Blokus Game Solver

Chin Hung Chao
California Polytechnic State University
United States of America
1-(628)-233-1424
cchao06@calpoly.edu

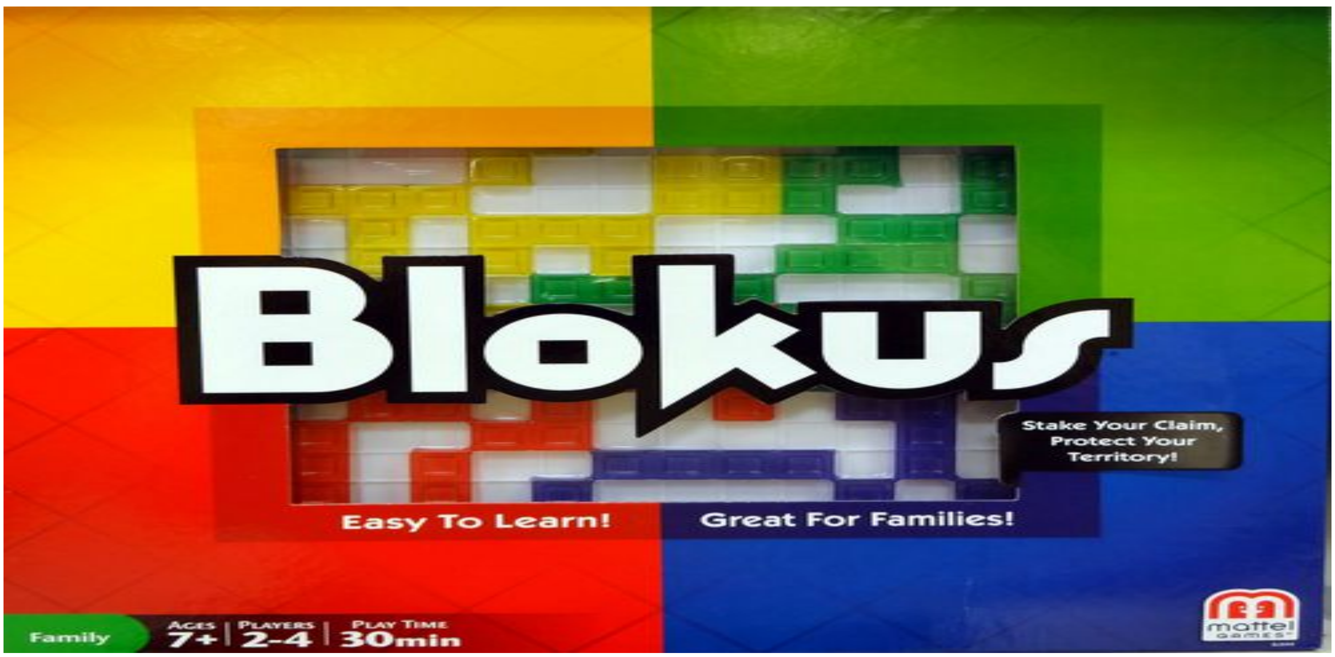

Figure 1: Blokus Game Board Cover

ABSTRACT

Blokus (officially pronounced as "Block us") is an abstract strategy board game with transparent Tetris-shaped, color pieces that players are trying to play onto the board. However, the players can only place a piece that touches at least one corner of their own pieces on the board. The ultimate goal of the game is to place as many pieces onto the board as a player can while blocking off the opponent's ability to place more pieces onto the board. Each player has pieces with different shapes and sizes that can be placed onto the board, where each block within a piece counts as one point. The player that scores the highest wins the game.

Just like other strategy board game such as chess, Blokus contains definite strategic patterns that can be solved with computer algorithms. Various algorithms were discovered and created to develop winning strategy and AI against human opponents. In this work, I am developing random and different greedy strategies to analyze the effectiveness of different factors such as pieces' size, corner availability, and first-player turn.

1 INTRODUCTION

Blokus is an abstract strategy board game like Chess and Chinese checkers. They are board games that can be solved purely with algorithms, which means no randomness or intuition. Every move can be determined based off of various board and game conditions such as current scores and total pieces used. Therefore, AI can be developed to effectively defeat human players in these board game, which allows people to formulate strategies that have a high probability of winning rate [1].

I develop a Blokus game solver to study game algorithms and AI along with exploring a new board game.

Due to abstract strategy board games having solvable patterns, an AI with human behaviors can be developed and evolved as more games are played [1].

There are various known strategies that can be used by the AI to solve abstract strategy board game. The most basic strategy would be simply randomly placing pieces on valid location on the board, which is not an effective strategy since the winning rate is purely dependent on chance. The Greedy algorithm and minimax algorithm are two known strategies to solve these types of strategy games. In Blokus, a basic greedy algorithm in general would make the player put down the largest pieces first before putting the smaller one. This strategy allows the player to get higher scores early on in the game while decrease the number of places on the board the opponent can use. However, there are variations of greedy algorithm that can be implemented in order to improve the effectiveness of the algorithm. By taking account of other variables in Blokus, the effectiveness of the algorithm can be improved.

A more complex decision-making algorithm would be minimax algorithm. It creates a search tree of players making alternate moves with different reward values at the leaf nodes, then the players will choose the child with the best rewards for their cases [2]. Each tree level alternate between different players. However, due to the number of possible moves in Blokus, a complete search tree will hit the memory limitations. Also, the amount of time required to calculate the best moves are computationally expensive. Therefore, the minimax algorithm usually creates a search tree only up to certain number of moves. Techniques such as alpha-beta pruning help decrease the time needed to search the optimal moves by reducing the branching factor in the search tree [3].

Different algorithms represent different strategies that can be used by human players. However, no strategies are perfect, especially in complex strategy game like Blokus. Some strategies do give higher chance of winning but introduce extreme complexity such as the minimax algorithm. While others are simple but not generally effective. Other strategies are effective but contain limitation. Each strategy can also have variations. Monte Carlo tree search is an algorithm derived from the minimax algorithm which can be used for Blokus as well [2].

