DSC 395T Algorithms and Data Structures | **Individual Project**
Sim | **Programming Assignment #4**

In this assignment, you will **work individually** to implement the optimal strategy for the game of Sim. As is typical with assignments in this course, you are only allowed to use Python standard library and Pytest.

1 The Game of Sim

Sim is a 2-player pen and paper game. 6 dots are drawn, and each dot is connected to each other dot as shown in Figure 1. Each player chooses a color, and then the players take turns coloring the BLACK lines. We will say that the first player is red, and the second player is blue. The object of the game is to avoid forming a triangle strictly from lines of your own color; the first player to do so loses.

Sim is a solved game: The second player can guarantee victory by following an optimal strategy, regardless of the moves of the first player. Your task is to implement this winning strategy.

2 Your Assignment

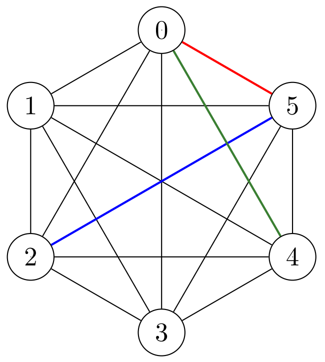
You will create a Sim gameboard (**sim_board.py**) and a player bot (**player2.py**) to complete the provided game implementation. The board simply manages the game state, while the player bot will implement the winning strategy for the second player.

2.1 Graphs

We have not yet discussed graphs in lectures, but to complete this assignment, all you need to know is that a graph is a set of nodes and edges and that graphs are more general than trees. Whereas trees do not allow cycles, graphs have no such restriction. And unlike trees, graphs do not have a designated root node.

2.2 GameBoard

The state of the board will be represented as an **adjacency matrix**. For a graph with n nodes, an adjacency matrix `A` is an $n \times n$ array (or matrix), where the entry `A[i][j]` (A_{ij} in matrix terms) represents the edge between nodes i and j. We will call such an adjacency matrix a `Board`. Note that a `Board` may have edges removed (denoted by setting the edge to `NONE`); these missing edges will be a useful construction as we define the rest of the assignment.



$$\begin{bmatrix} \text{NONE} & \text{BLACK} & \text{BLACK} & \text{BLACK} & \text{GREEN} & \text{RED} \\ \text{BLACK} & \text{NONE} & \text{BLACK} & \text{BLACK} & \text{BLACK} & \text{BLACK} \\ \text{BLACK} & \text{BLACK} & \text{NONE} & \text{BLACK} & \text{BLACK} & \text{BLUE} \\ \text{BLACK} & \text{BLACK} & \text{BLACK} & \text{NONE} & \text{BLACK} & \text{BLACK} \\ \text{GREEN} & \text{BLACK} & \text{BLACK} & \text{BLACK} & \text{NONE} & \text{BLACK} \\ \text{RED} & \text{BLACK} & \text{BLUE} & \text{BLACK} & \text{BLACK} & \text{NONE} \end{bmatrix}$$

Figure 1: A graph and its corresponding adjacency matrix

An **edge** will be represented as a tuple of the two nodes it connects. For example, `(0, 5)` and `(5, 0)` both represent the same edge (colored red in Fig. 1).

Your first task is to implement a GameBoard wrapper class to manage the state of the board.

```
class GameBoard:
    def __init__(self) -> None:
    def color_edge(self, edge: Edge, color: Color) -> None:
    def get_winner(self) -> Color | None:
```

You may assume that `color_edge` will always be called with an edge that exists in the board.

2.3 Player Agent

The bulk of the assignment is to create a player agent that implements the winning strategy for the second player in Sim. You will implement this in the file **player2.py**. Before describing the strategy, we will first define some useful terms.

2.3.1 Definitions

A *move* consists of an uncolored edge and a player color. To take a move means to color the edge, red for player 1 and blue for player 2. Note that `(x, y)` and `(y, x)` represent the same edge and thus the same move.

An **allowed move** is a move that does not immediately result in that player losing. Thus, a **P1-allowed move** is any uncolored edge that can be colored red without forming a red triangle, and a **P2-allowed move** is any uncolored edge that can be colored blue without forming a blue triangle.

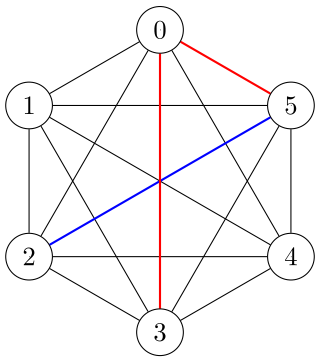


Figure 2: For the above board, `(3, 5)` is a P2-allowed move but not a P1-allowed move. `(4, 5)` is both P1-allowed and P2-allowed.

