COMP390

2022/23

Blokus Game

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Abstract

The increasing computational power of computers is paving the way for artificial intelligence advancements in various fields, including game theory, enabling the development of increasingly sophisticated algorithms. This project aimed to design a computer-based version of the Blokus board game, accommodating two and four players. The project employed Python as the coding language and PyGame library to create an interactive and visually appealing interpretation of the Blokus. The project also aimed to create various playing strategies for computer players to explore the absence of a definitive, flawless Blokus playing strategy, especially for Blokus Classic. The computer player strategies varied in their approaches, including expanding, obstructing, or both, tested by playing against each other and human players. The techniques explored utilised information on the size of the piece, the number of corners, the number of possible moves, and the number of influence squares.

To further enhance gameplay, the project incorporated advanced functions that enabled players to break through opponents' paths and play optimal pieces at the end of the game. Additionally, the project presented a sophisticated hybrid strategy that seamlessly combined all the above information, each with different strengths based on the game's progression. Despite the hybrid strategy's dominance, the dissertation proposes additional enhancements to upgrade each strategy, considering its limitations.

Contents

1. Introduction

Artificial Intelligence (AI) is a field of computer science that deals with developing intelligent machines that can simulate human cognitive processes, such as learning, reasoning, and self-correction. AI has become an essential tool in many areas, including medicine, finance, transportation, and education, revolutionising how we live, work, and interact with the world. In the gaming industry, AI has played a crucial role in developing game-playing agents that can compete with human players, providing a more engaging and challenging gaming experience. AI algorithms have been used to create intelligent agents that can learn from experience, make strategic decisions, and adapt to changing game environments. Developing AI strategies in games has become an active research area, attracting researchers from different fields, including computer science, mathematics, and game theory. The main goal of developing AI strategies in games is to create game-playing agents that can outperform human players or provide a fair challenge to them. By considering and computing many possible moves, AI agents can identify optimal strategies, explore game spaces, and predict opponents' moves, capabilities that human players cannot match.

Blokus is an abstract strategy board game that was first introduced in 2000. It was designed by Bernard Tavitian and is played by two to four players. The game has won several awards, including the Mensa Select, Teacher's Choice, and Parent's Choice Gold awards. Blokus is known for its simple rules and challenging gameplay that appeals to players of all ages and skill levels. The project enhances the Blokus Classic and Duo (versions for four and two players) experience by creating an interactive digital game. The project also explores various AI strategies, mainly greedy algorithms, which will be tested and evaluated. Understanding the game's rules is essential to fully comprehend the project [1].

Rules for Blokus Classic:
The game is played on a square board divided into 20 rows and 20 columns for 400 squares. There are 84 game tiles organised into 21 shapes in four colours: blue, yellow, red, and green. The 21 shapes are based on free polyominoes of from one to five squares (one monomino, one domino, two triominoes, five tetrominoes, and 12 pentominoes). The order of play is based on colour, with blue going first, followed by yellow, red, and green. The first piece played of each colour is placed in one of the board's four corners. Each new piece played must be put to touch at least one piece of the same colour, with only corner-to-corner contact allowed - edges cannot touch. When a player cannot place a piece, they pass, and play continues as usual. The game ends when no one can place a piece. When a game ends, the score is based on the number of squares in each player's unplaced pieces; a player loses one point for each square (e.g., a tetromino is worth 4 points). Players who play all their pieces get a bonus

score of +20 points. If the last piece played was a monomino, +15 points otherwise.

Rules for Blokus Duo:
Blokus Duo is for two players only and uses a smaller (14×14) board; the pieces are of blue and yellow colours. The two starting squares are placed not in the corner (as in the original Blokus game) but nearer to the centre ((5,5) and (10,10)). This makes a crucial difference in the flavour of the game because players' pieces may (and usually do) touch after the first move. Even more, than with the original game, Blokus Duo is an offence-centred game; it is also a much purer strategy game than the four-player game since one is not in danger of getting ganged up on by three other players (as sometimes happens with the four-player version).

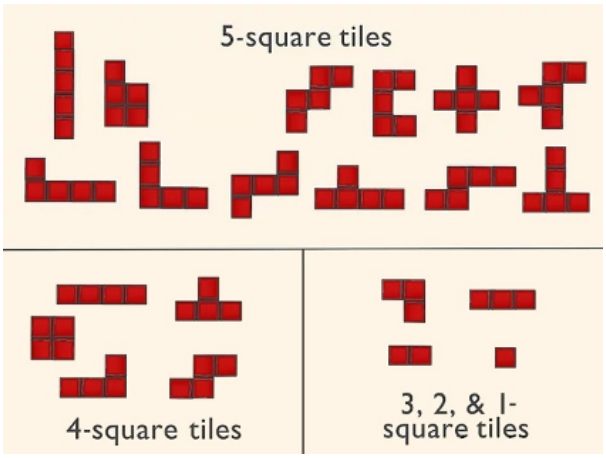

Figure 1.1: Game Pieces

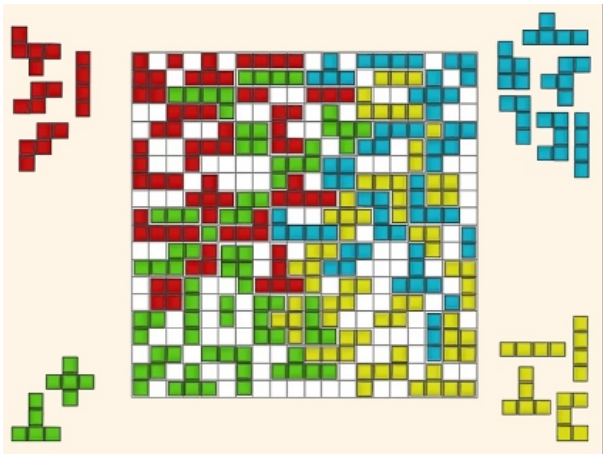

Figure 1.2: Blokus Classic Board

In this project, to monitor the point system throughout the game, points are awarded to players for each square they place on the board. For instance, a pentomino is worth 5 points. This method yields the same winner as the original rules but also offers players valuable information during the game.

Previous projects that simulated Blokus gameplay have primarily focused on developing AI strategies without a graphical user interface. The Python programming language was selected to evaluate strategies through observation. After considering various game engines, it was determined that many needed to be simpler for the relatively straightforward gameplay of Blokus. Thus, the decision was made to use Python with Pygame, a cross-platform library of Python modules designed for creating video games. Pygame provides tools for developing 2D games that can run on different operating systems, including features for managing user input, graphics, sound, and networking.

The Blokus game is fully observable, sequential, and deterministic. While the game is tantalisingly solvable with unlimited computational power, it is still too complex to solve comprehensively. This leads to the creation of AI strategies. To create robust algorithms, it is necessary to identify factors that make a move strong. The difference in the number of

players in Blokus Classic ad Blokus Duo, as well as the size of the board, changes most of the objectives in the game. It is crucial to consider these two types separately.

In Blokus Classic, moving towards the centre of the board [2] and playing larger pieces first is essential [3]. This strategy allows players to occupy more area, reach the centre of the board earlier, have more move possibilities, gain more points, and take advantage of the fact that playing larger pieces in later stages of the game may be difficult. Generally, it pays off to avoid confrontation, although contact with opponents may be unavoidable. In cases where the player blocks everyone from a specific area, they cannot fully utilise the space and will only take advantage of about half of it. The perfect strategy involves being only with one other player in each area and cooperating for self-gain.

Blokus Duo is much easier to solve than Blokus Classic; therefore, more research has been done on it. There is an evident first-player advantage in Blokus [4], and this must be considered during the testing process. The player who starts can reach a strategic area on the board first and may continue to have an advantage over the other player throughout the game, always being one step ahead. The same research also emphasises the importance of the size of the piece. Playing the most significant pieces possible in a random location won in a hundred-match game against the strategy of maximising the number of corners after each move. Furthermore, a hybrid concept balancing many strategies throughout the game has been developed [5].

TABLE I. WHEIGHT OF EACH STERATEGY DURING THE GAME

Phases	**Strategies**			
	Disrupt other player	***Increase chance of your next move***	***Bigger tiles***	***Strategic points***
Start of the game	High	High	Very High	Normal
Middle of the game	High	Normal	High	Very High
End of the game	Low	Low	Normal	Very High

Figure 1.3: Dynamic Assignment of Strategies

This strategy assigns different weights to each strategy depending on the phase of the game and has been proven to be strong, winning the round-robin contest against other strategies.

Another successful approach has been found by maximising the number of influenced squares, which explores the unvisited areas of the board [6].