For the algorithm analysis in Blokus, I am focusing on the effectiveness of different algorithms on winning in Blokus. The analysis includes the different factors that can be used in the algorithm when calculating optimal moves and the effectiveness of algorithms in comparison with one another. The players' turn is switched to determine whether a first-player advantage exists in Blokus or not.

2 BACKGROUND

Blokus was invented in year 2000 and released by a French company, Sekkoia. It has won many awards such as the 2003 Mensa Select [4]. This game has a deep resemblance to Tetris due to the shapes and blocks of the game pieces despite the big difference in game mechanics and genre. Also, the pieces allow some other various interesting studies to be made beside the actual game play like how squares can be covered on the board without any pieces touching each other [5].

2.1 How to Play

Blokus can be play by 2 to 4 people. The board is usually 20 rows by 20 columns; however, two-player version has 14 rows by 14 columns. Each player has 21 difference game pieces that can be place on the game board. The goal of the game for each player is to place as many of his or her own pieces onto the board as possible without violating any placement rule [4].

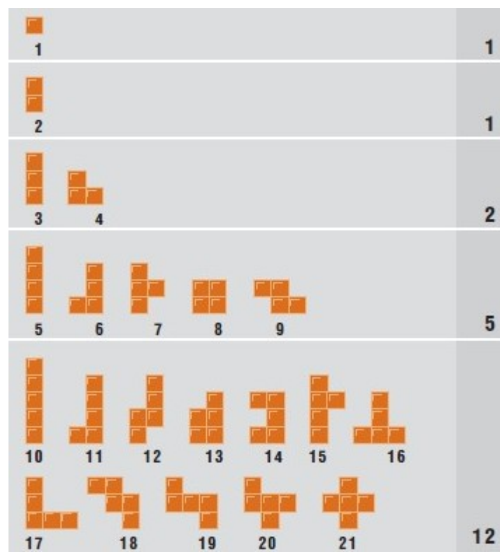

Figure 2: The 21 pieces that each player has at the beginning of the game, which can be rotated and flipped.

The general overview of Blokus gameplay is as follow:

1) Each player needs to first place one of his or her piece at one of the corner regions (one square in the piece must touches the corner square)

2) Each player takes turn putting down pieces onto the board.

3) Each new piece puts down by the player must touch at least one corner of his or her other piece.

4) No flat edge should touch one another for each player's pieces (Different player's pieces can touch one another).

5) Whenever one of the players is unable to place anymore pieces onto the board, the player must pass his or her turn.

6) The game ends when both players cannot put down anymore pieces onto the board.

7) The score is counted by the number of unit squares in the board for each player.

8) If a player places all of his or her pieces onto the board, then he or she gains an additional 15 points.

9) If a player places all the pieces with the smallest piece for the last one, then he or she gains an additional 5 points.

10) The player with the highest point wins the game.

2.2 Blokus Version for Algorithm Study

For this study, the Blokus is based off of the Travel Blokus/Blokus Duo version, which is 2 players with 14 by 14 size board instead of the original 20 by 20 size. The minor modification is the players shall start at the opposite corners instead of the center of the board. This version allows a better strategic study due to no possibility of teaming up, which can happen for 3-4 players. The smaller size also helps demonstrate the effectiveness of different strategies in limited spaces. The limited spaces also mean getting the game bonus for any player is not feasible; therefore, it is not taken account in the algorithm.

Figure 3: Blokus Duo game board, the board size the game solver algorithm uses, with the player starting at opposite corners instead of the center.

The algorithm study focuses on the entire game play in general. The player shall mainly focus on maximum piece placements on the board. The two main algorithms in this study are the random and greedy algorithms.

2.3 Random Strategy

The player who utilizes random strategy randomly chooses a piece to put onto the board in a valid configuration. The randomness may provide unpredictability in the player's move. However, it also means the player might place low priority piece first. The algorithm itself is very simple, but it is too unreliable to be an effective agent for Blokus.

2.4 Greedy Strategy

The player who utilizes greedy strategy place pieces based on the overall weight of different factors. Some pieces will have higher priority than others depending what is used to calculate their importance. The weight determines how beneficial a piece placement is. The piece with the highest weight will be chosen to be placed. However, if that piece cannot be place in any valid configuration, the next highest one is chosen and so on.

The factors that are used for the algorithm in this study are the size of the piece and the total corner difference between the players after a piece placement.

2.4.1 Size. The main focus of the greedy strategy. The main purpose of Blokus is to place as much unit block onto the board as possible. Placing larger size piece first is the fastest way to get higher scores quickly. The wider available spaces also make large size easier to place earlier than later. Having large size pieces later run the risk of them being left over due to no valid placement configuration [6].

2.4.2 Total Corner Difference Between Players. The capability to put a new piece onto the board for a player is partly determined by how many corners the players can try to find valid configurations. The more corners, the higher the chance of finding at least one valid configuration. Since the placement of a piece can also cut off the opponent's corners, opponent's corners should be taken into account of as well [6]. If a player has a higher available corner to place pieces than the opponent, the player then has an advantage in terms of possible board placement location and blocking other player's corners.

Due to the size actually matters directly to the score while the corner indirectly affects the possible score, the greedy strategy weighs the size of the piece more than the total corner difference. An advanced greedy strategy involves maximizing the player's own score while keeping its total available corners higher than the opponent. But the calculation of the corner difference increases time complexity of the greedy algorithm overall.

2.5 Other Strategies

Greedy strategy is simply one of the sophisticated algorithms to use. Strategies such as the Monte-Carlo and minimax are other alternatives that are effective as well.

Other strategy involves placing specific piece at specific region to cut off the opponents' movement. The Barasona Opening is a starting diagonal board placement and cut off strategy that is meant to split the board to reduce mobility against the opponents. The pieces used are usually very hard to put on the board effectively.

