Class 14: Games: TicTacToe
(and later Nim)
Games

• Games are both an industry and a way to try out algorithms useful in other fields
• We’ll see recursive search, sped up by dynamic programming in use here.
• Thus we are continuing to study algorithm patterns as we study games.
• Project 3 will be on Tic Tac Toe and Nim, 2 well-known games.
One player games

• A one-player game has a goal, and hopefully the player gets closer and closer to that goal.

• CS210 project: Slider puzzle. Many of you used A* search from AI to find the best way to play this.
  • A* works with an expanding set of paths to various nodes n, and works to minimize the cost of the path to the goal
  • Each path to n has a known cost from origin to n of g(n) and also has value h(n), a heuristic measure of the cost from n to the goal.
  • Greedy step: Find least-cost path so far, extend it one step.
One player vs. two players

- A one-player game has a goal, and presumably the player gets closer and closer to that goal.
- With two players, the future is more complicated—the other player has the opposite goal, so a simple measure of progress is not possible.
  - Thus A* isn’t applicable.
  - We have to consider everything the other player could do: the search space is huge.
Games in general: the board

- Games that computers can play usually have a “board” that describes the state of the game
- Slider puzzle: here are 5 board positions and transitions between them:

```java
int tiles[][] = {{1,2,3}, {4, 0, 5}, {7,8,6}};
```

Which one is it?
Tic Tac Toe

Empty = 2, Human = 0, Computer = 1 (Weiss setup)

```c
int board[][] = {{0,1,0},{2,2,2}, {2,2,2}};
```

Which one is it?

See [Wikipedia article](https://en.wikipedia.org/wiki/Tic-tac-toe) for basic coverage
Moves of the Game

- Moves cause changes in board positions, and changes in game state (board + which player next).
- You can tell by querying the game state whether the game is won (and by whom), lost, drawn, or not finished yet.
- There are a finite number of possible game states.
- We can track them with the appropriate data structures.
A move in tic tac toe is given by the coordinates of an unfilled square on the board chosen by the player: (row, column)

We need a value to associate with each possible move so we can choose the best one.

If the move leads to a certain win for the human, the value is 0 (lowest), a draw is 1, win for computer 3 (highest). That’s Weiss’s system (that we’ll use, since we want to use his code.) (Another system -1 (lowest), 0, +1 (highest))

So the computer is seeking the highest possible value when it searches for a move.
Game Trees are explored recursively: slider puzzle case
The tic-tac-toe Game Tree, in part

Note: here the arrows don’t represent data structure references, just player choice actions. We won’t “materialize” the tree.
We can give values to the leaves of the game tree

• A leaf of a game tree means the game is finished—there are no more choices to be made
• We look at the board and decide who wins or if it’s a draw.
• That gives us a value for the leaf: 0 if X wins, 1 if a draw, 3 if O wins (the computer) (Weiss values)
• See next slide...
Partial Game tree with values assigned to leaves

Win for X
Val=0

Win for O,
Val=3

Draw,
val=1

X to play (human)

O to play (computer)
We now have values for the leaf nodes

- We need to percolate values up the tree.
- The computer is seeking to maximize its value, so if it’s the computer’s move, it finds the max value over the possible moves: if the moves reach leaves we have these values already.
- If it’s the human’s move, the min val is chosen.
- This works its way up the tree...
Game tree with values assigned up the tree

Win for X
Val=0

Win for O,
Val=3

Draw,
val=1

O to play (computer), takes max

X to play (human), takes min, finds win

Draw,
val=1

Draw,
val=1

X to play
Try it out: find the value of this board

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>X</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>O</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Draw the 12 descendents of this board, mark the leaves: 3 if O wins, 0 if X wins, 1 if a draw
- Or peek at next slide to get a start.
- Percolate the values up the tree using minimax: max over children if X to play, min over children if O to play.

4/8/2020
• Here are three descendents, try to go further
  Values: 0 if X wins, 3 if O wins, 1 if a draw
Values: 3 if O wins, 0 if X wins, 1 if a draw

Val = 3  Val = 3
Values: 3 if O wins, 0 if X wins, 1 if a draw

4/8/2020
Result: O has a sure win

• O can win for sure if it plays properly
• If O makes mistakes, it could fail to win
Tic-tac-toe Moves

- Game states are represented inside the class by a 3x3 array of int, where the arr[i][j] is 0 for the human player or 1 for the computer player or 2 for unfilled.
- When using a 3x3 array, the player # whose turn it is can be calculated by seeing whether the #0s > #1's.
- The game is over if and only if there are 3-in-a-row in the array. If over, the winner is the player # having the 3-in-a-row.
Game State Example

<table>
<thead>
<tr>
<th>X</th>
<th>O</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>O</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

arr[0][0] = 0,  
arr[0][1] = 1,  
arr[0][2] = 0,  
arr[1][0] = 0,  
arr[1][1] = 2,  
arr[1][2] = 1,  
...

arr[i][j] is 0 for the human player (X) or 1 for the computer player (O) or 2 for unfilled
Making a Move

• Moves could be represented inside the implementation by, for example, (1, 1) for putting a mark in the middle spot, row 1, column 1.
• The game knows whose turn it is, so a call to make a move doesn’t need to specify the player # making the move, but it can (ours will).
• The legal moves are specified in this game by what spots in the array are still unfilled.
Making a Move (cont’d)