An **allowed set** is a set of allowed moves that can be applied to the existing board state without the formation of a triangle. As before, a **P1-allowed set** is a set of edges that can be colored red without forming any red triangles, and a **P2-allowed set** is a set of edges that can be colored blue without forming any blue triangles.

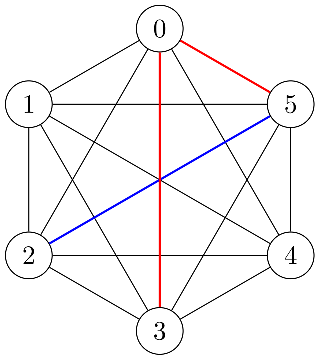
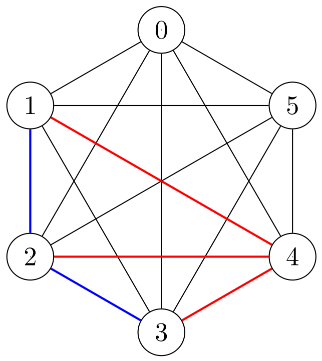


Figure 3: For the above board, `(3, 5)` and `(2, 3)` are both P2-allowed moves, but `{(3, 5), (2, 3)}` is not a P2-allowed set. By contrast, `{(3, 5), (4, 5)}` is a P2-allowed set.

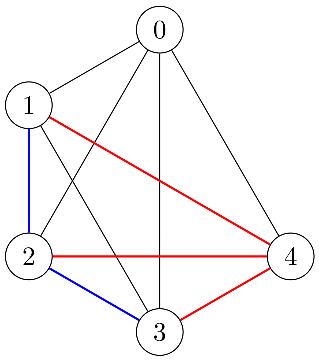
An allowed set **over** a graph can contain only edges that are part of that graph. A **maximal** allowed set (over a given graph) is an allowed set whose size is greater than or equal to the size of all other allowed sets (over the same graph).

A **miniboard** is a subset of the current board. It must:

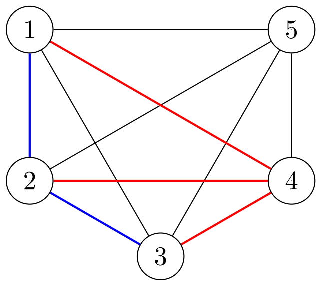
- Contain at least every colored edge on the board, so it must contain any node that has a colored edge.
- Be complete (each node in the miniboard must be connected to each other node).
- Contain at least one P2-allowed move.
- Be the minimal graph that satisfies all the above conditions. If you can remove a node and still have a graph that meets the conditions, your current graph is not a miniboard.



(a) Board state **A**

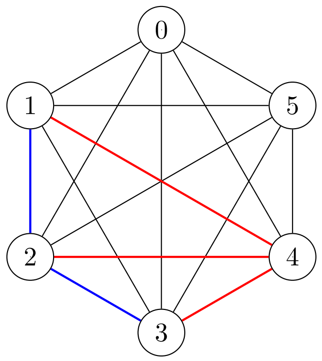


(b) One valid miniboard of **A**

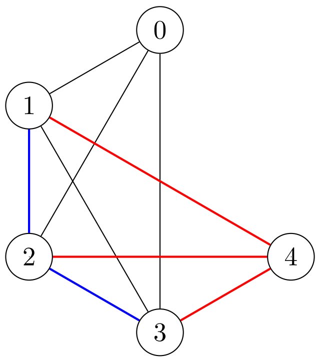


(c) The other valid miniboard of **A**

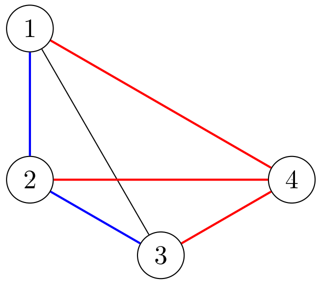
Figure 4: A board and its miniboards



(a) This is **not** a miniboard, since it is not minimal. Either node 0 or node 5 could be removed and the graph would still satisfy all the conditions (see Figure 4a and 4c).



(b) This is **not** a miniboard, because it is not a complete graph. (An edge is missing between nodes 0 and 4).



(c) This is **not** a miniboard, because there are no P2-allowed moves.

Figure 5: Invalid miniboards

2.3.2 The Strategy

Player 2's winning strategy is as follows:

1. Select a miniboard M for the current board state.
2. Consider the set of P2-allowed moves over M. If there is only one, select this move. Otherwise, proceed to the next step.
3. Consider the move(s) contained in the greatest number of maximal P2-allowed sets. If there is only one, select this move. Otherwise, proceed to the next step.

4. Consider the move(s) contained in the greatest number of maximal P1-allowed sets. You may arbitrarily select one of these moves.