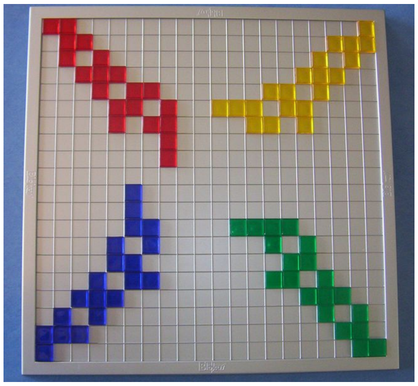

Figure 4: The Barasona Opening Used by All Players

While these other strategies are not used for this algorithm study, they are worth noting for their effectiveness to win in Blokus.

3 GAME IMPLEMENTATION

The Blokus simulation is developed in Python 3. The program allows 2 to 4 AIs to play a game of Blokus. The board size can be changed as well. For the purpose of this study, the board size will be 14 by 14 with only two AIs playing the game. The AIs can play Blokus using different strategy that is developed in program. They are random and two variations of the greedy strategies.

3.1 Blokus Pieces

The Blokus pieces are developed under a defined class *Shape* with different id to distinguish different pieces from each other. Each piece contains a list of points to determine to its attempted placement position and a list of corners of the pieces for checking valid placement based on the rule of Blokus. The pieces also support methods that perform the rotation and flipping of different pieces using a reference point on the pieces. A method is defined for each piece to set its points and corners using a reference point:

```
def set_points(self, x, y):
   self.points = [(x, y)]
   self.corners = [(x + 1, y + 1), (x - 1,
      y - 1), (x + 1, y - 1), (x - 1, y + 1)]
```

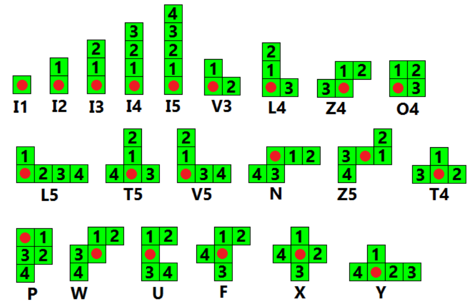

Figure 5: The id of the piece used in the program. The red dot shows an example of the reference point for each piece, which can be changed.

3.2 Board Validation

A *Board* class is defined to keep track of the state of the Blokus board. The object supports capability of updating the board for different players and print the current board configuration for debugging and viewing game progress. Based off from the official rule of the Blokus game [4], several validation methods are implemented to make sure the players are placing new pieces in a valid configuration. Each piece will be checked for adjacent, corner, overlap, and within-bound placement. The validation will be determined by checking each unit square of a piece. The validation will only be successful if all of them pass.

3.3 Blokus Player

A *Player* is defined to represent different Blokus player in a game. Each has a different player id to distinguish from each other. The players keep track of their available game pieces, current corners they have for piece placement, their scores, and the strategy they are using.

Each player shall calculate its own list of possible valid piece placement based on the current states of the board by going through every available piece, every possible rotation and flip on every possible reference point for each piece. The calculation for possible moves required the most computation in the simulation.

Each player's strategy shall determine what method it uses to determine the next move. Those methods will be algorithms to choose the next move.

3.4 Blokus Game

A *Blokus* is defined to represent an instance of a Blokus game. It keeps track of all the players in game, the current round, the game board, and all the initial starting pieces that are given to players. It also facilitates the simulation of each round of the game, performing all the validation and piece placement of each player.

The *Blokus* class sets up the validation rule for the players to use to determine possible piece placements. An invalid placement shall result in the immediate termination of the game to signify an unexpected error in the simulation that need to be fixed.

The class also determines when the game is finished by checking whether there is any possible move that can be perform by a player in the game. If no player can perform a move, the game is concluded. Then the scores of all players are compared and a winner or a tie is decided.

3.4 Player Strategy

Each different strategy is represented by different method that does the necessary calculation to get the next move for the players. In general, all the methods shall use all the player's pieces available piece to get a list of possible piece placements. Then the priority choice of these placements shall be decided by different methods. The random strategy simply chooses a random placement among the list.

The basic greedy strategy simply chooses a piece with the largest size that can be placed onto the board. Since there are multiple pieces that have size of 5 in the game, the strategy chooses randomly among them. The orientation of the piece is random as well.

The advanced greedy strategy sets up a weight for each possible placement using the piece's size and the total corner difference between players after the piece placement. The player chooses the placement with the highest weight. This removes all the randomness in the decision-making process for the piece placement. The weight is defined as:

```
weight = 2 * size + corner difference
corner difference = player's corner – opponent's
                    corner
```

Each strategy in the program is defined as a method. Each player has one of these strategies in its parameters. The random, greedy (basic), and greedy (advanced) are called *Random_Player*, *Greedy_Player*, and *Greedy_Player_Two* respectively.

4 SOLUTION STRATEGY

To study the effectiveness of each game play strategy. I compete two AIs with each other using different or same strategy. For each different study, I make the two AIs play a hundred games to determine the win ratio for different strategy. The two different greedy strategies are also matched together to determine the effectiveness of different heuristics. In the event of a tie, the result will not count toward a win for either player.

For a match that involves two different strategies, the order of the players is switched to study the impact of a player's first turn on the strategy. This determines whether a first turn advantage exists for Blokus.

The following lists are the matches I set up for the studies (The left side represents the player that goes first in the game):

- Random vs. Random
- Greedy (Basic) vs. Random
- Random vs. Greedy (Basic)
- Greedy (Basic) vs. Greedy (Basic)
- Greedy (Advanced) vs. Random
- Random vs. Greedy (Advanced)
- Greedy (Advanced) vs. Greedy (Advanced)
- Greedy (Advanced) vs. Greedy (Basic)
- Greedy (Basic) vs. Greedy (Advanced)

5 GAME RESULTS

Two players with the same strategies can be match together as well. This scenario is treated in terms of 'vs. itself'. Each versus match statistic is in the form of:

```
Strategy One vs. Strategy Two: Win, Loss, Tie
```

5.1 Random Strategy

Two players using random strategy were pitted against each other to evaluate the impact of the first turn advantage. The win ratio of the first turn player is **61, 35, 4**.

