CS310 – Advanced Data Structures and Algorithms

Intro PA3 on Games: TicTacToe and Nim

PA3: Games and DP: Background

CS310 Slides on tic tac toe:
• Intro in class of Apr. 1
• Repeat of intro plus rest of coverage in class of April 6

CS310 Slides on Dynamic Programming:
• DP for change-making with non-US-coinage: worked-out example in class of March 4, hw4 problem
• DP for binomial coefficients, same class
• Also slides on DP from Prof. Haspel (optional material)

PA3: Games and DP: Background

From pa3.html:
• Read Wikipedia Dynamic Programming intro, and Sec. 1.3 and 2.2. Also there: a fascinating story on how DP got its name.
• Also see Geeks for Geeks Dynamic Programming Basic Concepts.
• Weiss Chap. 7, esp. 7.6., and 10.2.

Pa3 Provided files

• TicTacToe.java: from Weiss, but without alpha-beta pruning, which is specific to 2-player games
• Best.java: a Best object is used to carry the best move from chooseMove to the client, so class Best needs to be known outside TicTacToe.java. Thus it has its own file.
• PlayTicTacToe.java: provided client code for TicTacToe.
• Nim.java: code for playing Nim, another 2-player game

Packages

• The provided files are all in the default package, so we put them in the src directory, and compile there.
• You will set up a cs310.games package as part of this assignment, which needs a src/cs310/games directory
• The source directories of the setup project are:
  • src for TicTacToe, Nim, PlayTicTacToe, Best
  • src/cs310/games for .java files in package games (originally empty), where all the development happens

Is TicTacToe.java encapsulated?

• This code certainly looks better encapsulated than JDK HashMap of pa2!
• It has one field, properly made private: private int[][] board = new int[3][3];
• It has 3 private methods, all helpers to chooseMove: place, squaresEmpty, and positionValue
• All the other methods are public, so should be the API for calls from clients
TicTacToe's API

public TicTacToe() (constructor)
public int[][] getBoard() // exports private data in bulk!
// Find optimal move
public Best chooseMove(int side, int depth)
// Play move, including checking legality
public boolean playMove(int side, int row, int column)
public void clearBoard()
public boolean isADraw()
public boolean isAWin(int side)

Fixing TicTacToe's API

- It's not a good idea to let the client have direct access to the class's private data.
- It's tantamount to having the board[][] public.
- With this method, the client could even change the data in the array, i.e., change what's on the board without going through the legitimate API playMove.
- You'll fix this encapsulation defect in pa3 step 1.

You could say “What’s the difference?”

- Printing a string with all the board information isn't much different than returning all the numbers in an array: it does externalize the contents of the private data.
- Except that with the provided array reference, the client could change the array inside the game. That's a serious violation of encapsulation.
- We could clone the array and return it, that would be better than what we had.
- Also note that parsing a string for internal values of an object is a code smell. It's better to renegotiate the public API if the client has need of more information. (However, sometimes you have to do it.)

Nim

- Wikipedia entry: "Nim is a mathematical game of strategy in which two players take turns removing (or "nimming") objects from distinct heaps or piles."
- For example: For one move, take any number of matches from one row.
- Our Nim setup has 3 rows, with 1, 3, 5 "stars" in each row.

Step 2: Nim.java

- This is provided, though you could try to write it yourself if you want.
- Its main has a little test code to try out.
- Then write PlayNim2.java, by starting from PlayTicTacToe.java and modifying it as little as possible.
- Since Nim.java has no chooseMove for the computer's move, you'll need to have the human supply both human and "computer" moves.

Step 3: chooseMove for Nim

- Provide Nim with chooseMove (recursive search without dynamic programming) so that the computer plays a good game.
- Use a new class BestMove.java for the best move (with i, j and val instead of row, column and val) so it can work for either game.
  - TicTacToe uses BestMove's i, j for row, col while Nim uses i, j for row, #stars.
- 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, i.e i=0,j=3.
**chooseMove**(*side*) Pseudocode: now we want to morph this to Nim: marked up phrases specific to TicTacToe

- 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

---

**Work on these spots**

- TTT “board is full”, a leaf position, check for 3 in a row
- Nim equivalent: ran out of stars, all heaps are empty: just see whose turn it is next, that’s the winner
- TTT “find the blank spots in the board” for the trial moves
- Nim equivalent: a trial move could be taking one star from row 0, or two stars, … up to how many there are in that heap. Or it could be taking one or more stars from row 1. Or … from row 2