The most advanced strategy developed in Blokus Duo is the min-max algorithm with a large depth. Therefore, most projects focus on minimising computational costs while

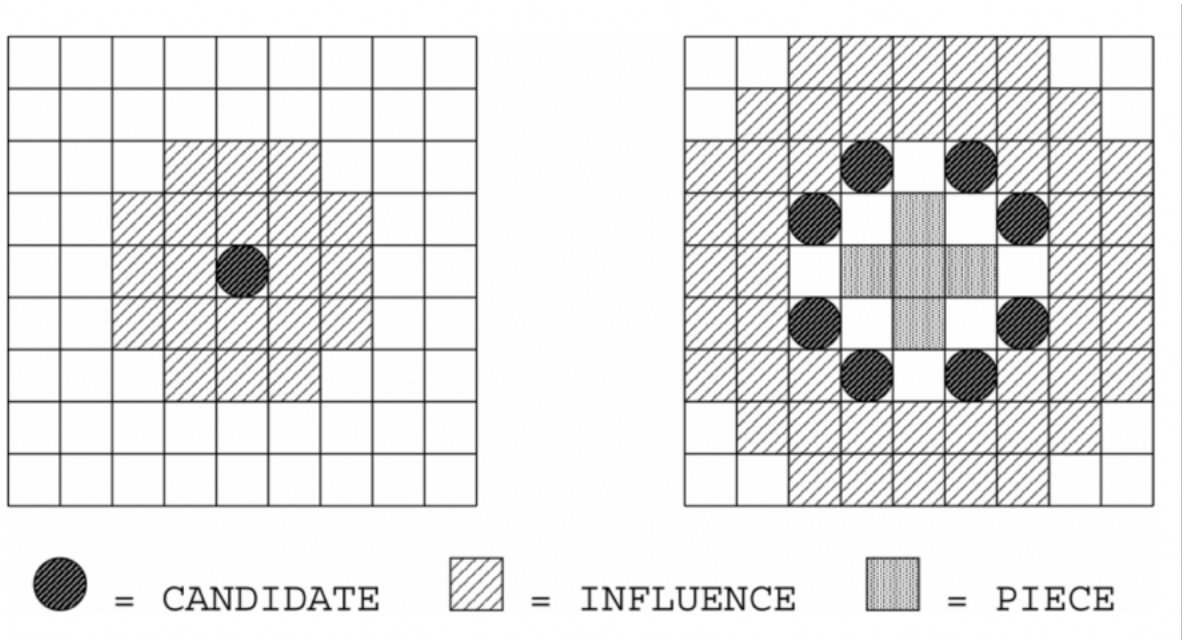

Figure 1.4: Description of the Influence Squares

maximising the possible depth [6]–[9]. This approach is only possible in Blokus Duo, as assuming that every player will play against you in Blokus Classic is unrealistic.

2. Aims & Objectives

The main objective of this project is to explore the fundamental greedy algorithms in Blokus gameplay, such as those utilising information on the size of the piece, the number of corners, possible moves, and influence squares. An updated version of the Hybrid strategy will be developed, combining all the basic strategies and altering their strengths and assignments more frequently. Different strategies will be balanced by testing and identifying their advantages and disadvantages during gameplay. Often, the strengths of one strategy can offset the weaknesses of another. As the development of a graphical game may slow down the optimisation for the max-min algorithm, this project will not delve into its optimisation.

Based on the gathered information, the project established key Aims and Objectives necessary for planning and executing the project. However, some objectives were revised throughout the development, and new ones were added.

The final list of Aims & Objectives:

1. Create a playable Blokus game.
    - Create a board, pieces, and a player.
    - Implement choosing, rotating, and placing a shape.
    - Set the rules of the game.
    - Terminate the game at the right time.
2. Implement mechanics of placing a shape.
    - Indicate if it's possible to play a shape.
    - Pass a turn if placing a shape is impossible, or finish the game.
    - Pick a piece, move it, rotate it, and put it in a specific place on a board.
    - Update the player's data.
3. Create a move ranking.
    - Create a result object.
    - Sort moves depending on their score.
    - Assign points for each move.
    - Combine several lists of results.
    - Determine the most vital move.

4. Implement basic computer player strategies.
    - Play a random piece.
    - Play the largest piece first.
    - Maximise the number of corners.
    - Maximise the number of moves.
    - Maximise the number of influence squares.
    - Minimise the number of moves for opponents.
    - Find the move that goes through the opponents' paths.
5. Implement a strong computer player strategy.
    - Test and evaluate basic computer player strategies.
    - Find out the significant move factors at each stage of the game.
    - Combine basic computer player strategies and additional functions.
6. Create interactive buttons.
    - Start a new game with predefined strategies.
    - Recommend the move for a human player.
    - Hide the move recommendation.

The removal of the objective [1] was necessary as it would have required a different approach to the AI strategies, which would have resulted in the need for a new implementation. This would have been both time-consuming and needed to be more complex to create a strategy solely based on the objective.

[1] Implement Strategy - Choose an area where there are fewer players but more than 0

3. Design

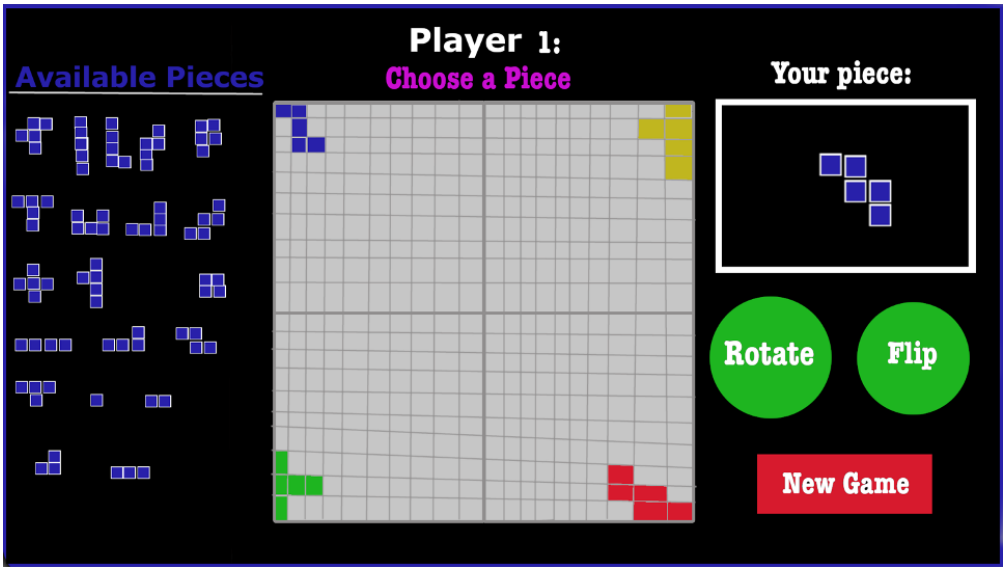

Figure 3.1: Initial UI & UX design

3.1 Pieces

The game pieces are designed as "Shape" objects based on the form of a similar piece implementation from another Blokus project [4]. Each shape must have a unique identifier, size, and arrays of points, corners, and sides. The shape must be able to rotate and flip, with its data being updated accordingly.

3.2 Player

The "Player" object represents human and computer players during the game. Each player must have a unique identifier, set of available pieces, corners, and influence squares. Additionally, the player must keep track of their current score and the number of possible moves. Although not all strategies may require all this data, it is essential to store it for easy accessibility by other players. Since the Blokus game is fully observable, this information is not considered private. Each player is responsible for managing their data, such as selecting

a piece, placing it on the board, removing it from their set, adding points, and updating all other relevant information. Lastly, the player must store information about their strategy, which will also determine whether they are an AI or a human player.

3.3 Board

The "Board" object is a representation of the game's board. It contains information about the board's size and an array representing it. The array will have numerical values, where 0 denotes an empty cell, and numbers 1 to 4 represent the players' IDs. Furthermore, the board will store information about the available points where it is legal to play the selected piece. Lastly, it will store the points of a recommended move.

3.4 Rules of the move

Every move must pass through several verification checks. Firstly, all the piece's points must be within the board's boundaries. Secondly, the squares that the piece will occupy must be vacant. Thirdly, the piece must touch at least one of the corners of its previously played pieces. Lastly, it cannot touch any of its previously played pieces' sides.