5.2 Greedy (Basic) Strategy

The match statistic and its turn order for the greedy (basic) player is as follow:

- First Turn vs. Random: **95, 4, 1**
- Second Turn vs. Random: **92, 6, 2**
- First Turn vs. Greedy (Advanced): **64, 32, 4**
- First Turn vs. itself: **59, 38, 3**

5.3 Greedy (Advanced) Strategy

The match statistic and its turn order, if applicable, for the greedy (advanced) player is as follow:

- First Turn vs. Random: **86, 13, 1**
- Second Turn vs. Random: **78, 19, 3**

- First Turn vs. Greedy (Basic): **42, 58, 0**
- First Turn vs. itself: **51, 39, 10**

6 DISCUSSIONS

6.1 Greedy versus Random

The results from the matches show the players using greedy strategies has a clear advantage over the players who place pieces randomly. Both greedy strategies show a winning ratio of above 75% against random strategy no matter if they go first or not.

Since Blokus is a score-based game and placing a larger piece gives a higher score, it provides an incentive to place the largest piece onto the board first. Placing large pieces first has the advantage of getting a higher score early on and taking away a hard piece placement before the chance is lost.

Larger pieces have the general disadvantage of being harder to place on the board due to the space they take up. Once a player lost the chance to put a piece, that player loses many possible points that can be gain. With more than half of the game pieces being large pieces, the loss can quickly stack up. Therefore, putting a larger piece first make sure the players have pieces that are easier to put onto board, even when board is filled up a lot [6]. The greedy algorithm emulates this type of behavior pattern in Blokus.

6.2 Greedy Weight Consideration

The results from the matches show the players using greedy (basic) strategy have a higher chance of winning than the players using greedy (advanced) strategy. By factoring the total available corners into the greedy strategy, the effectiveness of the strategy seems to decrease. While it has a distinct advantage over the players using random strategy, it is not effective against a simpler greedy strategy. However, the score difference tends to be closer. The chance of tie also seems to be higher as a result.

The usage of corner consideration in the algorithm makes it more a greedy strategy than the basic version due to each piece placement has a clear weight value that is different from another same size piece or the same piece with different placement configuration.

The greedy (advanced) strategy does not take into consideration the total corner beyond the next move. Therefore, it is possible a player cuts off another player's access to the other side of the board, which drastically reduce a player's possible future placements. This strategy does not take account of the total future possible placements, which would have more impact, since it would help prevent the above scenario from happening often.

The greedy strategy in general does not involve direct attack and defense against the opponent, it only concerns about its own scores. The additional factors besides score in the greedy algorithm would only be counterproductive unless an active attack and defense is implemented as well.

6.3 First Turn Advantage

Players in Blokus do experience first turn advantage. When two players using the same strategy match against each other, the first player has a higher chance of winning, which is evident in both stochastic and deterministic strategies. In all three strategies, the win ratio of the first player is higher than the second player. Also, deterministic strategies like greedy (advanced) strategy has a higher tie ratio as well.

Considered the Blokus board has only limited spaces, especially with only 14 by 14 size, players usually will not be able to put every piece onto the board. Each piece put onto the board reduces the spaces even further. Therefore, players who get to put their pieces first have more spaces to work with. They get to set the flows of the game by taking away the opponents' options first.

The decision-making may be random, but no matter what piece any player chooses, the board's available space will always reduce.

6.4 Other Considerations

While a general greedy strategy provides a high chance of winning chance in Blokus. There are other specific strategies that can be applied to boost the winning chance even more. The ability to block an opponent's option purposely and put down a complex piece among the same size pieces are keys to gain advantage over other opponents. Certain large pieces such as N, F, W, Y (Pieces ids from *Figure 5*) have various flexible usages that make them crucial in the game [7-8].

The Barasona's opening is an example of reducing the placement capability of other players. If the greedy (advanced) strategy takes the large versatile pieces into considerations, the effectiveness would improve instead of reducing. These strategies would involve more sophisticated algorithm to find the right pieces to use and perform the correct placements.

7 CONCLUSIONS

In summary, I have performed an experimental study of the Blokus game play strategy that has been used by actual players. The theory behind various concepts and factors in Blokus is put in practice to see their impacts in an actual match. The greedy algorithm is shown to be extremely effective in compared to the random algorithm. The simple greedy strategy has a better winning chance than a more complex greedy strategy when no active attack and defense against opponent is involved. The first turn advantage is also a prominent factor in Blokus also, which seems to be a factor that applies to other turn-based abstract strategy board games like Chess and even tiny game like Tic-Tac-Toe.

The development of the Blokus simulation also demonstrates the high time complexity to play multiple games. The different configurations of many board pieces due to the corner availability, rotation, and flip make the algorithms time-consuming. An improvement with the placement search and algorithm needs to be made before an effective AI for Blokus can be developed.

ACKNOWLEDGMENTS

This work was partially supported by Professor Bruce DeBruhl for California Polytechnic State University, Computer Engineering Senior Project in Fall quarter 2018.

REFERENCES

[1] Lou, A. (2012). *Computer vs. Human: Exploring AI in the Game Blokus*. Anna J. Lou. Available at: http://cssf.usc.edu/History/2012/Projects/J1415.pdf

[2] Nijssen, J.A.M. and Winands, M.H.M. (2012). An Overview of Search Techniques in MultiPlayer Games. Computer Games Workshop at ECAI 2012, pp. 50–61, Montpellier, France

[3] Eppstein, D. (1977). *Strategy and Board Game Programming*. [online] UC Irvine Information and Computer Science. Available at: https://www.ics.uci.edu/~eppstein/180a/970422.html

[4] BoardGameGeek. (n.d.). *Blokus*. [online] Available at: https://boardgamegeek.com/boardgame/2453/blokus

[5] Anon, (n.d.). *Blokus Discoveries*. [online] Available at: http://www.gottfriedville.net/blokus/ [Accessed 13 Dec. 2018].

