COL 106 : Data Structures and Algorithms
Semester II, 2022-23, IIT Delhi

Assignment - 6 (due on 21st April, 11:00 PM)

Important Guidelines:

- You must ensure that your submitted code runs in the JDK 11.0.17 environment.
- You are not allowed to share your code with anyone. Cheating of any form will lead to strict disciplinary action. Typical penalty would include Fail grade in the course.
- The assignment must be done individually, and no group submissions are allowed.
- All starter code files must be included in the submission, which must be uploaded on the gradescope without compressing the files.
- The names of files and method signatures must not be changed in the starter code.
- You should write the code in the portion of the starter code labelled **"To be filled in by the student"**.
- Do not declare global variables as your code will run on multiple testcases and hence might give incorrect results when you submit the code on autograder.
- While submitting the code zip the folders Q1 and Q2 into a single file rather than zipping the folder containing Q1 and Q2.

1 Wind Power Grid

Wind Energy is one of the well-known renewable sources of energy. Rotation of turbines at generating stations produces electricity, which is supplied to homes, offices and buildings for consumption. Traditionally, power grids are unidirectional, supply from stations to homes. Wind energy, however, can be produced locally too and thus we need bidirectional lines in the grid so that excess energy can be supplied from local turbines to power deficient locations. This allows for more efficient use of electricity.

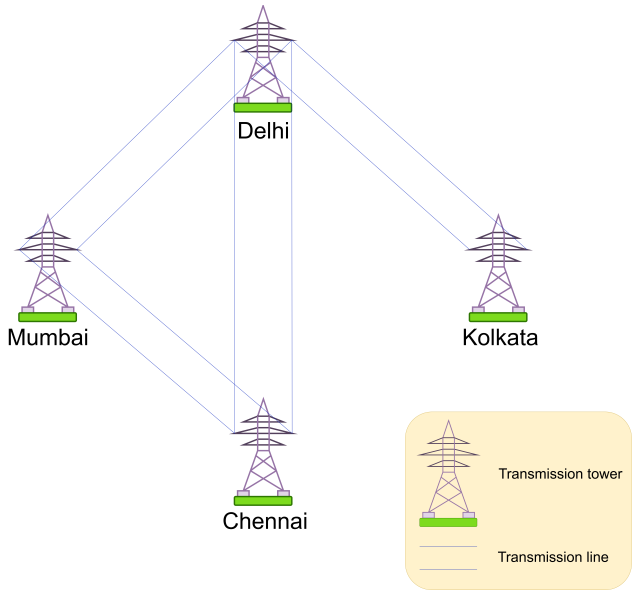


Figure 1: Example: Power Grid 1. Cities: Delhi(D), Kolkata(K), Chennai(C) and Mumbai(M). Transmission lines: D-K, D-C, D-M, M-C

We will work with a simplistic power grid of wind energy: there are `numCities` (n) cities, each with 1 transmission tower, and `numLines` (m) transmission lines connecting these cities. It is given that all the cities are connected to each other by some sequence of transmission lines.

We call a line **critical** if its breakage allows no further transmission of power between the cities it was connecting. In the example given in Figure 1, D-K is a critical line while D-C is not as there is a connection sequence $<D$-M-$C>$ between Delhi and Chennai that doesn't use D-C.

We call a line **important** for a city-pair (A, B) if it appears in all possible paths connecting A and B. In the given example, Delhi-Mumbai (D-M) line is not important for (Chennai, Kolkata) since there is a path $<C$-D-$K>$ connecting Chennai and Kolkata that avoids D-M link. However, D-K line is important for (Chennai, Kolkata) as all possible paths (i.e. $<C$-M-D-$K>$ and $<C$-D-$K>$) use the line D-K.

1.1 Implementation

You will be given two classes `PowerLine` and `PowerGrid`. `PowerLine` has two class variables `cityA` and `cityB` representing the endpoints of the line.

`PowerGrid` has the following class variables:

- `int numCities`: Total number of cities in the power grid.
- `int numLines`: Total number of transmission lines in the power grid.
- `String[] cityNames`: An array to store names of the cities in the power grid, of size **numCities**.
- `PowerLine[] powerLines`: An array of transmission lines, of size **numLines**.

You are allowed to add new private class variables.