---

**Undoing the move**

- You need to appreciate the power of local variables* to hold their value across calls to methods, even the same method.
- Look at TicTacToe: local variables row, column, side
  - place(row, column, side);
  - reply = chooseMove(opp, depth+1);
  - place(row, column, EMPTY);
- After the call, row and column have the same values they did before the call, even though this call will execute this same code. Local variables have “per-call” storage on the “execution stack”. You can depend on it.
- *Note that instance variables may change across a call

---

**Debugging chooseMove**

- Recursive methods are tricky to debug.
- Adding one println can cause hundreds of lines of confusing output in the full problem.
- So switch over to Micro-nim: 2 rows, 2 stars in one, one in other…
- Just put `SZ_ROW0 = 0, SZ_ROW1 = 2` and leave `SZ_ROW2 = 1`
- Then the game tree has only 11 nodes…
- You can “afford” a couple of println’s in chooseMove to see if it’s doing what you want.

---

**Values:** 0 if X (human) wins, 3 if O wins, 1 if a draw (can’t happen here)
Understanding argument “depth” in chooseMove

- API method: Best chooseMove(int side, int depth)
  - Side is COMPUTER or HUMAN
  - Depth is the recursion depth, there to help with debugging
- Original call from PlayTicTacToe (or PlayNim): depth = 0
  ```java
  Best compMove = g.chooseMove(TicTacToe.COMPUTER, 0); // depth 0 call
  ```
- Call from depth 0 execution of chooseMove: depth = 1
  ```java
  reply = chooseMove(opp, depth+1); // Since depth == 0, this is chooseMove(opp, 1);
  ```

Using argument “depth” in chooseMove

- We see that depth tracks how deep the recursion has gotten
- It’s very useful in debugging. You can look at the near-leaf executions by doing, for example (in TicTacToe):
  ```java
  if (depth > 6) {
    System.out.println( "board at depth " + depth + ": " + this);
  }
  ```
- Here “this” is TicTacToe, and its toString is called (which you will have working by the time you are debugging chooseMove)
- Or look at depths 0 and 1 with (depth < 2), etc.

Debugging

- First, simplify to micro-Nim with 2 stars and 1 star
- Have the marked up game tree handy
- Consider the first descent down the tree and make sure your code is handling it correctly

Tracking this

- Put printlns to show the result of the leaf evaluation and the result after the double loop, and the level
- Output for the arrows shown: just one report at that leaf, one report on way back:
  ```java
  A
  B: *
  C: *
  A
  Level 0
  Level 1
  Level 2
  Level 3
  ```
Final part: choice of two tasks

• 4a: Describe the commonality of the two game classes by devising and using a Game interface.
  • You will have to rename some methods to get this to work
  • or 4b: Add DP to Nim
    • Add a Map to hold previously computed values of game positions, as we see in TTT, and use them to short-cut the recursive search.

Part 4a: Game interface

• Write an interface Game.java that covers the actions needed for the clients for both games PlayTicTacToe4 and PlayNim4.
  • Many of these actions have different names in the two cases, so you need to change some method names, ending up with TicTacToe5 and Nim5.
  • For example, clearBoard() in TicTacToe has the same generic action as init() in Nim, namely, to bring the game to its initial state. Clearly init() is the more generic name, so rename clearBoard in TicTacToe to init and put init in the interface.

Part 4a, cont.

• Once you have aligned the APIs of the two game classes, and made them both “implement Game”, morph PlayTicTacToe or PlayNim into PlayGame, a single client that can handle either game.
  • Of course, we need to use the concrete classes to do “new TicTacToe4()” or “new Nim4()” in main(), and put the result in a Game variable.
  • To find out which game the user wants to play:
    • Usage: java cs310.games.PlayGame (no args): play TTT
    • java cs310.games.PlayGame Nim: play Nim

or 4b: Add DP to Nim

• You need a map like “store” in TTT to hold values for board positions.
  • Note that store is a HashMap<Position, Integer>, where Position holds the game state.
  • You need to devise a Position inner class for Nim following the general pattern of TTT’s Position.
  • Then populate and use a big HashMap<Position, Integer> as done in TTT.

We could go on...

• Further development ideas:
  • Making a Move type that can bridge the two cases better than our kludge of using i and j in BestMove. BTW, “kludge” has a Wikipedia page.
  • Set up a Move iterator that delivers all the Moves from a certain game state. Eventually could do chooseMove outside the Game class.
  • Other games, like 4x4 tic tac toe, described in Wikipedia’s Tic-tac-toe page.