[6] Hart, E. (n.d.). *Strategy: Beginner*. [online] Blokus Strategy. Available at: http://blokusstrategy.com/category/strategy-beginner/

[7] Hart, E. (n.d.). *Strategy: Beginner*. [online] Blokus Strategy. Available at: http://blokusstrategy.com/category/strategy-advanced/

[8] Simon, J. (n.d.). *The Most Important Pieces in Blokus*. [online] Magmic. Available at: https://magmic.com/the-most-important-pieces-in-blokus/

APPENDIX

A BLOKUS SIMULATION PROGRAM

The Blokus simulation is developed and ran in Python 3.

A.1 shape.py

```
import math

# Get the new x value of point pt (x, y) rotated about reference point
# refpt (x, y) by degrees clockwise
def rotatex(pt, refpt, deg):
   return (refpt[0] + (math.cos(math.radians(deg)) * (pt[0] - refpt[0]))
      + (math.sin(math.radians(deg)) * (pt[1] - refpt[1])))

# Get the new y value of point pt (x, y) rotated about reference point
# refpt (x, y) by degrees clockwise
def rotatey(pt, refpt, deg):
   return (refpt[1] + (-math.sin(math.radians(deg))*(pt[0] - refpt[0]))
      + (math.cos(math.radians(deg)) * (pt[1] - refpt[1])))

# Get the new point (x, y) rotated about the reference point refpt (x, y)
# by degrees clockwise
def rotatep(pt, refpt, deg):
   return (int(round(rotatex(pt, refpt, deg))),
      int(round(rotatey(pt, refpt, deg))))

# The Shape class
# Each difference game piece is a subclass of shape
# Each has a different id and specific total amount of block (size)
# points represent the shape of the pieces
# corners represent the corners to the piece
class Shape:
   def __init__(self):
      self.id = None
      self.size = 1

   # Set the shapes' point (x, y) locations on the board
   def set_points(self, x, y):
      self.points = []
      self.corners = []

   # Create the shapes on the board, num = square index of the piece
   # pt = reference point
   def create(self, num, pt):
      self.set_points(0, 0)
      pm = self.points
      self.pts_map = pm

      self.refpt = pt
      x = pt[0] - self.pts_map[num][0]
      y = pt[1] - self.pts_map[num][1]
      self.set_points(x, y)
```

```

    # Returns the points that would be covered by a
    # shape that is rotated 0, 90, 180, of 270 degrees
    # in a clockwise direction.
    def rotate(self, deg):
        self.points = [rotatep(pt, self.refpt, deg) for pt in self.points]
        self.corners = [rotatep(pt, self.refpt, deg) for pt in self.corners]

    # Returns the points that would be covered if the shape
    # was flipped horizontally or vertically.
    # orientation = 'h' (horizontal) or 'v' (vertical)
    # NOTE: For this project, vertical flip isn't needed
    def flip(self, orientation):
        # flip horizontally
        def flip_h(pt):
            x1 = self.refpt[0]
            x2 = pt[0]
            x1 = (x1 - (x2 - x1))
            return (x1, pt[1])

        # flip the piece horizontally
        if orientation == 'h':
            self.points = [flip_h(pt) for pt in self.points]
            self.corners = [flip_h(pt) for pt in self.corners]

# List of all the 21 Blokus shape objects
class I1(Shape):
    def __init__(self):
        self.id = 'I1'
        self.size = 1

    def set_points(self, x, y):
        self.points = [(x, y)]
        self.corners = [(x + 1, y + 1), (x - 1, y - 1), (x + 1, y - 1),
            (x - 1, y + 1)]

class I2(Shape):
    def __init__(self):
        self.id = 'I2'
        self.size = 2

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1)]
        self.corners = [(x - 1, y - 1), (x + 1, y - 1), (x + 1, y + 2),
            (x - 1, y + 2)]

class I3(Shape):
    def __init__(self):
        self.id = 'I3'
        self.size = 3

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x, y + 2)]
```

```
        self.corners = [(x - 1, y - 1), (x + 1, y - 1), (x + 1, y + 3),
            (x - 1, y + 3)]

class I4(Shape):
    def __init__(self):
        self.id = 'I4'
        self.size = 4

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x, y + 2), (x, y + 3)]
        self.corners = [(x - 1, y - 1), (x + 1, y - 1), (x + 1, y + 4),
            (x - 1, y + 4)]

class I5(Shape):
    def __init__(self):
        self.id = 'I5'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x, y + 2), (x, y + 3), (x, y + 4)]
        self.corners = [(x - 1, y - 1), (x + 1, y - 1), (x + 1, y + 5),
            (x - 1, y + 5)]

class V3(Shape):
    def __init__(self):
        self.id = 'V3'
        self.size = 3

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y)]
        self.corners = [(x - 1, y - 1), (x + 2, y - 1), (x + 2, y + 1),
            (x + 1, y + 2), (x - 1, y + 2)]

class L4(Shape):
    def __init__(self):
        self.id = 'L4'
        self.size = 4

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x, y + 2), (x + 1, y)]
        self.corners = [(x - 1, y - 1), (x + 2, y - 1), (x + 2, y + 1),
            (x + 1, y + 3), (x - 1, y + 3)]

class Z4(Shape):
    def __init__(self):
        self.id = 'Z4'
        self.size = 4

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y + 1), (x - 1, y)]
        self.corners = [(x - 2, y - 1), (x + 1, y - 1), (x + 2, y),
            (x + 2, y + 2), (x - 1, y + 2), (x - 2, y + 1)]