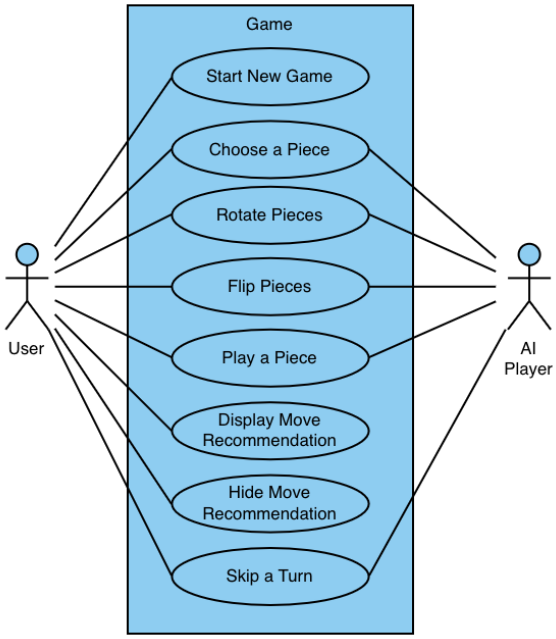

Figure 3.2: Use Case Diagram

3.5 Blokus and rules of the game

The "Blokus" class will act as the primary controller for the game, managing all the game components. It will need to store information about the players participating in the game and the board's size and initialise its objects with essential information, such as the set of pieces available at the start of the game. The class will also be responsible for determining the current player, ending a player's turn, starting the next player's turn, identifying the winner, and terminating the game when it ends. Moreover, the class must provide an option to skip a turn in case of no available moves. Additionally, the class will handle user interactions, such as creating a new game, displaying recommended moves, and allowing human players to select and place a piece.

3.6 Result

The "Result" object will contain the piece's ID, points, and score. It will be feasible to arrange the results, allocate points, and merge results lists to determine the move with the highest score.

3.7 AI Turn

Once the player completes all the checks, they can make a move. The player must generate a Result for each potential action and select the highest score to determine the best possible move. Initially, the player must identify all the available pieces they can play by checking each corner, each element and its eight possible rotations. Next, the player needs to determine all the possible locations and rotations where they can place the selected piece using the same method. Lastly, the strategy evaluates all potential moves by simulating them on copied objects and observing the outcome. The pseudo-code for this algorithm is provided in fig. 8.1.

The AI Turn will be managed through a strategy and its corresponding function. Each strategy is associated with a specific function that generates a list of Result objects. The strategy selects the Result with an optimal score and executes the move it contains. Subsequently, it updates the players' data, and the next player's turn commences.

3.8 Constants

The Constants class contains the constants utilised throughout the project, such as the window size, square board size, frames per second, and colours represented in the RGB system.

3.9 Graphics

The graphics class utilises information stored in the Blokus, Board, Player, and Shape objects to showcase the game's current state. The game graphics are composed of multiple rectangular objects stacked on each other. The left side of the window displays the current player's available pieces, the centre exhibits the game board, the right side holds buttons, and all communications are shown on the banner at the top of the screen.

I designed a detailed class diagram that describes each class, including the rules of a move and the functions for updating specific player data incorporated within the Board class (fig. 8.2).

4. Implementation & Realisation

4.1 Introduction

I researched to see if I could use any already developed software to achieve a working game. I designed the main components of the game and began programming the board object. Once a visible board was placed, pieces and player objects were programmed and rendered. Subsequently, interactions with the game were implemented, considering events like clicking the arrow keys and mouse button. A highlight for a chosen piece was then added, placing the selected element on the board. Once it was confirmed that the interaction was functioning correctly, game rules were implemented. A Blokus object was then created to control game status, with data being updated after each move. User feedback was obtained, and graphical changes were made to improve the user experience. Strategies based on collected data were subsequently developed, which involved automating piece selection and placement. New game buttons were added to test different variations of strategies without needing to restart the game each time. Based on supervisor feedback, an additional strategy was added, along with buttons that recommended different strengths of moves and a button to hide the hint. Finally, a hybrid strategy was created using the Result objects and all other strategies. Strategies were tested against each other, and final feedback was gathered, after which graphical changes were implemented. Below is a detailed description of the most crucial game components designed and implemented to achieve a working game and test AI strategies.

4.2 How to Play a Game

The user can select from various pre-defined strategies during gameplay using any of the eight new game buttons. Upon selection, the game commences, displaying the current player's pieces on screen, while a banner above the board highlights the required action. Users can rotate and flip their pieces using the arrow keys. Upon selecting a piece, legal moves are indicated by dots on the board. Additionally, a reference point is marked on each unplaced piece.

Players can then choose a piece and click on a board to place it. If an illegal move is selected, the piece is unmarked, and the turn remains with the same player. After placing a piece, points are added, and the turn goes to the next player. If a player cannot make a move, they can skip their turn by clicking on the board, with the game displaying an appropriate message. Once the game is over, the winner is announced, and players can check all pieces

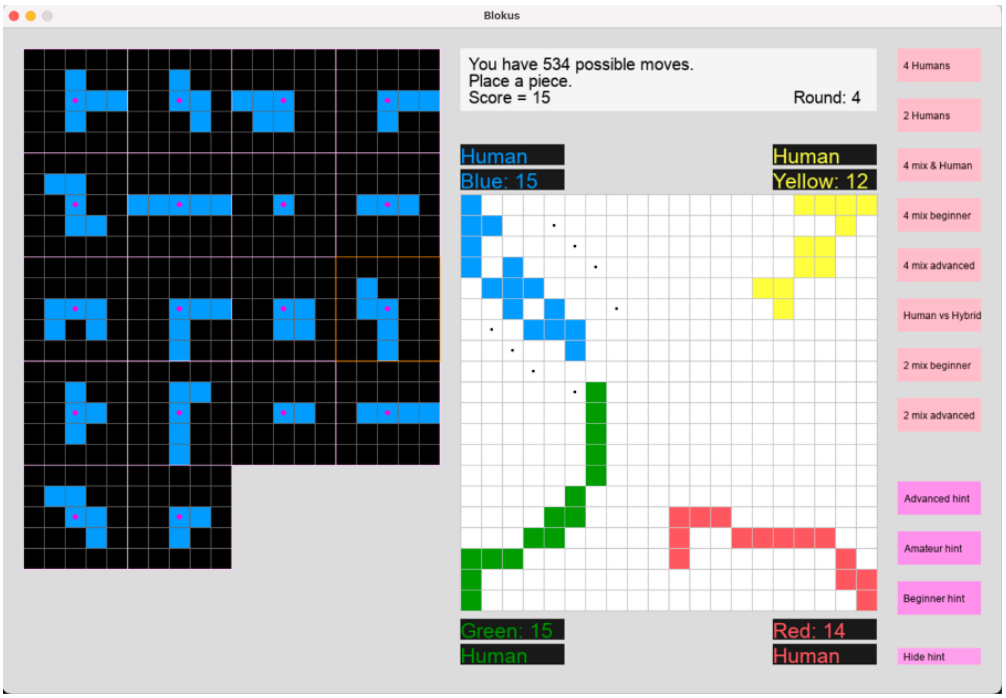

Figure 4.1: Final UI & UX Implementation

and rotate them. To start a new game, the user must click on one of the new game buttons. In case of difficulty finding a legal or strong move, human players can select a hint button to display a suggested move. The player can then hide the hint and make an independent decision. To exit the game, the user may close the window. (Aim 1)

4.3 Pieces

The game pieces were created as Shape objects with unique identifiers and sizes. These Shapes were defined by an array of points, corners, and sides, calculated using the "set_points" function that relied on a reference point (a tuple of integers). The game pieces were conveniently displayed on the left side of the screen, using 5x5 boards, with the reference point located at the centre of each board. This board size was deliberately chosen as it accommodated the largest and widest pieces, five squares in width and length, respectively.

To rotate a piece, its data was rotated by 90 (right arrow key) or 270 (left arrow key) degrees to the right, based on the piece's reference point. To flip a piece, all its data was transformed to the opposite side of the x-axis, again based on the reference point of the piece.

4.4 Player

The Player object represents both human and computer players during the game. Each player is identified by a unique ID number (ranging from 1 to 4) and has a set of available

pieces at the beginning of the game, initially in random order. After playing a piece, it is removed from the player's available pieces (Aim 2). The player also has a set of available corners, which initially contains only one of the corners of the board in Blokus Classic or the point (5,5) or (10,10) in Blokus Duo. After each player's move, new corners are added to the player's set, and all players' corner sets are filtered to ensure that each point is still a legal corner according to the game rules. Similarly, the player has a set of influence squares updated after each move. For each of the player's corners, an array of possible influence squares is created and checked to see if each point is in the bounds of the board and unoccupied before it is added to the set or rejected.

The player's current score is updated every time they make a move, with points added based on the size of the piece played. If the piece played is the last one in the player's set, they receive an additional 20 or 15 points, depending on whether it is a monomino or something else. The number of possible moves for each player is calculated after any move, and legal moves are checked by iterating through each player's corners, pieces, and possible representations. The player's strategy is stored as a string, with the name "Human" indicating a human player and any other name indicating an AI player.

The player object stores the position of the most recently clicked board location, with a value of -1 indicating that none has been clicked. The player object also contains a function called "which_piece" that takes a position as an argument and returns the selected piece. The Board class has a function that returns a square of the board that the position relates to, and the Blokus class contains a function to determine if the clicked coordinates on the board correspond to the new game or hint buttons.