Each rule is applied in successive order to break ties for the results of the previous rule. Figure 6 shows an example. To illustrate the case of multiple maximal allowed sets, suppose in step (e), the maximal P1-allowed sets were: `{(0, 1), (0, 2), (0, 3)}` and `{(0, 1), (0, 2), (0, 4)}`, then in step (f), for the moves from step (d), the move `(0, 1)` appears twice (in both sets), the move `(0, 3)` appears once (only in one set), the move `(0, 4)` appears once (only in one set), so we pick the move `(0, 1)` which appears the greatest number of the maximal P1-allowed sets.

2.3.3 Your Task

You will implement the following methods:

```
get_move(Board) -> Edge:
get_miniboard(Board) -> Board | None:
get_maximal_allowed_sets -> set[frozenset[Edge]]:
get_allowed_moves -> list[Edge]:
```

`get_move` returns an optimal move for player 2, as described above. If multiple moves are viable, this method can return any of them.

`get_miniboard` returns a miniboard from the given board state. If there are multiple miniboards, this method can return any of them.

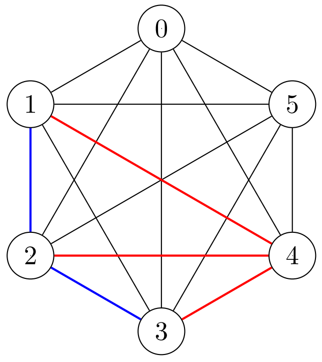
`get_maximal_allowed_sets` returns all the distinct maximal allowed sets for a certain player given a board state.

`get_allowed_moves` returns all distinct moves that are allowed for a certain player given a board state.

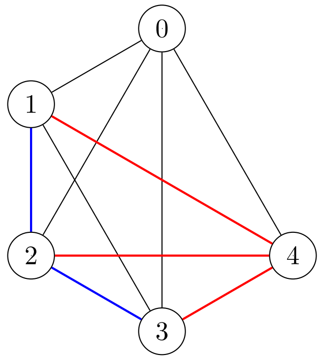
You may assume that if your strategy is implemented correctly, `get_move` will only be called on boards for which a valid move can be found using this strategy.

Acknowledgments. We thank Saffron Liu for creating this assignment. And we thank Joseph Zhu, Neha Biswas, and Ajay Chhawacharia for their helpful comments on this assignment.

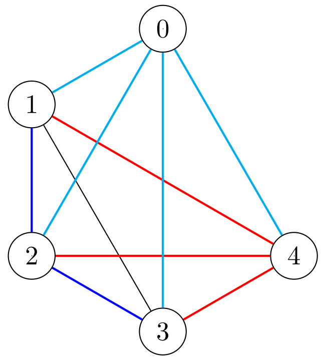
(a) Given board A



(b) Let us select miniboard M (see Fig. 4):

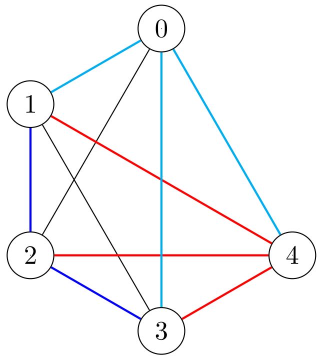


(c) The set of P2-allowed moves is `{(0,1), (0, 2), (0, 3), (0, 4)}`.

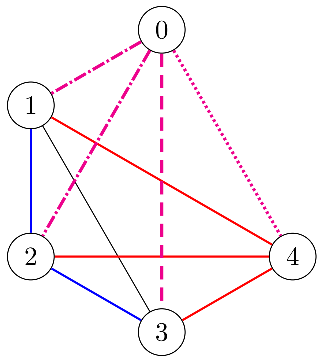


(d) The P2-allowed sets are listed below. Clearly, `{(0, 1), (0, 3), (0, 4)}` is the maximal set. Each of those moves are in the greatest number (1) of maximal P2-allowed sets.

- `{(0, 1)}`
- `{(0, 2)}`
- `{(0, 3)}`
- `{(0, 4)}`
- `{(0, 1), (0, 3)}`
- `{(0, 1), (0, 4)}`
- `{(0, 2), (0, 4)}`
- `{(0, 3), (0, 4)}`
- `{(0, 1), (0, 3), (0, 4)}`



(e) The maximal P1-allowed sets are: `{(0, 1), (0, 2), (0, 3)}`.



(f) Of the moves from step d (`(0, 1)`, `(0, 3)`, `(0, 4)`), we can select either move `(0, 1)` or move `(0, 3)`.

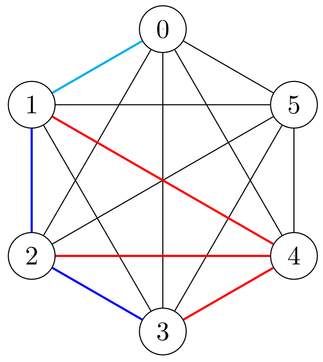


Figure 6: Strategy Example