```

```
class O4(Shape):
    def __init__(self):
        self.id = 'O4'
        self.size = 4

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y + 1), (x + 1, y)]
        self.corners = [(x - 1, y - 1), (x + 2, y - 1), (x + 2, y + 2),
            (x - 1, y + 2)]

class L5(Shape):
    def __init__(self):
        self.id = 'L5'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y), (x + 2, y), (x + 3, y)]
        self.corners = [(x - 1, y - 1), (x + 4, y - 1), (x + 4, y + 1),
            (x + 1, y + 2), (x - 1, y + 2)]

class T5(Shape):
    def __init__(self):
        self.id = 'T5'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x, y + 2), (x - 1, y), (x + 1, y)]
        self.corners = [(x + 2, y - 1), (x + 2, y + 1), (x + 1, y + 3),
            (x - 1, y + 3), (x - 2, y + 1), (x - 2, y - 1)]

class V5(Shape):
    def __init__(self):
        self.id = 'V5'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x, y + 2), (x + 1, y), (x + 2, y)]
        self.corners = [(x - 1, y - 1), (x + 3, y - 1), (x + 3, y + 1),
            (x + 1, y + 3), (x - 1, y + 3)]

class N(Shape):
    def __init__(self):
        self.id = 'N'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x + 1, y), (x + 2, y), (x, y - 1), (x - 1, y - 1)]
        self.corners = [(x + 1, y - 2), (x + 3, y - 1), (x + 3, y + 1),
            (x - 1, y + 1), (x - 2, y), (x - 2, y - 2)]

class Z5(Shape):
    def __init__(self):
        self.id = 'Z5'
```

```
      self.size = 5

   def set_points(self, x, y):
      self.points = [(x, y), (x + 1, y), (x + 1, y + 1), (x - 1, y),
         (x - 1, y - 1)]
      self.corners = [(x + 2, y - 1), (x + 2, y + 2), (x, y + 2),
         (x - 2, y + 1), (x - 2, y - 2), (x, y - 2)]

class T4(Shape):
   def __init__(self):
      self.id = 'T4'
      self.size = 4

   def set_points(self, x, y):
      self.points = [(x, y), (x, y + 1), (x + 1, y), (x - 1, y)]
      self.corners = [(x + 2, y - 1), (x + 2, y + 1), (x + 1, y + 2),
         (x - 1, y + 2), (x - 2, y + 1), (x - 2, y - 1)]

class P(Shape):
   def __init__(self):
      self.id = 'P'
      self.size = 5

   def set_points(self, x, y):
      self.points = [(x, y), (x + 1, y), (x + 1, y - 1), (x, y - 1), (x, y - 2)]
      self.corners = [(x + 1, y - 3), (x + 2, y - 2), (x + 2, y + 1),
         (x - 1, y + 1), (x - 1, y - 3)]

class W(Shape):
   def __init__(self):
      self.id = 'W'
      self.size = 5

   def set_points(self, x, y):
      self.points = [(x, y), (x, y + 1), (x + 1, y + 1), (x - 1, y),
         (x - 1, y - 1)]
      self.corners = [(x + 1, y - 1), (x + 2, y), (x + 2, y + 2),
         (x - 1, y + 2), (x - 2, y + 1), (x - 2, y - 2), (x, y - 2)]

class U(Shape):
   def __init__(self):
      self.id = 'U'
      self.size = 5

   def set_points(self, x, y):
      self.points = [(x, y), (x, y + 1), (x + 1, y + 1), (x, y - 1),
         (x + 1, y - 1)]
      self.corners = [(x + 2, y - 2), (x + 2, y), (x + 2, y + 2),
         (x - 1, y + 2), (x - 1, y - 2)]

class F(Shape):
   def __init__(self):
      self.id = 'F'
```

```
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y + 1), (x, y - 1), (x - 1, y)]
        self.corners = [(x + 1, y - 2), (x + 2, y), (x + 2, y + 2),
            (x - 1, y + 2), (x - 2, y + 1), (x - 2, y - 1), (x - 1, y - 2)]

class X(Shape):
    def __init__(self):
        self.id = 'X'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y), (x, y - 1), (x - 1, y)]
        self.corners = [(x + 1, y - 2), (x + 2, y - 1), (x + 2, y + 1),
            (x + 1, y + 2), (x - 1, y + 2), (x - 2, y + 1), (x - 2, y - 1),
            (x - 1, y - 2)]

class Y(Shape):
    def __init__(self):
        self.id = 'Y'
        self.size = 5

    def set_points(self, x, y):
        self.points = [(x, y), (x, y + 1), (x + 1, y), (x + 2, y), (x - 1, y)]
        self.corners = [(x + 3, y - 1), (x + 3, y + 1), (x + 1, y + 2),
            (x - 1, y + 2), (x - 2, y + 1), (x - 2, y - 1)]
```

A.2 blokus.py

```
import sys
import math
import random
import copy
import shape

# Blokus Board
class Board:
    # '_' represents empty square
    # board size: (row: nrow, col: ncol)
    def __init__(self, nrow, ncol):
        self.nrow = nrow # total rows
        self.ncol = ncol # total columns
        self.state = [['_'] * ncol for i in range(nrow)] # empty board

    # Takes in a player id and a move as a
    # list of position (x, y) that represent the piece location.
    def update(self, player_id, placement):
        for row in range(self.nrow):
            for col in range(self.ncol):
                if (col, row) in placement:
                    self.state[row][col] = player_id