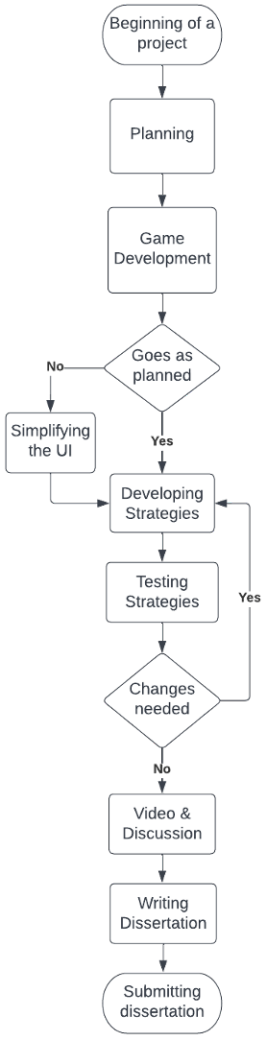

Figure 4.2: Workflow Diagram

4.5 Board

The Board object is responsible for representing the game board. The board is a two-dimensional array of either 14x14 or 20x20. Each square on the board is represented by a numerical value, with 0 indicating an empty square and numbers 1 to 4 indicating which player occupies the square. The object also maintains an array of the same size to tell whether a player can place a selected piece in each location based on its reference point. This is done by placing the points of a chosen piece in each player's corners and checking if the move is legal, with 0 indicating an illegal move and 1 indicating a legal one. To verify the legality of the move, an algorithm checks whether each of

the piece's points is within the bounds of the board, whether it's unoccupied, whether sides of the piece that are in bounds are not touching any already played pieces, and (if it's not the first round) whether at least one of the piece's corners that are inboard bounds touches an already played piece. If it's the first move, one of the piece's points must be in a player's corner that has been manually added at the beginning of the game.

The Board object also stores a set of points that should be highlighted on the board. The Blokus class chooses the appropriate strategy based on the clicked button. The strategy plays a move on a copy of the board, then subtracts the original board from the copy. The resulting squares are not equal to 0 are passed as a highlight, representing the newly placed piece as a recommendation. Additionally, the Board object updates each player's data, including the number of possible moves, the set of corners, and the set of influence squares. It also includes the "valid_pieces" function that returns a set of pieces that a player can legally play in the current state of the game, as well as the "valid_points" function that returns a set of points of a piece that can be played without violating the game's rules.

4.6 Blokus

The main controller of the game is the Blokus object, which integrates all the game components. It takes the number of players and an array of their strategies as arguments. Based on the number of players, it creates the appropriate board size and creates players using the given strategies. All pieces are added to the players' sets, and the number of moves is counted. The specific corners are also manually added to follow the first move rule.

The Blokus object has a "round" attribute and a "current_player" attribute that points to a player in the array who should make the next move. After the move is made, the "next_player" function is called. If the next player's turn is the first player in an array, one point is added to the number of rounds. The "current_player" attribute is then set to the next player in the queue. The Blokus object also has a "finished" attribute that indicates whether the game is over. After each move, the "find_winner" function determines the winner. If every player has no possible moves, the winner is found by comparing the scores of each player, and the "winner_id" attribute stores the string "The winner is: /Winner's colour/".

Furthermore, the Blokus class controls the game buttons (Aim 6). The "game_type_clicked" function returns the number of the clicked button. If the button number is from 1 to 8, the array of strategies is created and shuffled to randomise the order, and a new game is started. If the button number is 9 to 11, the recommended move is highlighted based on the chosen strategy. Button 9 is for the Hybrid strategy, button 10 is for the max_min_moves strategy, and button 11 is for a random strategy. Button 12 clears the Board's highlight_board set and hides the move recommendation (fig. 8.3).

The Blokus class contains a function "play_turn" used during a human player's turn or if the user clicks on any new game buttons. It controls the interaction with the human player and ensures that actions are taken in the correct order. For example, the user must choose a piece before placing it on the board. The piece will be unmarked if an illegal move is attempted, and the user must select it again. During the AI's turn, the Blokus class calls a function from the Strategy class. If a player has no moves, the appropriate communicate will be displayed, and the user can still rotate and flip the pieces but cannot play any of them.

To skip a player's turn, the user must click on any non-interactive part of the game window, such as anywhere but on the buttons. Finally, the Blokus class has an "update_screen" function that passes the arguments and calls functions from the Graphics class to update the screen display with up-to-date information.

4.7 Result

The Result object stores information about a piece, including its points and score, and is used to determine the optimal move for each strategy. The class provides functions that manipulate the Results lists to identify the best move. The Results are sorted based on their score, and points are assigned according to their value. The top 20 moves (piece and its points) are then awarded points, each receiving 0.5 points less than the previous one (unless there is a tie). Additionally, the class has a function to allocate extra points based on the size of the piece, which can be used to prioritise larger pieces in the case of a tie.

```
class Result:

    def __init__(self, piece, points, score):
        self.piece  = piece
        self.points = points
        self.score  = score
```

Figure 4.3: Result Object

Furthermore, the class includes a function to combine lists based on their moves, adding up the points for identical pieces and their points and returning the merged list. Lastly, there is a function that performs all these operations, sorts the final list, and returns the move with the most points. It's important to note that in the event of a tie, the first move in the list will be chosen, and this process is wholly randomised since the initial order of pieces in each player's set is randomised. (Aim 3)

4.8 Graphics

The Graphics class functions are invoked during each main loop iteration to ensure that the screen remains updated. These functions are called by the Blokus Class and utilise information from the Blokus, Board and Player objects. For example, the draw_board function loops through each square of the board and displays squares of different colours: white if the square is unoccupied, blue if it's occupied by Player with id = 1, yellow if occupied by Player with id = 2, red if occupied by Player with id = 3, and green if occupied by Player with id = 4. Furthermore, the banner at the top of the screen presents the current round of the game (as indicated by the round attribute of the Blokus object), the number of possible moves (obtained from the current turn Player object), and the action that needs to be taken based

on the number of possible moves or displays the winner once the game is over. Each function displays rectangles of varying colours or text in their appropriate locations. Ensuring that the order of functions is accurate is critical since rectangles are shown on top of one another, resulting in a user-friendly interface.

4.9 Strategy Functions

The strategy functions are responsible for performing all the necessary calculations to make move decisions during the AI's turn, and they are the most complex part of the software. Each function creates an empty list to store Result objects and then checks which pieces can be played in the game's current state. For each piece, the function determines all possible combinations of the piece's points that would result in a legal move. This is accomplished by attempting to play the piece in each of the player's corners for every possible rotation of the piece. Next, the function loops through all legal moves a player can make, creating copies of the Blokus and Board objects to simulate playing the piece. This results in a hypothetical game state that can be evaluated to determine the move's potential impact. The player's or all players' data may be updated based on the evaluation, including score-based evaluation, such as the number of corners. Finally, a Result object containing the considered piece, its points, and the score is created and added to the Results list. Once Result objects have been created for all legal moves, the function returns the array of Results.

4.10 Strategies

The Strategy class is invoked during an AI turn to determine the best possible move. The selection of an appropriate move depends on various factors unique to each strategy. The proper strategy function is called (matching the Player's strategy attribute name - fig. 8.4), and it returns the list with Result objects in which the score depends on evaluated factors. After sorting the Results and assigning points, the move with the highest score is chosen and played. Notably, all the strategies, except for the random strategy, are designed to be greedy (Aim 4).

4.10.1 Strategy Random

The strategy involves the selection of a piece, its rotation, and its location in a completely random manner. While this approach is straightforward, it is considered ineffective since it needs to prioritise exploring or blocking. Moreover, it may lead to the selection of smaller pieces early in the game, which can have significant and irreparable consequences.

4.10.2 Strategy Largest

The strategy entails selecting a piece randomly, with the largest possible size, from the set of legal moves. After choosing the piece, its location and rotation are then determined. Despite its simplicity, this strategy is remarkably effective, as it inadvertently facilitates exploration and blockage of other players while maximising the player's points after the

move. It highlights the importance of placing the largest possible piece, making it an essential component of a strong move in the game of Blokus.

4.10.3 Strategy Corridors

The strategy begins by scanning the game's board for any opponent's path in its range and assessing if it can pass through it. If a corridor is identified, the strategy will prioritise the placement of a piece that exploits this factor. However, if no corridors are within range, the strategy will default to the "Largest" approach to determine the move. Including corridor, detection is advantageous as it enables the player to traverse the opponent's path and access new territories. As a result, identifying corridors can significantly enhance the player's longevity in the game.

