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.
- Chess playing by an IBM supercomputer, Big Blue, beat the world chess champion in 1996 [Wikipedia].

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:
  ```
  int tiles[][] = {{1,2,3}, {4, 0, 5}, {7,8,6}};
  Which one is it?
  ```

Tic Tac Toe

```python
Empty = 2, Human = 0, Computer = 1 (Weiss setup)
int board[][] = {{(0,0), (2,2), (2,2)};
Which one is it?
See Wikipedia article 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.

Tic Tac Toe

- 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.

The tic-tac-toe Game Tree, in part

- 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. (In Nim, we also need to know who just played)
- That gives us a value for the leaf: 0 if X wins, 1 if a draw, 3 if O wins (the computer) (Weiss values)
- The important thing here is that X’s (human’s) win-value is lowest, O’s (computer’s) win-value is highest, draw is in-between. X seeks low values, O seeks high values in the search.
- See next slide...

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. (In Nim, we also need to know who just played)
- That gives us a value for the leaf: 0 if X wins, 1 if a draw, 3 if O wins (the computer) (Weiss values)
- The important thing here is that X’s (human’s) win-value is lowest, O’s (computer’s) win-value is highest, draw is in-between. X seeks low values, O seeks high values in the search.
- See next slide...

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:

- O to play, takes max
- X to play (human), takes min
- O to play (computer), takes max

Another example to try:

- What's the value of this board?

- We can tell it's Os turn to play because #X's = #Os

There are three descendents, try to go further.

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

Here are three descendents, try to go further.

Values: 0 if X wins, 3 if O wins, 1 if a draw
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.
- The computer (player O) does the first move, so #0's >= #1s
- When using a 3x3 array, the player # whose turn it is can be calculated by seeing whether the #0s > #1's, as it is after the very first move, when it’s X’s turn.
- 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

```
X O
---
X O
---
X O
```

arr[0][0] = 0,
arr[0][1] = 1,
arr[0][2] = 2,
arr[1][0] = 0,
arr[1][1] = 2,
arr[1][2] = 1,
arr[2][1] = 1,
...  
arr[i][j] is 0 for the human player (X) or 1 for the computer player (O) or 2 for unfilled

Since #Os = #Xs, it’s O’s turn to move next

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.

Result of 0 at root:
X has a sure win

- X can win for sure if it plays properly
- If X makes mistakes, it could fail to win

Making a Move

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

```
Old game state
X O
---
X O
---
X O
```

Game state representation:

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

```
New game state
X O
---
X O
---
O
```

Game state representation:

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

```
X O
---
X O
---
O
```

O’s move next.
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

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)

```java
public 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 */
    }

    // The board
    private int[][] board = new int[3][3];

    // the board--
    private int 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

The minimax search, part 1

```java
public Best chooseMove(int side) {
    int opp; // The other side
    int simpleEval; // score for this move
    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;
    // the board--
    private int int[] board = new int[3][3];
    } }
```

Weiss’s TicTacToe class, part 2 of 2

```java
public boolean playMove(int side, int row, int column) {
    if (boardIsFull()) return;
    // Board is still expandable
    board[row][column] = piece;
    // the board--
    private int int[] board = new int[3][3];
    }
```

```java
public boolean isAWin(int side) {
    // Implementation in Figure 7.29/slide */
}
```
The minimax search, part 2 of 2

returns best move from here

```java
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 use 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.

How can we speed this up?

• We can calculate that there are (less than) $3^9 =$ 19,683 different boards, so the 549,946 calls to chooseMove must visit each board many times.
• You might worry that you need to save which player is to move next along with the board, but in fact the board itself determines which player moves next, so the board represents the full state of the system (in tictactoe, but not all games).
• With less than 20K states, we should be able to use dynamic programming effectively for this problem.

Dynamic Programming

• For dynamic programming, we save results as we go along, and reuse them.
• Here, each board position has a value, so we need to save that value for each board, up to 19,683 of them.
• Idea: we need a Map of board $\rightarrow$ value

Position class

• Code: at Weiss’s website, for Chap. 10, in TicTacToe.java
• The Position constructor copies in the board array contents.
• Position’s equals checks for type (not perfectly*), then compares this.board and rhs.board element-by-element
• hashCode: Computes hashVal = 4*hashVal + board[i][j] across array elements, so 2 bits/value for array elements that can be 0, 1, or 2, i.e., binary 00, 01, 10. BTW, a perfect hash.
• So Position qualifies as a good HashMap element class.

* It’s OK as long as no code creates a subclass of Position, then tries to .equals its object using this code. We’re supposed to use getClass() for the typecheck.
Dynamic Programming

Add to chooseMove:
- At the start, after checking for leaf cases, see if current board has a value in the Map, and if so, return it.
- If not, do the usual recursive code to find the value
- When ready to return a value, also put (board, val) in the Map.
Cuts #recursive calls for first move down to 16168, much better!
And a little below the 19K upper limit we figured out earlier.
Note: Weiss's code from Chapter 10 also has some other optimizations not discussed here (alpha-beta pruning).

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.
- DP: Check if this board has a saved value and if so 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.
- DP: Save the newly computed board value
- Return the best move from here for this side to the caller

Project 3

- In project 3, you will try out dynamic programming in tic tac toe and in nim, another simple game.
- You can learn about nim at Wikipedia: here is an image from there.

Nim Game Tree

- Each player can take any number of stars/matches from one row
- Last player to take a star loses
- Micro-nim: 2 rows, 2 stars in one, one in other...

chooseMove for Nim

- In Project 3, you will provide Nim with chooseMove (recursive search without dynamic programming) so that the computer plays a good game.
- Note that making a trial move in chooseMove requires changing one of the heaps and the nextPlayer value.
- The computer, once optimal, should choose move: (row 0, take 3 stars) to start the game.
- Also, you will either add DP to this, or design a Game interface spanning TicTacToe and Nim

Values: 0 if X (human) wins, 3 if O wins, 1 if a draw (can’t happen here)
Dynamic Programming (DP) in general

- What is it about chooseMove that makes DP work so well for it?
- Once we have computed the value of a subtree, it is useful for many bigger calculations.
- Example of a problem that can’t use DP: find the median value in a tree of nodes with values.
  - The median of a subtree is pretty useless.
  - Need to sort the values, take the middle one.