```

```
    # Check if the point (y, x) is within the board's bound
    def in_bounds(self, point):
        return 0 <= point[0] < self.ncol and 0 <= point[1] < self.nrow

    # Check if a piece placement overlap another piece on the board
    def overlap(self, placement):
        return False in [(self.state[y][x] == '_') for x, y in placement]

    # Checks if a piece placement is adjacent to any square on
    # the board which are occupied by the player proposing the move.
    def adj(self, player_id, placement):
        adjacents = []

        # Check left, right, up, down for adjacent square
        for x, y in placement:
            if self.in_bounds((x + 1, y)):
                adjacents += [self.state[y][x + 1] == player_id]
            if self.in_bounds((x - 1, y)):
                adjacents += [self.state[y][x - 1] == player_id]
            if self.in_bounds((x, y - 1)):
                adjacents += [self.state[y - 1][x] == player_id]
            if self.in_bounds((x, y + 1)):
                adjacents += [self.state[y + 1][x] == player_id]

        return True in adjacents

    # Check if a piece placement is cornering
    # any pieces of the player proposing the move.
    def corner(self, player_id, placement):
        corners = []

        # check the corner square from the placement
        for x, y in placement:
            if self.in_bounds((x + 1, y + 1)):
                corners += [self.state[y + 1][x + 1] == player_id]
            if self.in_bounds((x - 1, y - 1)):
                corners += [self.state[y - 1][x - 1] == player_id]
            if self.in_bounds((x + 1, y - 1)):
                corners += [self.state[y - 1][x + 1] == player_id]
            if self.in_bounds((x - 1, y + 1)):
                corners += [self.state[y + 1][x - 1] == player_id]

        return True in corners

    # Print the current board layout
    def print_board(self):
        print("Current Board Layout:")
        for row in range(len(self.state)):
            for col in range(len(self.state[0])):
                print(" " + str(self.state[row][col]), end = '')
            print()

# Player Class
```

```python
class Player:
   def __init__(self, id, strategy):
      self.id = id # player's id
      self.pieces = [] # player's unused game piece, list of Shape
      self.corners = set() # current valid corners on board
      self.strategy = strategy # player's strategy
      self.score = 0 # player's current score

   # Add the player's initial pieces for a game
   def add_pieces(self, pieces):
      random.shuffle(pieces)
      self.pieces = pieces

   # Remove a player's piece (Shape)
   def remove_piece(self, piece):
      self.pieces = [p for p in self.pieces if p.id != piece.id]

   # Set the available starting corners for players
   def start_corner(self, p):
      self.corners = set([p])

   # Updates player information after placing a board piece (Shape)
   # like the player's score
   def update_player(self, piece, board):
      self.score += piece.size # update score
      if len(self.pieces) == 1: # If the current piece is the last unused piece
         self.score += 15 # bonus for putting all pieces
         if piece.id == 'I1':
            self.score += 5 # bonus for putting the smallest piece last
      for c in piece.corners: # Add the player's available corners
         if board.in_bounds(c) and not board.overlap([c]):
            self.corners.add(c)

   # Get a unique list of all possible placements (Shape)
   # on the board
   def possible_moves(self, pieces, game):
      # Updates the corners of the player, in case the
      # corners have been covered by another player's pieces.
      self.corners = set([(x, y) for (x, y) in self.corners
         if game.board.state[y][x] == '_'])

      placements = [] # a list of possible placements (Shape)
      visited = [] # a list placements (a set of points on board)

      # Check every available corners
      for cr in self.corners:
         # Check every available pieces
         for sh in pieces:
            # Check every reference point the piece could have.
            for num in range(sh.size):
               # Check every flip
               for flip in ["h", "v"]:
                  # Check every rotation
```

```
                  for rot in [0, 90, 180, 270]:
                     # Create a copy to prevent an overwrite on the original
                     candidate = copy.deepcopy(sh)
                     candidate.create(num, cr)
                     candidate.flip(flip)
                     candidate.rotate(rot)
                     # If the placement is valid and new
                     if game.valid_move(self, candidate.points):
                        if not set(candidate.points) in visited:
                           placements.append(candidate)
                           visited.append(set(candidate.points))
      return placements

   # Get the next move based off of the player's strategy
   def next_move(self, game):
      return self.strategy(self, game)

# Blokus Game class
class Blokus:
   def __init__(self, players, board, all_pieces):
      self.players = players # list of players in the game
      self.rounds = 0 # current round in the game
      self.board = board # the game's board
      self.all_pieces = all_pieces # all the initial pieces in the game
      self.previous = 0  # previous total available moves from all players
      self.repeat = 0 # counter for how many times the total available moves are
                      # the same by checking previous round
      self.win_player = 0 # winner

   # Check for the winner (or tied) in the game and return the winner's id.
   # Or return nothing if the game can still progress
   def winner(self):
      # get all possible moves for all players
      moves = [p.possible_moves(p.pieces, self) for p in self.players]

      # check how many rounds the total available moves from all players
      # are the same and increment the counter if so
      if self.previous == sum([len(mv) for mv in moves]):
         self.repeat += 1
      else:
         self.repeat = 0

      # if there is still moves possible or total available moves remain
      # static for too many rounds (repeat reaches over a certain threshold)
      if False in [len(mv) == 0 for mv in moves] and self.repeat < 4:
         self.previous = sum([len(mv) for mv in moves])
         return None # Nothing to return to continue the game
      else: # No more move available, the game ends
         # order the players by highest score first
         candidates = [(p.score, p.id) for p in self.players]
         candidates.sort(key = lambda x: x[0], reverse = True)
         highest = candidates[0][0]
         result = [candidates[0][1]]
```

```python
         for candidate in candidates[1:]: # check for tied score
            if highest == candidate[0]:
               result += [candidate[1]]
         return result # get all the highest score players

   # Check if a player's move is valid, including board bounds, pieces' overlap,
   # adjacency, and corners.
   def valid_move(self, player, placement):
      if self.rounds < len(self.players): # Check for starting corner
         return not ((False in [self.board.in_bounds(pt) for pt in placement])
            or self.board.overlap(placement)
            or not (True in [(pt in player.corners) for pt in placement]))
      return not ((False in [self.board.in_bounds(pt) for pt in placement])
         or self.board.overlap(placement)
         or self.board.adj(player.id, placement)
         or not self.board.corner(player.id, placement))

   # Play the game with the list of player sequentially until the
   # game ended (no more pieces can be placed for any player)
   def play(self):
      # At the beginning of the game, it should
      # give the players their pieces and a corner to start.