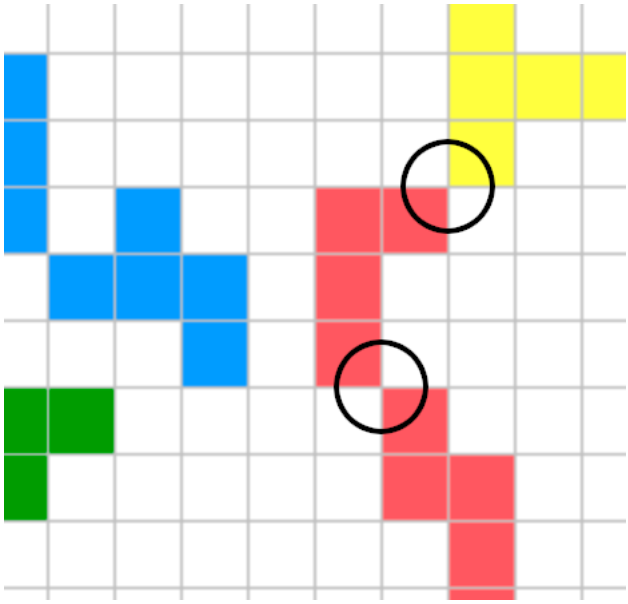

Figure 4.4: Corridors for Blue Player

4.10.4 Strategy Corners

This strategy capitalises on the significance of corners in Blokus gameplay. The strategy generates a Result object for each potential move based on the number of available corners after the move. It aims to select the move that maximises the number of corners. It is important to note that having more corners does not always mean having more possible moves. Therefore, the strategy may prioritise having more corners over exploring and blocking opponents.

4.10.5 Strategy Max Moves

The strategy generates a Result object for each potential move. The score for each move is determined based on the number of feasible moves that will remain after the piece's placement. Consequently, this strategy prioritises playing larger pieces first, with the primary

goal of exploration. However, this approach has a significant drawback, as it may need to consider the strategic placement of the piece, leading to a reduction in possible moves, especially in the late game. Furthermore, it would start the Blokus Duo with a monomino piece as it increases the number of possible moves.

4.10.6 Strategy Min Moves

The strategy evaluates every possible move and assigns a score based on the number of possible moves for opponents after the move. The chosen move is the one that minimises this score, with the primary objective of the strategy being blocking the opponents. This is a powerful strategy since, in Blokus, a defensive move can also be an offensive one. However, the strategy may encounter difficulties if there are no opponents in range to block or if it can only block a small area of the board.

4.10.7 Strategy Max-Min Moves

The strategy evaluates all possible moves and creates a score based on the number of possible moves for players and their opponents. The score equation is calculated as:
*(player's moves) * (number of players)/2 – sum (opponents' moves).*
In Blokus Duo, the strategy values both exploration and blocking equally. In contrast, Blokus Classic prioritises exploration more than blocking to avoid getting blocked by other opponents. The strategy effectively balances exploration and obstruction, making it a strong strategy. However, it inherits some limitations of both "Max Moves" and "Min Moves".

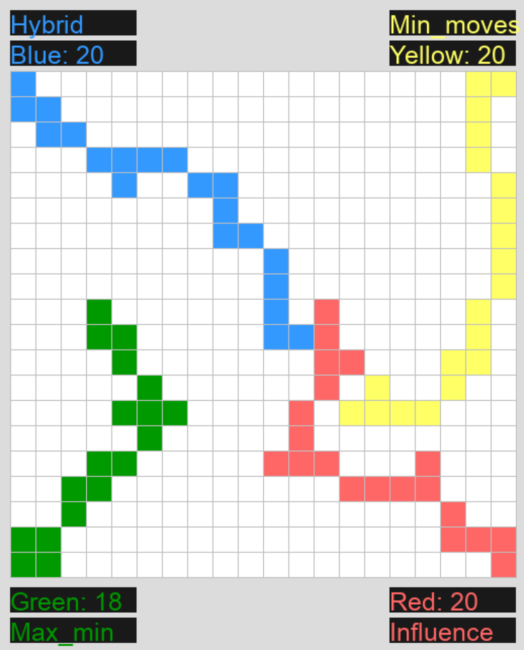

Figure 4.5: Advanced Strategy Game

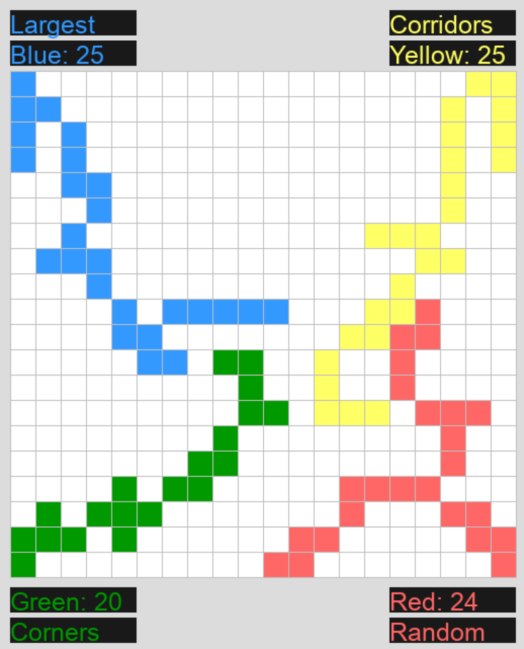

Figure 4.6: Beginner Strategy Game

4.10.8 Strategy Influence

The Influence strategy evaluates every possible move and creates a Result object based on the number of squares the player and their opponents influence. The score for each move is calculated as:
*(player's influence) * (number of players)/2 – sum (opponents' influence).*
This strategy introduces a novel approach to determining the move. It maximises the number of squares the player influences by increasing the number of corners and ensuring that the area around them is unoccupied. Additionally, the strategy introduces a new way of blocking opponents by limiting access to free space around them. This forces opponents to play smaller pieces to compete in such an area. However, the strategy has its drawbacks. It may not place a piece in the area it influences as it would reduce the number of squares influenced. This strategy is effective, but players must weigh its advantages against its limitations in the late game.

4.10.9 Strategy Hybrid

Optimal Finish Function

The strategy can be used in the last few rounds of a Blokus Duo game. It employs brute force to check all possible sequences of moves until the end of the game. The strategy returns a piece, its rotation, and location that will lead to an optimal move considering both players playing the best moves for themselves (Player A maximising the score and Player B minimising). The score value is the difference between the points achieved by Player A and B. This strategy is only applicable in Blokus Duo games from round 16 onwards due to the computational intensity of the brute force algorithm.

Infinite Moves Function

The function is an alternative to the "Optimal Finish function" in Blokus Classic. In this version of the game, it's not safe to assume that all opponents will play against one player. The function employs a brute force approach to check all possible sequences of moves until the end of the game if a player can make an infinite number of moves in a row. This approach helps the player find a move that can give them access to an area where they can score more points. The function uses Result objects, allowing it to be incorporated into the Hybrid strategy for a more comprehensive game state analysis.

Strategy Hybrid

The Hybrid strategy is a unique approach that combines various strategies and functions dynamically based on the type and state of the game (Aim 5). The strategy generates multiple lists of results using other strategies and assigns different weights to them. Each list is scored, then all the lists are merged, and the scores for the same moves are aggregated. Finally, the merged list is sorted, and the move with the highest score is chosen. This approach allows the player to balance various factors and select a move that may not excel in every list but is the best overall.

However, the Hybrid strategy faces some challenges. Firstly, determining the correct weights for each list depends on the current state of the game, which can be complex and dynamic. There may need to be more than just the round of the game to accurately assess the game's state. Additionally, the point assignment for the Result object is based on the ranking rather than the actual score. Thus, the point assignment may not reflect large differences in scores between Results. A more effective approach could be to normalise the scores and assign points based on the actual score rather than position in the ranking.

4.11 Problems Encountered During Strategy Development

The goal of creating various strategies was to generate robust strategies and explore different possibilities in the Blokus game. Thanks to detailed planning, the implementation of strategies went smoothly with few errors. However, some issues were encountered during implementation. The late addition of the infinite moves function meant that there needed to be more time for thorough planning and execution. It can be programmed using a dynamic array to remarkably reduce the complexity.

4.11.1 Max-Min Moves and Influence Analysis

Moreover, the Max-Min Moves strategy has a significant time complexity, making it unsuitable for quick Blokus games against human players. To conduct a detailed analysis, the Corners strategy was played several times individually on the Blokus Duo and Blokus Classic boards to determine the maximum number of possible corners (fig. 8.5 & fig. 8.6). An analysis was performed to identify the most time-consuming part that needs modification.

Function	worst case	variable
set_of_pieces	21	a
set_of_corners	42	b
possible_rotation_of_a_piece	8	c
set_of_points	336	b*c
piece_size	5	d
add_corners	42	e
all_filter_corners	84	f
count_possible_moves_all	2963520	g
num_of_players	2	h
all_update_influence	1764	i

Figure 4.7: Max_min and Influence Algorithms Analysis

The Complexity of the Max-Min Moves algorithm (fig. 8.7) for Blokus Duo:

$O(a * b * c * (d + e + f + h + g))$

The Complexity of the Influence algorithm (fig. 8.8) for Blokus Duo:

$O(a * b * c * (d + e + f + h + i))$

The difference is "all_update_influence" to "count_possible_moves_all" ('g' to 'i'). The Max-Min Moves strategy was found to be up to 1680 times slower than the Influence strategy due to counting the number of possible moves for every player each time a new piece, rotation, and location are checked.

