_pieces = state[3]
        other_player_pieces = state[2]


    new_pieces = deepcopy(player_pieces)


    newboard = place_piece( move[3], move[2], move[0], move[1],
state )
    new_pieces.remove(move[3])


    if no_of_levels-1 == 0:
        return heuristic(newboard, player)


    if (current_player == 1):
        new_state = (next_player, newboard, new_pieces,
other_player_pieces, state[4] + 1)
    else:
        new_state = (next_player, newboard, other_player_pieces,
new_pieces, state[4] + 1)


    if (max_or_min == 'min'):
        v = INFINITY
    else:
        v = -INFINITY
```

```
    next_player_moves = possible_moves(new_state)


    # if there are more levels than can be searched for current
player, skip search of next player
    if next_player_moves == [] and no_of_levels > 2:
        if(current_player == 1):
            new_state = (current_player, newboard, new_pieces,
other_player_pieces, state[4] + 2)
        else:
            new_state = (current_player, newboard,
other_player_pieces, new_pieces, state[4] + 2)


        return skip_turn(new_state, player, no_of_levels-1,
max_or_min)


    #    analyse each move
    for m in next_player_moves:
        if(max_or_min == 'min'):
            r = max_min_value(new_state, player, 'max',
no_of_levels-1, m)
            v = min(v, r)
        else:
            r = max_min_value(new_state, player, 'min',
no_of_levels-1, m)
            v = max(v, r)


    if next_player_moves == []:
        return heuristic(newboard, player)


    return v
```

```
def skip_turn (state, player, no_of_rounds, max_or_min):
    global maxcalls, mincalls
    if (max_or_min == 'min'):
        mincalls += 1
    else:
        maxcalls += 1

    moves = possible_moves(state)

    if moves == []:
        return heuristic(state[1], player)

    if (max_or_min == 'min'):
        v = -INFINITY
    else:
        v = INFINITY

    for m in moves:
        if(max_or_min == 'min'):
            r = max_min_value(state, player, 'min', no_of_rounds-1,
m)
            v = max(v, r)
        else:
            r = max_min_value(state, player, 'max', no_of_rounds-1,
m)
            v = min(v, r)
    return v

def computer_turn( state ):
```

```
    moves = possible_moves(state)


    if(moves == []):
        return (state[1],'')


    pos = -1
    heu = -INFINITY
    inc = 0
    for m in moves:
        h = max_min_value( state, 1, 'min', MIN_MAX_LEVELS, m)    #
place the piece and find heuristic value
        if ( h > heu ):
            pos = inc
            heu = h
        inc += 1


    move = moves[pos]
    newboard = place_piece( move[3], move[2], move[0], move[1],
state )
    return (newboard, move[3])


#    find best move using this function
def heuristic( board, player ):
    if player == 1:
        other_player = 2
    else:
        other_player = 1


    open_corners = get_all_corners_edges(board, player)[0]
```

```
    open_corners_other = get_all_corners_edges(board,
other_player)[0]


    return (len(open_corners)*2) - len(open_corners_other)


#    implements a human turn of the game
def human_turn( state ):
    board = state[1]
    newboard = board


    while (newboard == board):  # until their move is valid keep
asking
        print()
        p = input('Your move? ')
        if(p.lower() == 'pass'):
            return (board,'')
        l = p.split('.')
        piece = list(piece_spec)[int(l[0])-1]
        y = int(input('Y coordinate '))
        x = int(input('X coordinate '))
        try:
            newboard = place_piece(piece,l[1],y,x,state)
        except IndexError:
            print('PLEASE TRY AGAIN')


    return (newboard, piece)


# PLAY GAME
def game( state ):
```

```
    print(INTRO)
    print()
    print(INTRO2)
    print()
    print("GAME START")
    move_number = state[4]
    player = 1
    finish1 = False
    finish2 = False
    while True:
        display_state(state)
        board = state[1]
        print( '\nPlayer to move: ', state[0], "\n" )
        if(move_number%2 == 0): #    Player 2
            (newboard, piece) = human_turn(state)

            if(newboard == board and finish1):
                break
            elif(newboard == board):
                finish2 = True
            else:
                state[3].remove(piece)

            player = 1
        else:                #    Player 1
            (newboard, piece) = computer_turn(state)

            if(newboard == board and finish2):
                break
```

```
            elif(newboard == board):
                finish1 = True
            else:
                state[2].remove(piece)

            player = 2

        move_number += 1
        new_state = (player, newboard, state[2], state[3],
move_number)
        state = new_state

    win(state)

def main():
    setup()
    game(initial_state)

main()
```