Implement the following functions in `PowerGrid`:

- `public PowerGrid(String filename)`: The constructor for the class `PowerGrid`, takes a text file as input and initialises the class variables. The file will have number of cities(n) in the first line, number of transmission lines(m) in the second line, followed by the names of n cities in the next n lines and m pairs of cities in the next m lines. Each pair will be space separated.
  Expected time complexity is $\mathcal{O}(m+n)$.
- `ArrayList<PowerLine> criticalLines()`: This function returns the list of all critical transmission lines in the power grid.
  Expected time complexity is $\mathcal{O}(m+n)$.
- `void preprocessImportantLines()`: This a helper function that you will need to answer `numImportantLines` queries. You can build the required data structures by making use of critical lines computed in the previous function. You are allowed to define a new class in the `PowerGrid.java` file if needed.
  Expected time complexity is $\mathcal{O}(m+n\log n)$.
- `int numImportantLines(String cityA, String cityB)`: This function returns the number of **important** lines from cityA to cityB. If there are no important lines from cityA to cityB, return 0. Note that there will be at least one path from A to B.
  At evaluation time, this function will be called only after `preprocessImportantLines()` has been called once. You might need to do some precomputation in that and populate necessary data structures for an efficient implementation of `numImportantLines`.
  Expected time complexity is $\mathcal{O}(\log n)$.

You are allowed to define new classes and member variables. Do not change the classes and variables given in the starter code.

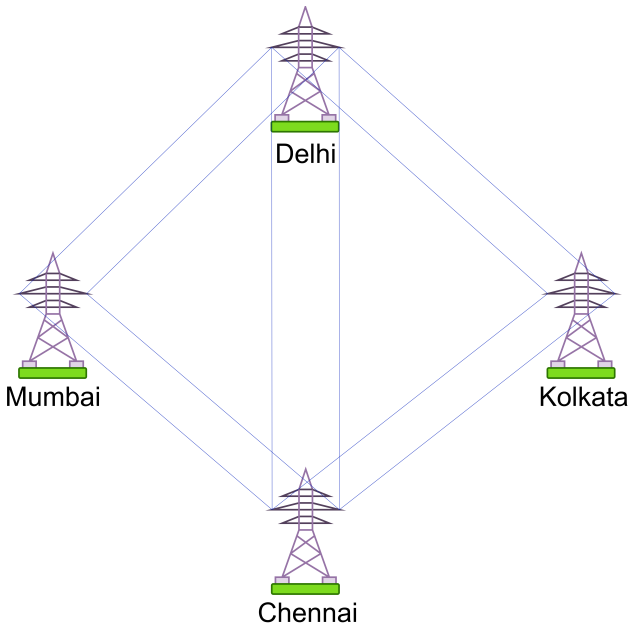


Figure 2: Example: Power Grid 2.

```
Test case 1: For the power grid shown in Figure 2, the input.txt file will
    be as follows:

Input file:
    4           // Number of cities in the power grid
    5           // Number of transmission lines in the power grid
    Chennai     // city
    Delhi       // city
    Kolkata     // city
    Mumbai      // city
    Chennai Delhi       // transmission line
    Chennai Kolkata     // transmission line
    Chennai Mumbai      // transmission line
    Delhi Kolkata       // transmission line
    Delhi Mumbai        // transmission line

PowerGrid pg = new PowerGrid("input.txt");
pg.criticalLines();                         // returns empty ArrayList
pg.preprocessImportantLines();              // must be called before
    numImportantLines()
pg.numImportantLines("Delhi","Chennai");    // returns 0
pg.numImportantLines("Mumbai","Kolkata");   // returns 0
```