To address computational limitations, storing more unique data within each piece object is suggested, such as its possible rotations and determining when to use specific pieces. Additionally, identifying the game's critical areas to place a piece would help reduce computational costs, making it feasible to apply depth algorithms. Furthermore, some of the already implemented functions can be optimised.

4.12 General Software Testing During Development

While developing the software, I used the graphical interface to test the functions with general and edge cases. To ensure the proper filtering and manipulation of data, I displayed the data for each function creation.

To evaluate the impact of each function on the user experience, I presented the game to various users, including computer scientists, individuals familiar with the game rules, and random people. Following a brief introduction, I allowed them a few minutes to play the game and asked for feedback on its intuitiveness, visual appeal, and any suggested improvements or missing features. After receiving feedback, I made changes to the software, such as altering the colours of specific blocks, changing the font type, relocating the new game buttons, and modifying the banner above the board.

Additionally, I consulted with users to identify desirable features to add to the game. As a result, I incorporated the hint buttons but decided not to implement the visibility of the chosen piece while dragging the cursor above the board. This decision was made to encourage creativity and abstract thinking, like the chess rule prohibiting players from reversing their moves once they touch and move a piece.

5. Experiments & Analysis

5.1 Experiments

I matched selected AI strategies against each other in different configurations. The number of games in each possible order was the same not to get first-turn bias. To test the strategies, I edited the main loop that initialised specific games in a particular order and counted the number of wins (fig. 8.9).

5.1.1 Blokus Duo

```
---------
Points Largest Strategy:  82
Points Corners Strategy:  16
Draws:  2
---------
```

Figure 5.1: Strategies: Largest vs Corners (100 games)

```
---------
Points Max_moves Strategy:  0
Points Min_moves Strategy:  10
Draws:  0
---------
```

Figure 5.2: Strategies: Max_moves vs Min_moves (10 games)

```
---------
Points Influence Strategy:  5
Points Min_moves Strategy:  4
Draws:  1
---------
```

Figure 5.3: Strategies: Influence vs Min_moves (10 games)

```
---------
Points Largest Strategy:  0
Points Min_moves Strategy:  10
Draws:  0
---------
```

Figure 5.4: Strategies: Largest vs Min_moves (10 games)

```
---------
Points Hybrid Strategy:  10
Points Influence Strategy:  10
Draws:  0
---------
```

Figure 5.5: Strategies: Hybrid vs Influence (20 games)

```
---------
Points Hybrid Strategy:  6
Points Min_moves Strategy:  4
Draws:  0
---------
```

Figure 5.6: Strategies: Hybrid vs Min_moves (10 games)

5.1.2 Blokus Classic

```
---------
Points Largest Strategy:  16
Points Corners Strategy:  8
Draws:  0
---------
```

Figure 5.7: Strategies: Largest (x2) vs Corners (x2) (24 games)

```
---------
Points Max_moves Strategy:  6
Points Min_moves Strategy:  6
Draws:  0
---------
```

Figure 5.8: Strategies: Max_moves (x2) vs Min_moves (x2) (12 games)

```
---------
Points Influence Strategy:  11
Points Max_moves Strategy:  1
Draws:  0
---------
```

Figure 5.9: Strategies: Influence (x2) vs Min_moves (x2) (12 games)

```
---------
Points Hybrid Strategy:  18
Points Max_moves Strategy:  0
Points Influence Strategy:  6
Points Largest Strategy:  0
Draws:  0
---------
```

Figure 5.10: Strategies: Hybrid vs Max_moves vs Influence vs Largest (24 games)

5.2 Analysis

Several conclusions can be drawn from observing the gameplay of Blokus Duo. Playing large pieces first is a critical aspect of a good strategy, and blocking the opponent's moves is more important than exploring. The strategy that minimises the opponent's moves easily won against the one that maximises their moves. Strategies Min_moves, Influence, and Hybrid had close games, with the winner often being the player who started the game. The Hybrid strategy didn't win because it needed to prepare to be blocked so early in the game, and all the players were playing solid moves. By using the mix of strategies Influence, Corridors, and Largest or making it more aggressive in the early stages of the game, it would be possible to make strategy Hybrid win most games against the strategy Influence.

The analysis of strategies in Blokus Classic has shown that playing larger pieces first and moving towards the centre of the board are critical factors for success. The strategy Largest was more effective than the strategy Corners, although the point difference was smaller than in Blokus Duo. The strategies Min_moves and Max_moves tied in a 12-match game, showing that exploring is at least as crucial as blocking. The strategy Min_moves had chances against the strategy Max_moves because every defensive move can be offensive. The strategy Hybrid came first in a 24-match game involving strategies Hybrid, Influence, Max_moves, and Largest. Although its strength was predefined based on the current round number, this result shows the potential of the strategy Hybrid and the importance of balancing optimal factors.

6. Conclusion

The project set out to create an engaging and visually captivating environment for the Blokus board game and implement basic computer strategies. In achieving this goal, the project delivered a platform where human players could challenge AI players in a competitive setting and where multiple basic AI strategies were deployed, culminating in the developing of a sophisticated strategy Hybrid.

By playing Blokus Duo, the most effective strategy is to block the opponent. In Blokus Classic, the player should focus on a balance of blocking and exploring while building a path towards the centre of the board. In both game variations, using the largest available pieces is crucial.

As anticipated, the strategy Hybrid was the strongest, although it only sometimes held a significant advantage over the other strategies. More precise information about the game state beyond the current round number is required to improve it.

Ultimately, the project delivered on its goals and objectives, resulting in a fully functional, interactive Blokus board game and basic and advanced AI strategies. Beyond its gameplay and entertainment value, the project built upon previous research and advanced strategies, providing valuable insights into the game's optimal gameplay.

7. BCS Criteria

The development of a Blokus board game and AI strategies met the six expected outcomes of the Chartered Institute for IT:

Applying practical and analytical skills gained during the degree programme. An Objected Oriented Programming language was utilised to develop interactive software in the project. In addition, the project applied algorithmic knowledge to create AI strategies and analyse their effectiveness and complexity.

Innovation and creativity. The project incorporates a Result object system, enabling the AI player to merge multiple strategies and develop a hybrid strategy. The project's user interface (UI) and user experience (UX) meet the creative goals by offering a user-friendly display throughout the game and guiding the user through the game, assisting them in identifying available positions on the board to play their pieces and communicating the current action required.

Synthesis of information, ideas, and practices to provide a quality solution together with an evaluation of that solution. The project included testing and evaluating the basic strategies to create a balanced advanced strategy. The resulting strategy Hybrid was then tested against the most vital basic strategies and the conclusions drawn from the testing outcomes were documented.

That your project meets a real need in the broader context. The project delves into the game theory of the Blokus board game, where a single definitive strategy cannot be identified as the best. Through testing various strategies against each other, the project aims to draw new conclusions that may be applied to develop a more robust strategy. The Hybrid strategy emerged due to this exploration, providing valuable insights into the game's dynamics.

7.1 An ability to self-manage a significant piece of work

To complete the project, I needed to develop a working, graphical implementation of the board game from scratch and create and test AI strategies. This required strong time management skills. Thanks to detailed planning and constant work, I completed most parts by the planned deadline and even had some time for additional development. Each time

a new objective arose, I identified the necessary sub-objectives and worked on them in the correct order. This approach facilitated fast software creation and minimised errors before testing. The software is designed with attention to detail, making it impossible to terminate unexpectedly and allowing users to interact only with the necessary parts of the game. Additionally, the software was tested with an appropriate set of users, and their valuable feedback was used to implement changes, resulting in a user-friendly game.

7.2 Critical self-evaluation of the process

Although the project development was mainly flawless, I made several mistakes. If I had the same amount of time, I would have focused on implementing a perfect user game experience or developing solid algorithms. The graphical implementation of the game slowed down the algorithms and influenced the suboptimal design for the strategies. While my approach allowed for basic strategies to be implemented and tested carefully, it had many drawbacks for advanced and computationally demanding strategies. Despite spending much time on research, I should have spent even more on it. The lack of projects and information about Blokus Classic was a significant problem, and with more research, I could have set new objectives and developed more robust strategies. Additionally, I should have spent more time planning after each major success or change. I quickly realised that having a detailed plan and design significantly reduced the time needed to implement the solution, resulting in fewer bugs and a better structure.

