CS310 – Advanced Data Structures and Algorithms

Class 15: Games: TicTacToe and Nim
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
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; }
}

side: HUMAN or COMPUTER
The minimax search, part 2 of 2
returns best move from here

```
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.
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 ➔ value
Dynamic Programming

- Idea: we need a Map of board → value
- HashMap sounds great for this.
- The board is held in an array, int[3][3]
- Arrays in Java are objects, but they don’t have hashCode or equals based on their content, just the Object methods, not good enough
- So we need to wrap the array in an object that we can provide with these crucial methods
- That’s what Weiss’s Position class does.

3/23/2020
Position class

- Code: at Weiss’s website, for Chap. 10, in TicTacToe.java
- The Position constructor copies in the board array contents.
- 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 type check.
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
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:
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.
DP for Change Making

- Change making can use solutions of smaller problems: for change for N, consider using one penny along with change for N-1, one nickel along with change for N-5, etc., then choose best
  - \#coins = change(N) = \min_k (change(N-k) + 1)
  - Where k = coin size (1, 5, 10,...)
  - We saw that a greedy algorithm worked with US coinage, but not with other possible coin sizes
  - So DP can help with those harder cases.
DP for Change Making

- \#coins = \text{change}(N) = \min_k (\text{change}(N-k) + 1)
  - Where k = coin size (1, 5, 10, 21, 25) for example
- We can turn this into recursive search with DP helping
- Or build up a table of coinsUsed by looking at change-for-1 cent, then change-for-2 cents, ...
- Get coinsUsed = \{0,1,2,3,4, 1,2,3, 4,5,1,2,3,4,5, ...\} up to 14 cents, the last being coinsUsed[14] = 5 (one dime, 4 pennies)
- Now work on 15 cents: first value 15 coins, try to do better...
Idea of DP change making

- So far coinsUsed = {0,1,2,3,4,1,2,3,4,5,1,2,3,4,5, ...} up to 14 cents
- Now work on 15 cents: first value is 15 coins, try to do better...
  - Try a penny as part, leaves 14 cents, look up coinsUsed[14] = 5, so new value is 5+1 = 6, better than 15, so the new min value.
  - Try a nickel as part, leaves 10 cents, look up coinsUsef[10] = 1 coin, so 2 in all, much better, a new min value
  - Try a dime as part, leaves 5, look up and find 1 coin, 2 in all, not better
  - Try a 21-cent piece, too big, try 25, too big, done: answer is 2 coins, fill in coinsUsed[15] = 2
- The general code is shown in earlier slide in class 12, and in next slide set.

3/24/2020
DP Slides from Nurit Haspel

• Prof. Nurit Haspel has been teaching cs310 from 2009-2019 at umb.
• She uses DP in her own research, try Googling “Nurit Haspel DP” and see a paper.
• She is an enthusiast about it, so let’s look at her slides on the subject.
• The slides cover three topics...
• The change-making problem, with same setup and code as our coverage.

• The LCS problem: longest common subsequence, important in her work on genomic sequencing. We’ll just get the general idea here, not expect to understand all the details.

• Weighted Interval Scheduling: We covered the unweighted Interval Scheduling problem in the last class, as another example of a greedy algorithm. But with weights on intervals, no greedy algorithm is known, so we need a search, aided by DP. This problem is covered in K&T Sec. 6.1.

• So we turn the class over to Nurit’s slides, in the next video.