- suppose “O” (computer) makes the move (2, 1) from the old game state

<table>
<thead>
<tr>
<th></th>
<th>Game state representation:</th>
<th></th>
<th>Game state representation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>X O X</td>
<td>arr[0][0] = 0,</td>
<td>X O X</td>
<td>arr[0][0] = 0,</td>
</tr>
<tr>
<td></td>
<td>arr[0][1] = 1,</td>
<td></td>
<td>arr[0][1] = 1,</td>
</tr>
<tr>
<td>X O</td>
<td>arr[0][2] = 0,</td>
<td></td>
<td>arr[0][2] = 0,</td>
</tr>
<tr>
<td></td>
<td>arr[1][0] = 0,</td>
<td>X O</td>
<td>arr[1][0] = 0,</td>
</tr>
<tr>
<td></td>
<td>arr[1][1] = 2,</td>
<td></td>
<td>arr[1][1] = 2,</td>
</tr>
<tr>
<td></td>
<td>arr[1][2] = 1,</td>
<td></td>
<td>arr[1][2] = 1,</td>
</tr>
<tr>
<td>X O</td>
<td>arr[2][1] = 2,</td>
<td></td>
<td>arr[2][1] = 1,</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Old game state</td>
<td>...</td>
<td>New game state</td>
<td>...</td>
</tr>
</tbody>
</table>
Minimax search for the best move

• We think of how X could move, and from there how O could move, down to the end positions, the leaves of the tree.
• We rate those leaves:
  3 for win by computer
  1 for a draw
  0 for win by human
Minimax search for the best move

- The basic technique is to use the leaf values to determine values higher in the game tree.
  - The values at the leaves are evident from the board (win by human, win by computer, draw)
  - The values are propagated up the tree. If a node corresponds to a board position where it is the computer’s move, the value is the maximum of the values of the children of that node, otherwise the minimum
  - This all happens by recursive search, with the values being delivered from the search below each point in the tree.
Weiss’s TicTacToe class

(online it’s TicTacToeSlow)

class TicTacToe {
  public static final int HUMAN = 0;
  public static final int COMPUTER = 1;
  public static final int EMPTY = 2;
  public static final int HUMAN_WIN = 0;
  public static final int DRAW = 1;
  public static final int UNCLEAR = 2;
  public static final int COMPUTER_WIN = 3;

  public TicTacToe() { clearBoard(); }

  // Find optimal move: the minimax search
  public Best chooseMove(int side) {
    /* Implementation in Figure 7.29/slide */
  }
}
public boolean playMove(int side, int row, int column) 
{ ... }

public void clearBoard() { ... }

public boolean boardIsFull() { ... }

public boolean isAWin( int side ) { ... }

private void place( int row, int column, int piece) 
{ board[ row ][ column ] = piece; }

private boolean squareIsEmpty( int row, int column) 
{ return board[ row ][ column ] == EMPTY; }

// the board--
private int [ ] [ ] board = new int[ 3 ][ 3 ];
chooseMove(side)  

Pseudocode

- See if the board is full (a leaf), and if so determine value: 0 for win by human, 1 for draw, 3 for computer win, and return it.
- If side == COMPUTER, set opponent opp = HUMAN and initialize value to 3 (highest) for running min. Else if side == HUMAN set opp = COMPUTER and init. value to 0 (lowest) for running max.
- Find the blank spots in the board, and for each:
  - Fill it in for this side (the trial move)
  - Call chooseMove opp to find best countermove by opponent, and update running max or min with it (looking for weakest best-countermove that the opponent could come up with)
  - Undo the fill-in to return board to old state to try next spot.
- Return the best move-from-here for this side to the caller.

4/8/2020
public Best chooseMove( int side )
{
    int opp;  // The other side
    Best reply;  // Opponent’s best reply (a move)
    int simpleEval; int bestRow = 0; int bestColumn = 0;
    int value;

    if( ( simpleEval = positionValue() ) != UNCLEAR )
        return new Best( simpleEval );  // leaf

    // set up opp, the opponent, for non-leaf case--
    if (side == COMPUTER)
    {
        opp = HUMAN; value = HUMAN_WIN;
    }
    else
    {
        opp = COMPUTER; value = COMPUTER_WIN;
    }
}
for( int row = 0; row < 3; row++ )
for( int column = 0; column < 3; column++ )
if( squareIsEmpty( row, column ) ) // possible move
{ place( row, column, side ); // do move
  reply = chooseMove( opp ); // recursive call
  place( row, column, EMPTY ); // undo move
  // Update if side gets better value
  if( side == COMPUTER && reply.val > value ||
      side == HUMAN && reply.val < value ) {
    value = reply.val;
    bestRow = row; bestColumn = column; }
}
return new Best( value, bestRow, bestColumn );
Serious recursion here

- We see a recursive call inside a double loop, so we should really worry.
- The recursion can only go down 9 levels, because there are at most 9 moves in any game.
- But still, this will generate 549,946 recursive calls to find the first move.
- At 1 us each, that’s .549 secs execution to find the first move. Not too bad. Actually even faster on my PC.
- There are ways to speed this up... we’ll look at dynamic programming for this in next class.

4/8/2020