Moreover, the project could be more scalable. The weakest aspect was the design, which could have been much more straightforward. Although the project was completed successfully, some software components could have been better. Reducing the code in most strategies and optimising some functions is possible. If I were to do the project again, I would have spent more time planning, designing, and creating scalable software without redundant code.

Overall, the project successfully fulfilled all the aims and objectives without any necessary cuts. However, some components could have been of better quality. This project gave me experience in considering more aspects during the planning process and delivering better-quality software in the future.

References

[1] S. N/A, *Blokus rules*, 2007. [Online]. Available: https://web.archive.org/web/20071227074218/http://www.blokus.com/en/regles_plateau.html (cit. on p. 3).

[2] A. "Infamouswhiteknight", *Space and the start of the game (1)*, Apr. 2011. [Online]. Available: http://blokusstrategy.com/space/ (cit. on p. 5).

[3] A. "Infamouswhiteknight", *Big pieces first (2)*, Apr. 2011. [Online]. Available: http://blokusstrategy.com/space-big-pieces-first/ (cit. on p. 5).

[4] C. H. Chao, *Blokus game solver - digitalcommons.calpoly.edu*, Dec. 2018. [Online]. Available: https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1305&context=cpesp (cit. on pp. 5, 9).

[5] A. Jahanshahi, M. K. Taram and N. Eskandari, "Blokus duo game on fpga," in *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013)*, 2013, pp. 149–152. DOI: 10.1109/CADS.2013.6714256 (cit. on p. 5).

[6] N. Sugimoto, T. Miyajima, T. Kuhara, Y. Katuta, T. Mitsuichi and H. Amano, "Artificial intelligence of blokus duo on fpga using cyber work bench," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 498–501. DOI: 10.1109/FPT.2013.6718427 (cit. on pp. 5, 6).

[7] A. Kojima, "Fpga implementation of blokus duo player using hardware/software codesign," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 378–381. DOI: 10.1109/FPT.2014.7082825 (cit. on p. 6).

[8] H. Borhanifar and S. P. Zolnouri, "Optimize minmax algorithm to solve blokus duo game by hdl," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 362–365. DOI: 10.1109/FPT.2014.7082821 (cit. on p. 6).

[9] E. Qasemi, A. Samadi, M. H. Shadmehr *et al.*, "Highly scalable, shared-memory, monte-carlo tree search based blokus duo solver on fpga," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 370–373. DOI: 10.1109/FPT.2014.7082823 (cit. on p. 6).

8. Appendices

```
function strategy(blokus, board, player)
    list_of_results <- strategy_function(board, player, blokus)
    top_result      <- get_top_move([list_of_results])
    update_data(blokus, board, player, top_result.piece)
    blokus.next_player()

function strategy_function(board, player, blokus)
    return_list = []
    valid_piece <- player.valid_pieces()
    for each valid_piece do:
        set_of_valid_points <- valid_piece.valid_rotations()
        for each set_of_valid_points do:
            blokus_copy <- copy(blokus)
            board_copy  <- copy(board)
            player_copy <- copy(player)
            play_piece(blokus_copy, board_copy, valid_piece, set_of_valid_points)
            update_data(blokus_copy, board_copy, player_copy, valid_piece)
            score = (some_formula)
            result = new Result(valid_piece, valid_points, score)
            return_list.add(result)
        end for
    end for
    return return_list

function get_top_move(lists_of_results):
    ret_list = assign_points_concat_and_sum_up_lists(lists_of_results)
    ret_list = sort_results(ret_list)
    return ret_list[0]