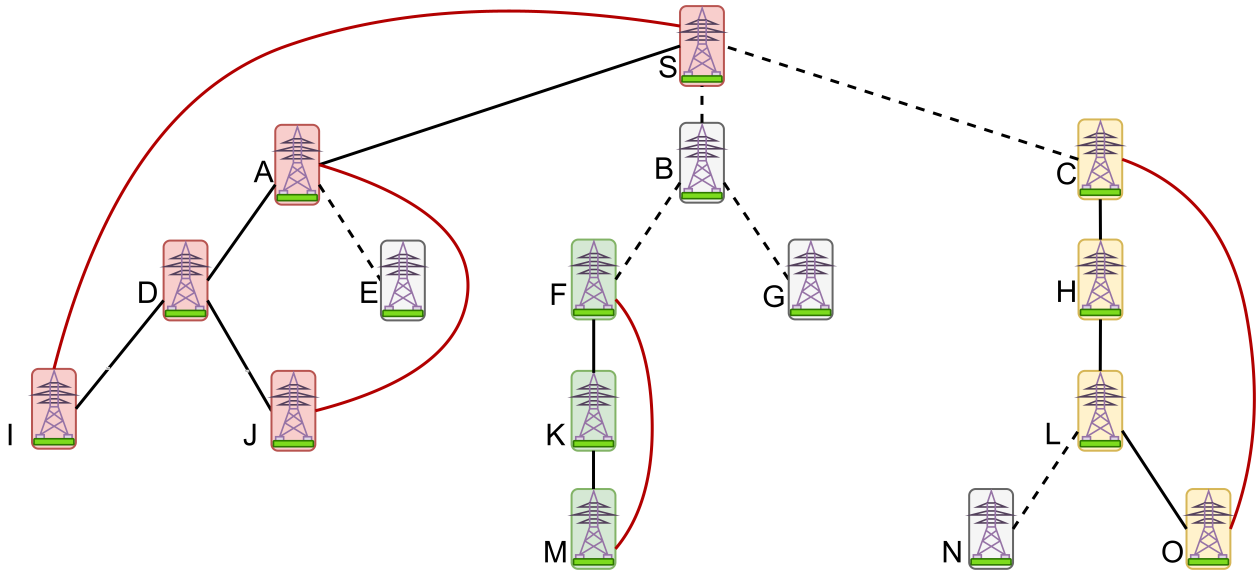


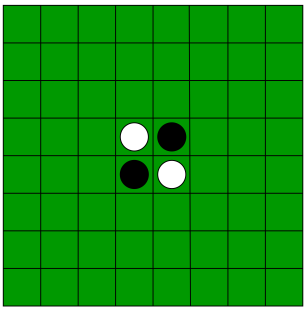
Figure 3: Example: Power Grid 3.

```
Test case 2: For the power grid shown in Figure 3:

PowerGrid pg = new PowerGrid("input.txt");
pg.criticalLines()  // returns returns ArrayList<PowerLine> arr =
    {(A-E),(B-F),(B-G),(B-S),(S-C),(L-N)}

pg.preprocessImportantLines(); // must be called before numImportantLines()
pg.numImportantLines("D","E") // returns 1
pg.numImportantLines("K","N") // returns 4
pg.numImportantLines("H","O") // returns 0
pg.numImportantLines("G","J") // returns 2
```

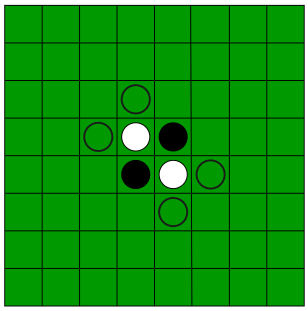
2 Intelligent Strategy for Othello

Although not very well-known, Othello is a very interesting strategy board game for two players. It is played on a 64-tile 8×8 board, starting with 2 white and 2 black pieces arranged in a diagonal manner as shown in the figure. The player with black pieces moves first. The rules are relatively simple, compared to the complexity of the game.

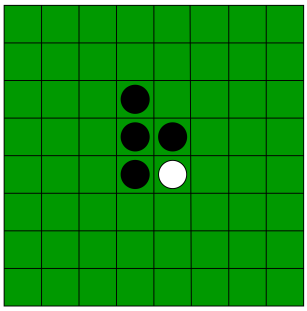
- In any turn, the player to move (for simplicity, say it is the one with black pieces) must place a black piece in such a way that there is at least one contiguous straight line (horizontal, vertical or diagonal) of white pieces between a black piece on the board and the new black piece being placed.



