CS310 - Advanced Data Structures and Algorithms

Fall 2014 – Class 16

April 8, 2014

Backtracking Algorithm

- The stub is in games/src.
- Just do the recursion in getValue (for next pa) – the top-level code compares various move-value combos:
    - int getValue(Game g)
    - compute the value of g recursively
    - return the value

Pseudocode for Minimax Search for Best Move

- A backtracking algorithm: Weiss uses low values for positions good for humans, high for computer.
- It's easier for us to use a high value (1) for game states good for player ONE, low value (-1) for player TWO, since the Game doesnt know human from computer players (it has no need to.) A draw is 0 in this system.
- Please use this convention for pa4/5.

Running a game:PlayOneGame

The minimax search we need for getValue(Game g) is as follows, working from the code in Weiss, pg. 337:

```
If the base game state is a won game or a lost game or a draw,
return the appropriate value immediately
Determine the current player# from the base game state
Initialize a running min/max, to worst possible outcome for player#
Loop through moves from this base game position.
For a certain possible move:
    Make the move on a copy of the game state
    Get the value for this new game position by calling getValue
    Use the value to update the running max or min, depending on player#
Return the min/max value
```

Minimax Search

- The side parameter in Weiss' code is not really needed with our Game ADT because the game state knows whose turn it is and that can determine whether to do the max or min.
- We do need a Game parameter, however, because this code is outside the Game class.

Implementing findBest

- There are two ways to implement findbest.
- If the helper returns only the value, not the move for it, the top-level findbest code can do its own min/max to figure out the best one.
- The advantage with this approach is a simpler getValue, since it simply finds a value for a certain Game state.
- The disadvantage of this approach is the distribution of the min/max code – it's nicer to have it all in one method.
- The simpler getValue is easier to upgrade to dynamic programming.

Pseudocode for findBest Using DP

Now we add a map of values for various game positions, so we avoid re-computations:

```
int getValue(Game g)
    try to look up value of g in a map
    if you succeed return the value
    else
        compute the value of g recursively as in Backtrack
    save the computed value in the map
    return the value
```

Dynamic Programming findBest

- In pa5, you'll use dynamic programming for findbest, and measure its effectiveness with the timing results.
- In that case the map goes from Game state to value for it.
- The Game state includes the board position and whose turn it is.
- In Tictactoe you can tell whose turn it is from the board position, but in other games such as Sticks or Nim, you can't.

Dynamic Programming findBest

- Be careful about object sharing when working with the Map of Game to Integer here.
- We need the Game objects to keep the same value once inside the Map.
- That's required to keep the Map lookups working properly, with HashMap or TreeMap.
- So make copies of Game objects as needed.

Memory Usage in Dynamic Programming Map

- How much memory is used by the dynamic programming map?
- There are 3^9 game states in tictactoe, since each of 9 spots can have an X, an O, or be blank. $3^9 = 19,683$.
- Here game state is a 3x3 integer array that has size 9*4 $= 36$, so hash entry size $= 4 + 4 + 36 = 44$ bytes.
- So if the map held all game states it would use about 50*20,000 $=$ 1M. Not much memory for current systems.

Memory Usage in Dynamic Programming Map

- Of course chess would be much worse.
- Using Weiss' estimate for the first move in TicTacToe, the original backtracking code makes 549,946 recursive calls (pg. 336).
- Obviously many game states are visited many times.

Testing Backtracking Using Easy

- Backtracking is a challenge to debug.
- Remember it's crucial to believe in recursion to get it to work.
- Trust getValue to return the actual value of the various positions.
- If it doesn't in testing, dive down a level and see what's happening there.
- If the human moves first and takes the 1, we leave the computer with a fairly simple but not trivial computation of its best move.

Testing Backtracking using Easy

- Here the base game position is G0 (by pa4 numbering) = [1] [], and player 2 is to move.
- The possible moves from here: 0, 2, 3
- findbest should loop through these moves and find the best move for player 2, that is, the one of lowest value.

1. move = 0 from g0 (give-up move.) findbest calls getValue to evaluate this, which should return 1, loss by player2
2. move = 2 from g0 to g1 = [1][2] findbest calls getValue and gets back 0, since this is a draw no matter what (the recursion in getValue should return this value.)
3. move = 3 from g0 to g1 = [1][3] findbest calls getValue and gets back 1, since now there's a win for player1 on the next move.

Testing Backtracking using Easy

- findbest compares the values 1, 0 and 1 for player 2, and chooses the minimum one, with move $= 2$.
- Thus if you print out the value returned from getValue in your findBest, it should show 1, 0 and 1 in this case. If it doesn't, look at how it got the wrong value.
- If necessary, add a "level" variable to getValue so you can print out just the values it is working with at one level.

Weiss Code for Dynamic Programming findbest (Optional)

- Weiss has code on pp. 436-437 using dynamic programming via a map called a transposition table.
- However this code also does alpha-beta pruning, so there's more going on as well.
- We're not covering alpha-beta pruning, but if you want to get further into game programming, it's a good thing to learn.

Weiss Code for Dynamic Programming findbest (Optional)

- It has some application beyond games, but does require a minimax situation.
- Weiss' code doesn't copy/clone a game state because it uses a special Position object (def. pg. 434), that actually has the same kind state (3x3 array of int) as his TicTacToe class (pg. 335.)
- Thus he new's a Position object and initializes it from the current TicTacToe object.
- The effect is a copy action on the 3x3 array that defines game state.

Tricks With Recursion Level in Dynamic Programming (Optional)

- In Weiss' code, the recursion depth is tracked, and only depth ≤ 5 game states are saved in the transposition table.
- The max recursion level is 9 at the start of the game, 8 after the first move, then 7, etc., so only a few levels are being eliminated here.
- The idea is that these are less valuable values, since they are easier to compute.
- You could try the same idea, at least for Tictactoe.

Tricks With Recursion Level in Dynamic Programming (Optional)

- Also, dynamic programming doesn't actually require us to keep all the higher levels either.
- You could try saving only every third level, say. The idea is that holding all these entries in the map takes a significant amount of memory, so it might start slowing things down. You could find the size of the map to estimate its memory use. Each entry needs a key (Game) and a value (Integer) and a flag, maybe 100 bytes.
- The depth ≤ 5 restriction is tuned to Tictactoe.
- Another game might have deeper but skinnier recursion and this limit of 5 would not be right for it.

Wrap up on Games

- We have covered backtracking and dynamic programming applied to backtracking.
- The backtracking algorithm uses a minimax strategy, that is, it is finding the best move based for the current player (a max, say) on best play by the opponent, which in turn involves min value choices, etc.
- The idea of finding the best choice based on pessimistic evaluation of each option can be applied in other realms such as investment. If you have two choices for retirement investment, one of which guarantees 3% return and the other 2-5% return, which should you take?

Wrap up on Games

- Minimax says the 3% one, which maximizes the minimum outcome.
- Similarly, speeding to work to get there on time yields to minimax, because getting a ticket delays you a lot.
- There is more we could do with speeding up the backtracking. Sec. 10.2.1 discusses alpha-beta pruning, which is a way to recognize work that doesn't need to be done to find the minimax.