```

Figure 8.1: AI Strategy Pseudo Code

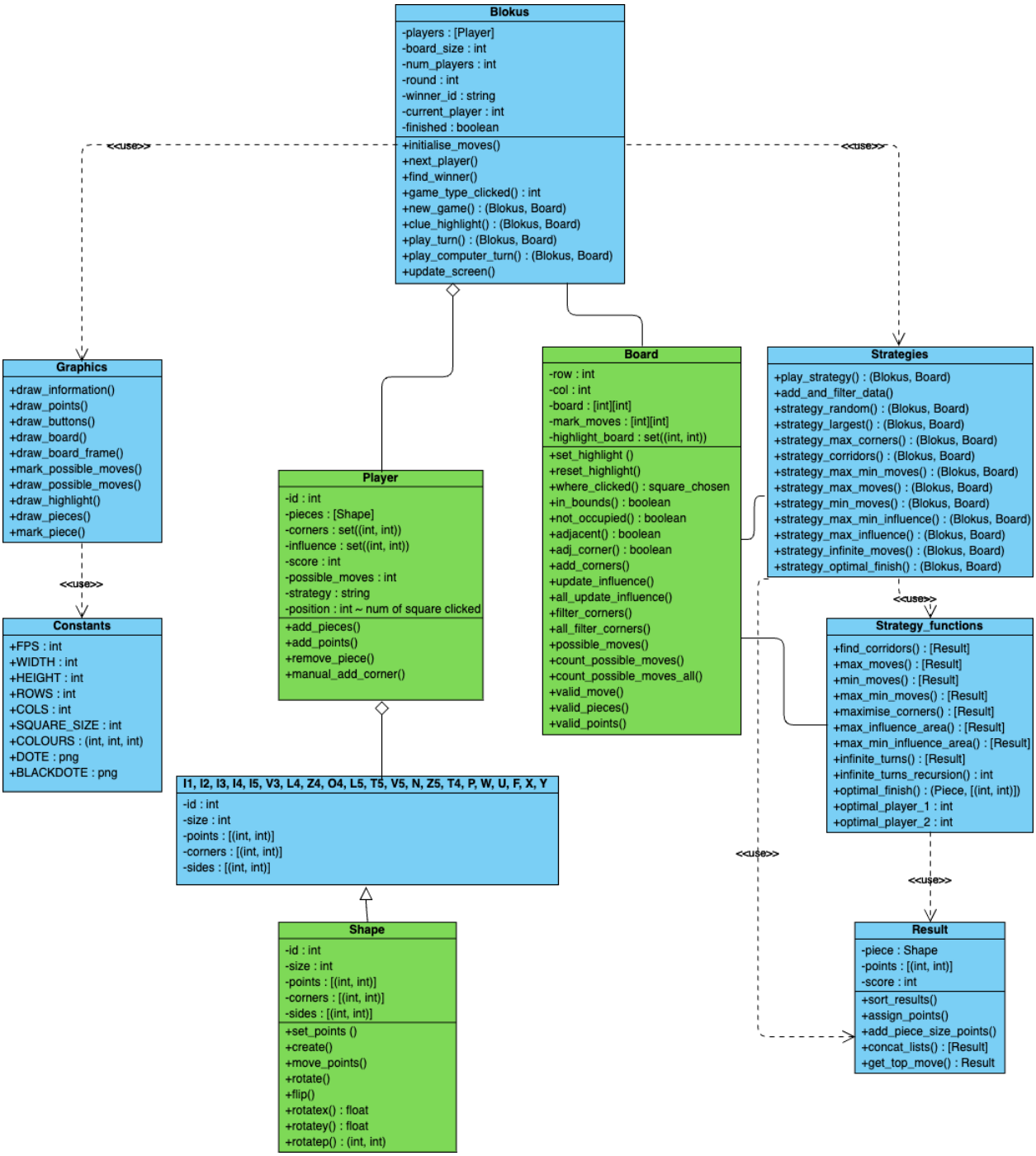

Figure 8.2: Class Diagram

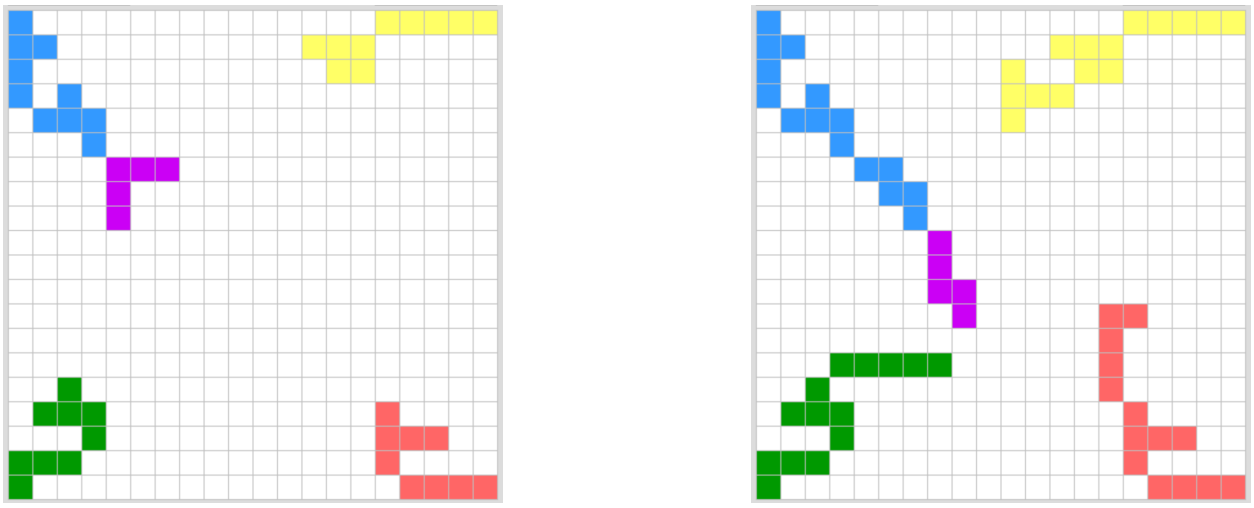

Figure 8.3: Advanced Move Recommendations

```python
class Strategies:

    # Use specific strategy depending on the name
    def play_strategy(blokus, board, strategy_name):
        player = blokus.players[blokus.current_player]
        if player.possible_moves == 0:
            blokus.next_player()
            return(blokus, board)
        else:
            # Call specific strategies, depending on a name
            if strategy_name == "Random":
                Strategies.strategy_random(blokus, board)
            elif strategy_name == "Largest":
                Strategies.strategy_largest(blokus, board)
            elif strategy_name == "Max_moves":
                Strategies.strategy_max_moves(blokus, board)
            elif strategy_name == "Min_moves":
                Strategies.strategy_min_moves(blokus, board)
            elif strategy_name == "Max_min":
                Strategies.strategy_max_min_moves(blokus, board)
            elif strategy_name == "Corners":
                Strategies.strategy_max_corners(blokus, board)
            elif strategy_name == "Corridors":
                Strategies.strategy_corridors(blokus, board)
            elif strategy_name == "Influence":
                Strategies.strategy_max_min_influence(blokus, board)
            elif strategy_name == "Influence +":
                Strategies.strategy_max_influence(blokus, board)
            elif strategy_name == "Hybrid":
                Strategies.strategy_hybrid(blokus, board)
            blokus.find_winner()
        return(blokus, board)
```

Figure 8.4: Computer Player Turn Strategy Choice

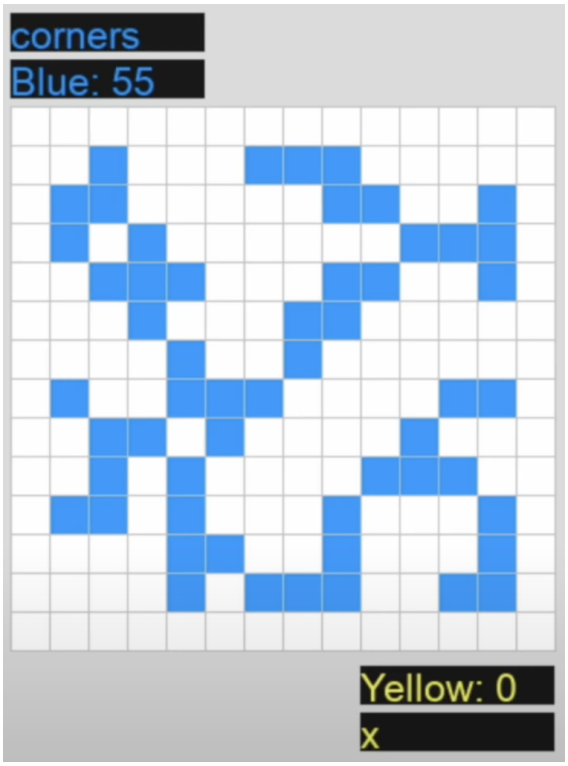

Figure 8.5: The Maximum Number of Corners in Blokus Duo

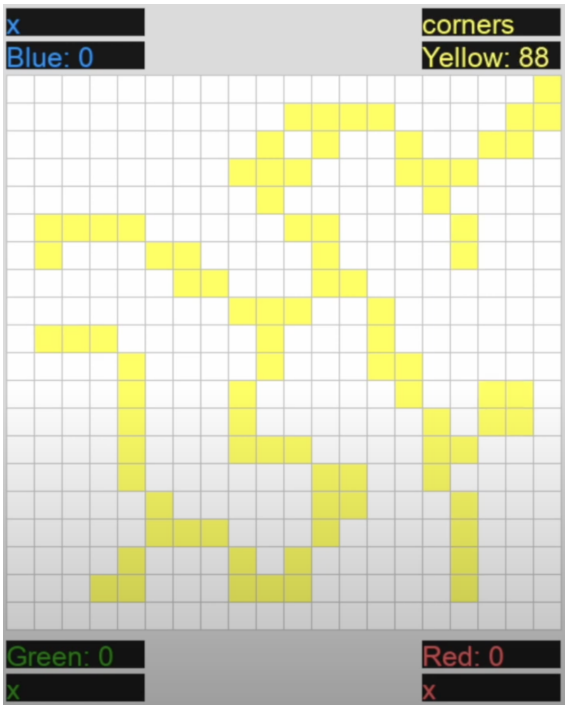

Figure 8.6: The Maximum Number of Corners in Blokus Classic

```
def max_min_moves(board, player, blokus):
        return_list = []
        setOfPieces = board.valid_pieces(player)
        for piece in setOfPieces:
            setOfPoints = board.valid_points(piece, player)
            for points in setOfPoints:
                blokus_copy = copy.deepcopy(blokus)
                copyBoardObj = copy.deepcopy(board)
                player1 = blokus_copy.players[blokus_copy.current_player]
                fake_corner_array = []
                for pt in points:
                    copyBoardObj.board[pt[0]][pt[1]] = player1.id
                    fake_corner_array.append((pt[0]+1, pt[1]+1))
                    fake_corner_array.append((pt[0]-1, pt[1]+1))
                    fake_corner_array.append((pt[0]-1, pt[1]-1))
                    fake_corner_array.append((pt[0]+1, pt[1]-1))
                copyBoardObj.add_corners(player1, fake_corner_array)
                copyBoardObj.all_filter_corners(blokus_copy)
                player1.remove_piece(player.pieces.index(piece))
                copyBoardObj.count_possible_moves_all(blokus_copy)
                resultMoves = player1.possible_moves*blokus_copy.num_players/2
                for opponent in blokus_copy.players:
                    if opponent != player1:
                        resultMoves = resultMoves - opponent.possible_moves
                return_list.append(Result(piece, points, resultMoves))
        return(return_list)
```

Figure 8.7: Max-Min Moves Algorithm

```
def max_min_influence_area(board, player, blokus):
        return_list = []
        setOfPieces = board.valid_pieces(player)
        for piece in setOfPieces:
            setOfPoints = board.valid_points(piece, player)
            for points in setOfPoints:
                blokus_copy = copy.deepcopy(blokus)
                copyBoardObj = copy.deepcopy(board)
                player1 = blokus_copy.players[blokus_copy.current_player]
                fake_corner_array = []
                for pt in points:
                    copyBoardObj.board[pt[0]][pt[1]] = player1.id
                    fake_corner_array.append((pt[0]+1, pt[1]+1))
                    fake_corner_array.append((pt[0]-1, pt[1]+1))
                    fake_corner_array.append((pt[0]-1, pt[1]-1))
                    fake_corner_array.append((pt[0]+1, pt[1]-1))
                copyBoardObj.add_corners(player1, fake_corner_array)
                copyBoardObj.all_filter_corners(blokus_copy)
                copyBoardObj.all_update_influence(blokus_copy)
                resultMoves = len(player1.influence)*blokus_copy.num_players/2
                for opponent in blokus_copy.players:
                    if opponent != player:
                        resultMoves = resultMoves - len(opponent.influence)
                return_list.append(Result(piece, points, resultMoves))
                player1 = copy.deepcopy(player)
        return(return_list)
```

Figure 8.8: Influence Algorithm

```
array = ["Largest", "Corners"]
blokus = Blokus(2, array)
board = Board(14, 14)
wins_1 = 0
wins_2 = 0
draws = 0

round = 100
while round > 0:
    if(blokus.finished):
        if(blokus.players[0].score > blokus.players[1].score):
            if(blokus.players[0].strategy == "Largest"):
                wins_1 += 1
            else:
                wins_2 += 1
        elif(blokus.players[0].score < blokus.players[1].score):
            if(blokus.players[1].strategy == "Corners"):
                wins_2 += 1
            else:
                wins_1 += 1
        else:
            draws += 1
        print("Points Largest Strategy: ", wins_1)
        print("Points Corners Strategy: ", wins_2)
        print("Draws: ", draws)
        print("---------")
        if round > 50:
            array = ["Largest", "Corners"]
        else:
            array = ["Corners", "Largest"]
        blokus = Blokus(2, array)
        board = Board(14, 14)
        round -= 1
    else:
        blokus, board = blokus.play_computer_turn(board)
```

Figure 8.9: Edited Main Loop for Strategy Testing