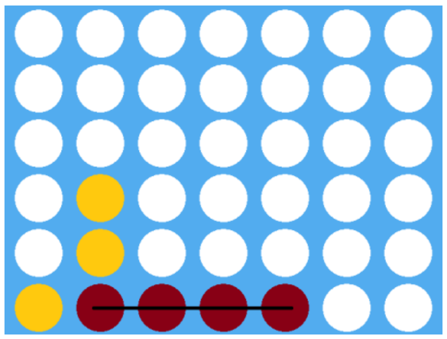
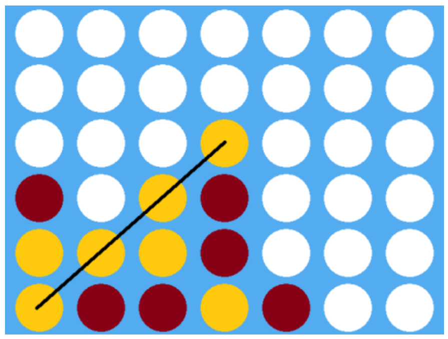
Adversarial Search- Minimax

Your task for today is to develop an AI for the game connect-4.

Rules: Each player takes turns placing coins in columns. The first player to get 4-in-a-row of his coin wins (the 4 coins can be in any horizontal, vertical or diagonal configuration, anywhere on the board)


Red wins!


Yellow wins!

Write a minimax algorithm that takes as input the depth ‘d’ up to which it will search (d=1 means two layers or *plies*: If I play this (max), then you’ll play that(min). d=2 means 4 plies: If I play this(max), then you’ll play that(min), then I’ll play this(max), then you’ll play that(min).

The state representation for this problem is a simple 2D character array with ‘.’ being an empty space, ‘O’ being the AI, and ‘X’ being the enemy. Each action is represented by C, the column number where the coin is dropped. The board will look something like this:

```
--------------                                  --------------
. . . . . . .                                   . . . . . . .
O . . . . . .                                   O . . . . . .
X . . X O . .    generateSuccessor(‘X’,5) -->   X . . X O . .
O . O O X . .                                   O . O O X . .
O . X O X . .                                   O . X O X X .
O X X O X O X                                   O X X O X O X
--------------                                  --------------
```

We’ve provided several handy snippets of code that you can use to solve this problem. They are as follows:

The **State class**— as described above with the following class methods

getLegalActions(agent)—returns a list (list/arraylist/vector) of all possible actions that one can take from a given state (basically, returns an array of all the column numbers that are not full)

generateSuccessor(agent, action)—returns a new State object that occurs when *action* performed by *agent* is applied to the state.

isGoal(agent)—painfully looks through the board for occurrences of ‘XXXX’ or ‘OOOO’ along the rows/columns/diagonals and returns True if it finds any.

evaluationFunction()—Returns the score for a state for minimax to min/max over at the lowest depth.

printBoard()—prints the State’s current board layout for a simple text based visualization.

After you code up a generic Minimax agent, you should be able to play versus it if you do something like this (in java)

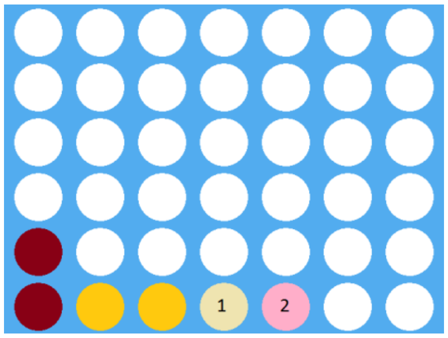
```
MinimaxAgent mma = new MinimaxAgent(depth)
State s=new State(shape)
while(1){
      int action = mma.getAction(s)
      s = s.generateSuccessor('O', action)
      s.printBoard()

      //check if O won? break

      int enemy_move = Scanner.nextInt()
      s = s.generateSuccessor('X', enemy_move)

      //check if X won? break
      //pause
}
```

Implementation note: When breaking ties, please pick the leftmost action generated by getLegalActions(). Even though it would probably be better to select a random move, it’ll be easy to evaluate your agent if you pick the left-most action.

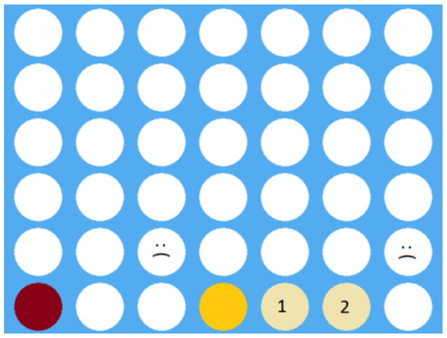
C++ compilation: g++ -std=c++11 state.cpp

Fun stuff:

Once you're done with the minimax algorithm, take some time to play against your AI and see how it does.
Start with depth=1. Notice that the AI does very well blocking you in situations like A, but it's very easy to trap it like in B, letting you win in the next move. This is because the AI only sees one step ahead. It can't tell that in 1 move, you're setting up a guaranteed victory for the second move.

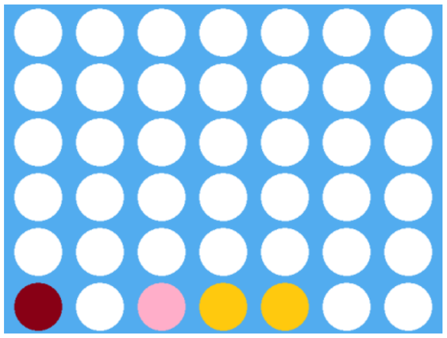


A: If you play 1, the AI will 100% play 2, blocking your potential victory.



B: If you play 1, the AI has no way of seeing 2 as a threat (since it's only searching 1 level) even though you're setting up a guaranteed win.

Increase the depth to 2. Now all of a sudden, the AI can see two moves into the future, and it'll quickly defuse the trap you're trying to set for it by putting a coin at one of the extremes so you don't get a guaranteed victory in 2 moves.


Oh, you sly AI.

Increase the depth to 3. The AI will take nearly forever to search your statespace, so you probably won't get an output any time soon. (Here's where alpha-beta pruning comes in, but we'll leave that out for now)

Evaluation:

To evaluate your solutions, we're going to play against your AI and see if it can:

- **Win** if it has the chance to. ($d \geq 1$)
- **Block** the simple victory mentioned in the first half of the previous section. ($d \geq 1$)
- Anticipate the **trap** described in the second half of the previous section. ($d \geq 2$)

To set up the situations, you can use the following enemy moves:

- **Win**: 3,5,5 (O wins)
- **Block**: 1,2,0,3
- **Trap**: 3,4,0

Extra Credit:

1. Evaluation Function: Modify the trivial evaluation function to include more state-specific features. You should be able to re-use most of the code from the isGoal() function to generate a score. (Hint: remember what we used for tic-tac-toe?)

2. Alpha-Beta pruning: Modify your minimax routine to use alpha beta pruning. This would cut down the state space size by a lot, allowing you to reach deeper into the game tree in the same amount of time. With $d=3$, you'll find some pretty bizarre strategies being played by the AI.
Note: To fully appreciate the benefit of alpha-beta pruning, you should implement a decent evaluation function first.