Initial board

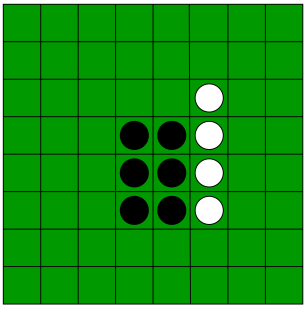


Possible moves (black)

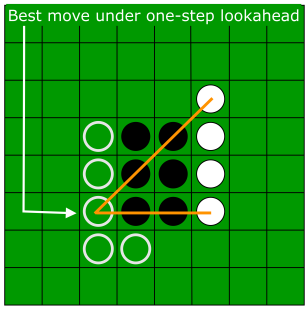


After move white flips

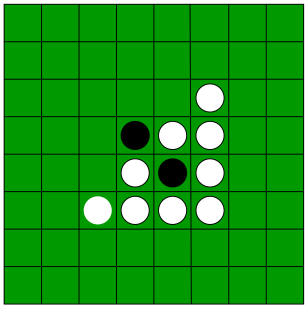
- After a black piece is placed, all the contiguous straight lines of white pieces between the newly placed black piece and any other black piece on the board will be flipped to black.



Board



Possible moves (white)



After move black flips

- After the move, the turn switches to the other player, who plays under the same rules but with colours reversed.
- If a player has no valid move, the turn is passed to the other player.
- The game proceeds until either the board is full or there are no valid moves on the board for both players. At this point, the player with higher number of pieces on the board wins.

One of the simplest and most intuitive approaches of building a good strategy for such turn-based two player games is look-ahead. In one-step look-ahead, from the current board position, the player looks at what the board will look like after all possible sequences of one move (your own), and based on some scoring function and tie-breaking rules chooses the best move. A simple scoring function for the player with black pieces is *number of black pieces minus number of white pieces on the board.* So, the aim of player with black pieces is to maximize this score and the opponent's aim is to minimize the score.

A generalisation to one-step lookahead is k-step lookhead strategy. To also model an intelligent opponent's moves, MiniMax algorithm was introduced. This can be viewed using a decision tree comprising of alternating layers of Max and Min nodes, thus the name. A demonstration of the MiniMax algorithm applied to 4-step look-ahead is given in the figure below.

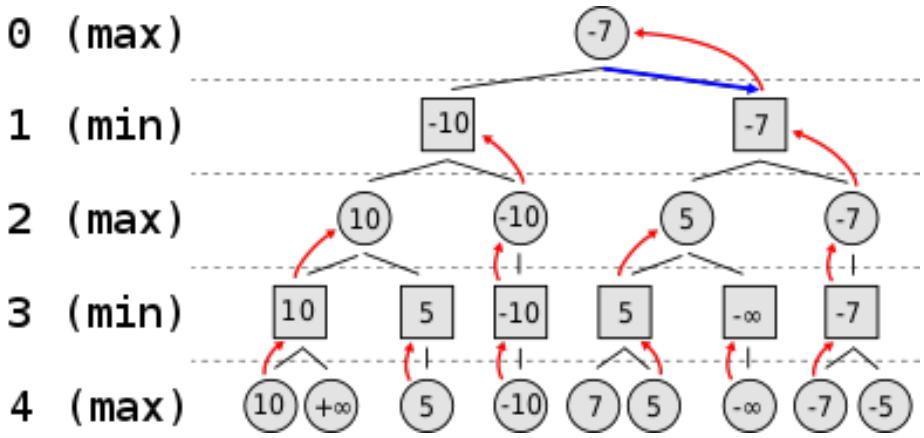


Figure 4: A depiction of 4-step look-ahead in a strategy game where each player has at most two strategies: left and right. Each of the circular nodes in the the decision tree is a Max node and the square nodes are Min nodes. After evaluating the scores upto a depth 4, the scores are propagated upwards based on the function in each node (min or max). At layer-0, i.e. current state, the action is taken which gives the maximum score, and thus the action leading to the right subtree is taken.

Coming back to Othello, your task is to implement an intelligent game-playing agent for Othello which uses k-step look-ahead and MiniMax algorithm to select its moves. For this task, you'll be given a class `Othello` with the following class variables:

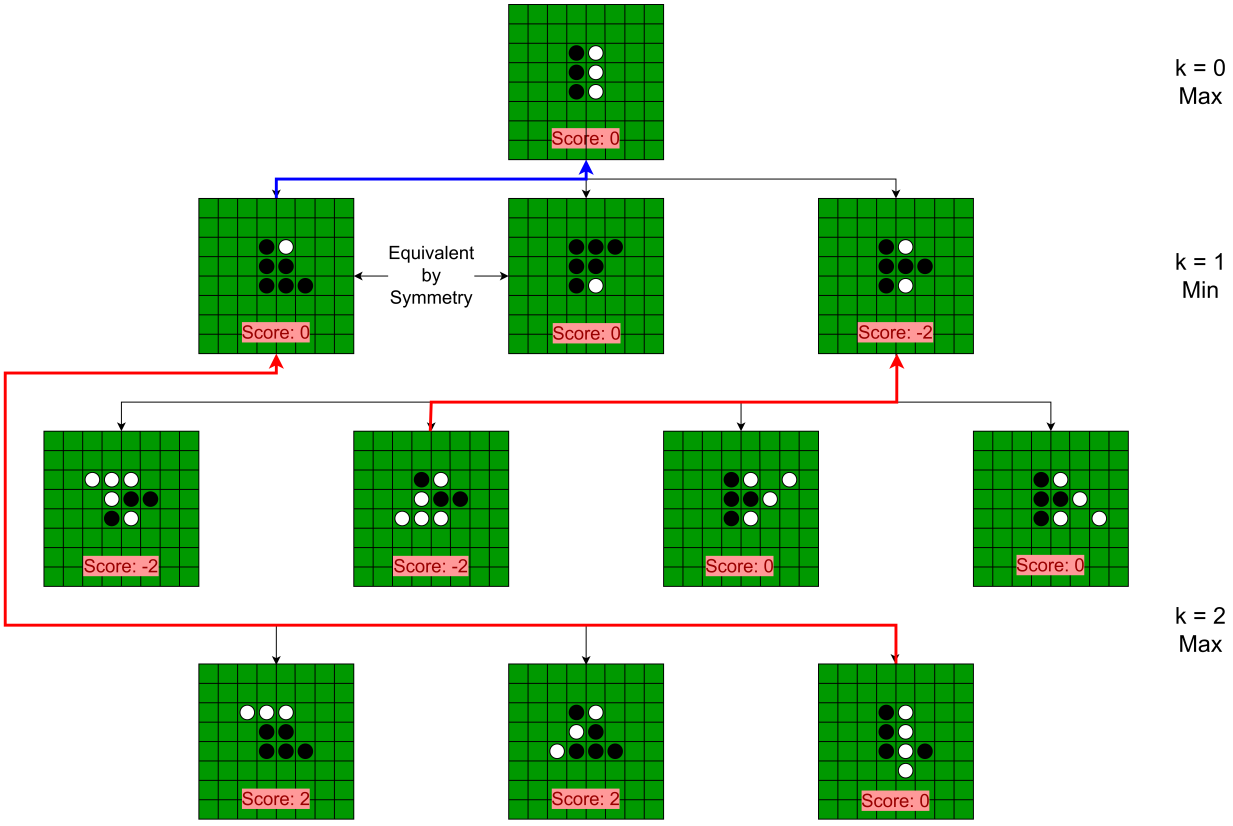
- `int turn`: 0 if it is Black's turn to move and 1 otherwise.
- `int board[8][8]`: A 8×8 array to represent the current board state, using integers -1 for empty tiles, 0 for black pieces, and 1 for white pieces.
- `int winner`: -1 initially, 0 if black wins and 1 if white wins. In case of a draw (both players have equal pieces on the board), keep it as -1.

You will have to implement the following four class functions:

- `public Othello()`: Initializes the board and other class variables based on input given in file `input.txt`. The input will contain 9 lines, first line would contain 1

integer representing the turn. The next 8 lines will contain 8 space separated integers from $\{-1, 0, 1\}$ representing the board starting from `board[0][0]` to `board[7][7]`, row-wise.

- `int boardScore()`: The scoring function for evaluation of boards at the maximum depth of the decision tree. This will be a simple difference of the number of pieces on the board of the player whose turn it is and the opponent's pieces. To clarify, when `turn = 0`, this will be `num_black_pieces - num_white_pieces`. Similarly, when `turn = 1`, this will be `num_white_pieces - num_black_pieces`.
- `int bestMove(int` k`)`: Using k-step look-ahead from the current board, return the best move for the player whose turn it is as an integer. If the best move is to place the piece at `board[`i`][`j`]`, return $(i * 8 + j)$. To break ties, return the smallest $(i * 8 + j)$ value with the maximum score.
- `ArrayList<Integer> fullGame(int` k`)`: Assuming that both players are using k-step look-ahead starting from the current board, return a list of moves encoded as integers of how the game proceeds. Update the board accordingly and set the winner at the end.



A demonstration of 2-step lookahead for Othello with difference scoring function