CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 13

April 1, 2014

Fun and Games

- Everyone knows tic-tac-toe and other games – now we can program it to be played well by a computer!
- The games we will work with have two players, a set of possible moves, and a notion of game state, such as position on a board (or other device that lets both parties see what's what), and knowledge of whose turn it is to make the next move.
- Some background material in section 7.7 and chapter 10 (I will not follow the book closely, though).

Fun and Games

- Moves cause changes in board positions, and changes in game state.
- 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.
- Mostly – a tree structure for all possible states.

The Tic-Tac-Toe Game Tree

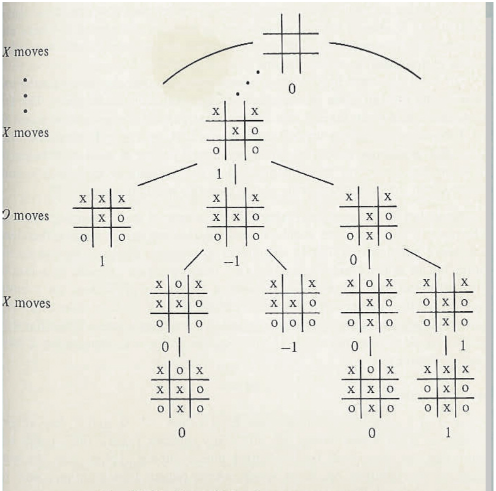


Fig. 10.16. Part of the tic-tac-toe game tree.

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:

for X:	for O:
1 for a win,	-1 for a win
0 for a draw,	0 for a draw

Minimax Search for the Best Move

- The basic technique is to use the leaf values to determine corresponding values higher in the game tree.
- First, we assign values to the leaves (-1,0,or 1 depending on whether the board position corresponds to a loss, draw or win for player X).
- Second, the values are propagated up the tree. If a node corresponds to a board position where it is player X's move, the value is the maximum of the values of the children of that node

The Technique

- Look at the board in the middle, rated -1. Its children are rated 0 and -1.
- Consider being “O” at that board position. What the 0 child is saying is that if you put an “O” in the top row it leads to a draw (i.e., you are assured of a draw if you play properly.)
- The -1 child is saying if you put your “O” in the bottom row you can win. Thus your best move is the latter, and that gives you a -1 value here.
- Any reasonable O-player would do that move, so we might as well settle on a -1 value for that position.

To Win

- Similarly consider the X-to-move position rated 1 just above that one.
- It has 1, -1 and 0 children, meaning win, loss and draw for X, of which any X-player should choose the 1.
- We see that in an X-to-move position we choose the max child value, and for any O-to-move position we take the minimum child value, for the current position.
- This does assume the opponent is a decent player!

Games ADT

- By encapsulating the games in a Game ADT, we'll be able to write general game-playing strategies that can then be used with each of the games.
- A Game ADT is different from the data-holding ADTs we've been studying, showing how general the ADT technique is.
- Anything you can boil down to a set of operations can be expressed as an ADT

The Game ADT

- The game package has about 32 classes.
- Only a few of them are needed to describe a particular game: Game, Move, MoveIterator, and PlayerNumber.
- This group is called the core Game ADT.
- These are the ones plus a few others you need to master to do the future homework.

PlayerNumbers Class

- PlayerNumbers are enum values.
- `public enum PlayerNumber {ONE, TWO, GAME_OVER, GAME_NOT_OVER, DRAW};`
- We are defining 5 immutable enum objects PlayerNumber.ONE, ...
- The actual fancier setup allows us to specify what an enum's toString() will be, overriding the default of its name, and its associated "realPlayer" status.
- With this setup, any non-null PlayerNumber is an actual valid value.
- And these are smart constants: you can ask them if this value represents a real player or not.

Game ADT

Game:

- An abstract class describing what needs to be implemented for a game, plus some common code.
- Its most important operations are make, for making a move, and getMoves(), for getting a MoveIterator.
- The "game state" is the current value of the Game object. A move changes the game state.
- Tic-tac-toe, for example, is a class extending Game. Tic-tac-toe is a concrete class, i.e., not abstract, and can be instantiated.

Game ADT

Move:

- An interface describing operations needed for handling moves
- they are just the basic Object operations of clone, equals and hashCode, plus copy().
- The implementation of a game is expected to generate Move objects (really objects of some class implementing Move) to provide to the client via a **MoveIterator**.
- A MoveIterator provides all the legal moves (for the current player) at some particular game state, including resigning if that is allowed.

Tic-tac-toe Moves

- Game states are represented inside the ADT by a 3x3 array of int, where the arr[i][j] is 1 or 2 for ONE or TWO or 0 for unfilled.
- See Weiss, Fig 10.13 on pg. 434 to see that this representation is used there, except that different numbers stand for the three cases.
- When using a 3x3 array, the player # whose turn it is can be calculated by seeing whether the #1's > #2'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

X	O	X
X		O

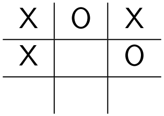
arr[0][0] = 1,
arr[0][1] = 2,
arr[0][2] = 1,
arr[1][0] = 1,
arr[1][1] = 0,
arr[1][2] = 2,
...

Making a Move

- Moves (class, say, TicTacToeMove) could be represented inside the ADT 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 move doesn't need to specify the player # making the move.
- The legal moves are specified in this game by what spots in the array are still 0 (unfilled)

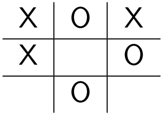
Making a Move

suppose "O" makes the move (2, 1) from the old game state:



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

Old game state



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

New game state

Core Game API

- Reference: gamesAPI.html
- void make(Move m), makeObservable(Move m) – apply a move to a game state, causing it to change to a new game state.
- makeObservable not only does the basic “make” action, but also makes the move user visible.
- MoveIterator getMoves(); – supply a MoveIterator for this current game position (all the moves from here)
- void init(), initObservable(); – set up the game state for the beginning of a game (plus poke display code)

Core Game API

- Game status:
    - boolean isGameOver()
    - PlayerNumber winner() – PlayerNumber for winner, or special values to indicate not-yet-done or draw.
    - PlayerNumber whoseTurn()
- plus getName, getAuthor, copy, hashCode, equals
- Move: just copy, equals, hashCode. Note that Moves are immutable: once created, they can't be changed.
- MoveIterator: next, hasNext. Not itself immutable, but provides immutable objects.

Working with a Move Iterator

Loop through all the moves possible from the current game state of TicTacToe game g:

```
Iterator<Move> itr = g.getMoves();
while (itr.hasNext()) {
    Move m = itr.next();
    ...
}
```

- As mentioned above, moves arent very interesting in themselves. However, they drive changes in the game state:
- g.make(m); // make move m, changing game state in g

Note on Weiss Coverage on Tic-Tac-Toe

- In Weiss, tic-tac-toe is implemented with special-case code, in pp. 334-337 and 427-435.
- There are many parallels between this special case and our general case.
- The class TicTacToe contains a 3x3 array of ints to hold the "board position", i.e., the main part of the game state.
- playMove is like Game's make, and changes the game state (board array) due to the move.

Note on Weiss Coverage on Tic-Tac-Toe

- A move itself is specified by a row and column for the new mark.
- There is no move iterator, just double loops over rows and columns looking for empty spots, a similar action.
- Weiss's TicTacToe class has chooseMove() (find the optimal move) and positionValue(), which have no counterparts in our Game's API.
- Instead, finding the best move (through various position evaluations) is done outside Game in our setup.
- Our Game class corresponds to the rest of the TicTacToe class, responsible for holding a game state and handling moves to a new game state.