      if self.rounds == 0: # set up starting corners and players' initial pieces
         max_x = self.board.ncol - 1
         max_y = self.board.nrow - 1
         starts = [(0, 0), (max_x, max_y), (0, max_y), (max_x, 0)]

         for i in range(len(self.players)):
            self.players[i].add_pieces(list(self.all_pieces))
            self.players[i].start_corner(starts[i])

      winner = self.winner() # get game status
      if winner is None: # no winner, the game continues
         current = self.players[0] # get current player
         proposal = current.next_move(self) # get the next move based on
                                            # the player's strategy
         if proposal is not None: # if there a possible proposed move
            # check if the move is valid
            if self.valid_move(current, proposal.points):
               # update the board and the player status
               self.board.update(current.id, proposal.points)
               current.update_player(proposal, self.board)
               current.remove_piece(proposal) # remove used piece
            else: # end the game if an invalid move is proposed
               raise Exception("Invalid move by player " + str(current.id))
         # put the current player to the back of the queue
         first = self.players.pop(0)
         self.players += [first]
         self.rounds += 1 # update game round
      else: # a winner (or tied) is found
         if len(winner) == 1: # if the game results in a winner
            self.win_player = winner[0]
            print('Game over! The winner is: ' + str(winner[0]))
```

```
         else: # if the game results in a tie
            print('Game over! Tied between players: '
               + ', '.join(map(str, winner)))

# Random Strategy: choose an available piece randomly
def Random_Player(player, game):
   options = [p for p in player.pieces] # get all player's available pieces
   while len(options) > 0: # if there are still options to find possible moves
      piece = random.choice(options) # get a random piece
      # get a list of all possible moves from that piece
      possibles = player.possible_moves([piece], game)

      if len(possibles) != 0: # if there is possible moves
         return random.choice(possibles) # choose a random placements to use
      else: # no possible move for that piece
         options.remove(piece) # remove it from the options
   return None # no possible move left

# Basic Greedy Strategy: chooses an available piece with the highest size
def Greedy_Player(player, game):
   options = [p for p in player.pieces]
   # order the piece based on highest size first
   options.sort(reverse = True, key = lambda x: x.size)

   while len(options) > 0:
      piece = options[0] # get the largest piece
      possibles = player.possible_moves([piece], game)

      if len(possibles) != 0:
         return random.choice(possibles)
      else:
         options.remove(piece)
   return None

# Advanced Greedy Strategy: chooses an available piece based on a hueristic
# It is based on the piece's size and the total corner difference from
# its placement
def Greedy_Player_Two(player, game):
   shape_options = [p for p in player.pieces]
   board = game.board
   weights = [] # array of tuples, (piece's placement, weight)

   for piece in shape_options:
      possibles = player.possible_moves([piece], game)
      if len(possibles) != 0:
         for possible in possibles:
            # set a test player and board to simulate a future move,
            # then determine the average total available corners difference
            # between the player and its opponents
            test_players = copy.deepcopy(game.players)
            opponents = [p for p in test_players if p.id != player.id]
            test_board = copy.deepcopy(board)
            test_board.update(player.id, possible.points)
```

```
            test_player = copy.deepcopy(player)
            test_player.update_player(possible, test_board)
            my_corners = len(test_player.corners)
            total = 0 # total corner difference between player and each opponent
            for opponent in opponents:
               opponent.corners = set([(x, y) for (x, y) in opponent.corners
                  if test_board.state[y][x] == '_'])
               total += (my_corners - len(opponent.corners))
            average = total / len(opponents) # average corner difference
            weights += [(possible, 2 * piece.size + average)]
   weights.sort(key = lambda x: x[1], reverse = True) # sort by highest weight
   if len(weights) != 0:
      return weights[0][0] # get the highest weighted placement
   else:
      return None # no possible move left

# Play a game of blokus without showing the board
def test_blokus(blokus):
   blokus.play()
   # game continues until a winner (or tied) is decided
   while blokus.winner() is None:
      blokus.play()

# play a round of blokus including printing the board
def play_blokus(blokus):
   print("Round: " + str(blokus.rounds))
   blokus.board.print_board()
   print('=================================================================')
   blokus.play()
   print("Round: " + str(blokus.rounds))
   blokus.board.print_board()
   for player in blokus.players:
      print("Player " + str(player.id) + " score " + str(player.score) + ": "
         + str([sh.id for sh in player.pieces]))
   print('=================================================================')

   while blokus.winner() is None:
      blokus.play()
      print("Round: " + str(blokus.rounds))
      blokus.board.print_board()
      for player in blokus.players:
         print("Player " + str(player.id) + " score " + str(player.score) + ": "
            + str([sh.id for sh in player.pieces]))
      print('=================================================================')

# run multiple blokus games with a strategy for each player
# Precondition: Only two players
def multi_run(printout, repeat, one, two):
   winner = {1 : 0, 2 : 0} # player one and two's scores
   for i in range(repeat): # Play multiple times
      print("New Game " + str(i))
      order = []
      first = Player(1, one) # first player
```

```
    second = Player(2, two) # second player
    all_pieces = [shape.I1(), shape.I2(), shape.I3(), shape.I4(), shape.I5(),
                  shape.V3(), shape.L4(), shape.Z4(), shape.O4(), shape.L5(),
                  shape.T5(), shape.V5(), shape.N(), shape.Z5(), shape.T4(),
                  shape.P(), shape.W(), shape.U(), shape.F(), shape.X(),
                  shape.Y()] # set up all the initial game pieces
    board = Board(14, 14) # 14 by 14 board
    order = [first, second] # order of the player in the game
    blokus = Blokus(order, board, all_pieces)
    if printout: # print or not print the board each round
       play_blokus(blokus)
    else:
       test_blokus(blokus)

    blokus.board.print_board() # print the final board
    blokus.play()
    print("Final Score:")
    plist = sorted(blokus.players, key = lambda p: p.id)

    for player in plist:
       print("Player " + str(player.id) + ": " + str(player.score))
    if blokus.win_player > 0: # if there is a winner, not a tie
       winner[blokus.win_player] += 1 # update the winner's win count
    # print players' win count
    print("Player one win count: " + str(winner[1]))
    print("Player two win count: " + str(winner[2]))
    print()


def main():
   printout = False
   if len(sys.argv[1:]) > 0 and sys.argv[1] == 'print':
      printout = True
   print("Senior Project Blokus Game")

   # For the project, play each competition for 100 games
   # Three possible strategies in the game:
   # Random_Player, Greedy_Player, Greedy_Player_Two
   multi_run(printout, 100, Random_Player, Greedy_Player_Two)


if __name__ == '__main__':
   main()
```