CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 14

April 3, 2014

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.

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

Working with a Move Iterator

- Note that there is no way to undo a move, so if we want to explore possible moves, we should make a copy of g and do "make" on the copy, and see what happens:
- Game g1 = g.copy(); // g1 as "scratch copy" of g
- g1.make(m); // a move changes g1s state
- PlayerNumber result = g1.winner(); // see if game won after this move, and by whom

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.

How Is It Done, Then?

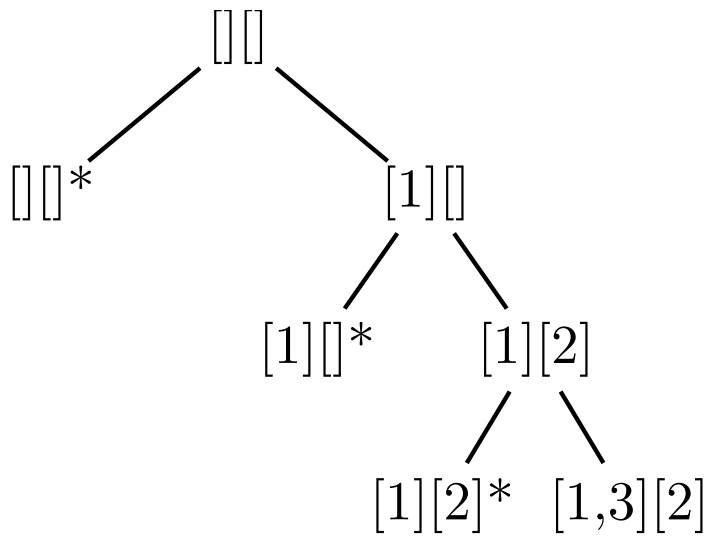
- "Easy" is a very stupid game for demonstration purposes.
- Two players, 3 board positions. Player #1 wants the first position, player #2 wants the second.
- If one player got what they want and the other didn't –that player won.
- Otherwise the game is drawn.

Game Tree for PA4

- pa4 is an intro to the game package. Instead of backtracking to find the best move, which is tricky, the search only looks at the states in the tree and prints them out.
- For pa4, here are the first few lines of expected output for DisplayTree1:
  - [][] (1 next) (level 0)
  - [1][] (2 next) (level 1)
  - [1][2] (1 next) (level 2)
  - [1, 3][2] (2 next) (level 3)

Game Tree for PA4

This part of the game tree looks like this, with stars for states after give-up moves:



Note: game trees are like recursion trees, traversed by execution, not built into data structures.

Running a game:PlayOneGame

- Games and Players come together in PlayOneGame objects.
- Each PlayOneGame object HASA Game and 2 Players, one of which ISA HumanPlayer and the other ISA a ComputerPlayer in the normal way we use it (but it can be 2 ComputerPlayers or 2 HumanPlayers)
- We see that the Player that goes with Game.FIRST_PLAYER is Player "one" of PlayOneGame, the player that takes the first turn, and the other is "two".
- The Player "one" can be human or computer, depending on our answer to the question about going first.

The PlayOneGame Structure

- Thus the correspondence between PlayerNumbers and Players is held in the PlayOneGame object.
- It was decided on in PlayGame.
- Naturally, we need code to run a game.
- All this needs is a loop of calls to findbest of Player and make of Game.
- This is in PlayOneGame.java, in the games package.

The PlayOneGame Structure

- This shows the power of abstraction: PlayOneGame can play the game without knowing much about it: what it needs is provided by the Game API, and similarly with the Players.
- Its code has a loop which calls findbest of Player to get the next move and then makes the move via the Game API.

The PlayOneGame Structure

- In practice it's not quite that simple.
- In PlayOneGame.java, we see calls to makeObservable rather than to make, so the basic play-game loop has calls to findbest and makeObservable.
- What is this all about?
- In Game.java, we can read makeObservable and see that it calls make and then notifyObservers().
- Who are these observers we're notifying? They are the timing module and the UI.
- Both of these modules are interested in the event of a move happening.

The Observer Pattern

- The Observer interface and the Observable class are in the standard JDK.
- Together they give us the "observer pattern".
- In this pattern, the Observable object sends out "notifications" when it changes to registered observer objects, by calling notifyObservers(). The observer objects then can check further by calling various methods of the observable.
- The Observable doesn't know anything about the observers, so you can imagine that the observers are (roughly) spying on the observable, interested in when it changes.

The Observer Pattern

- Here the Game ISA Observable. The game state changes occur because of moves, so each move (made via makeObservable) causes a notification of change. Objects interested in tracking the moves of a Game can register themselves as Observers.
- The timing package wants to separately time each player, so wants to know about moves.
- The UI wants to update or redraw the board after each move, so is also interested in moves. The Game itself does not need to know about these specific needs once it has become an Observable – any interested object can sign up for notifications.

The Observer Pattern

- When the timing package is notified about a move, it stops timing one player and switches over to timing the other player.
- When the UI is notified, it repaints the representation of the game for the user, because the move has changed things.
- PlayOneGame loops through the moves of a game and, by making moves in the Game, indirectly notifies appropriate parties about the moves.
- Again note that it knows the Game only as Game, players only as Players, yet still can play the game.

API Details

- These are JDK classes, so you can see all the details in that Javadoc.
- Another reference: Head First Design Patterns, by Eric Freeman and Elizabeth Freeman, O'Reilly, 2004, ISBN 0-596-00712-4, where this is covered in Chap. 2. (MVC is covered there in Chap. 12)

1. Setup: Game extends Observable, itself a concrete class, bringing to Game the implementation of the notification-registration system and the notification-sending system: The calls used in game classes are addObserver(Observer o), setChanged(), notifyObservers(), notifyObservers(Object arg);

API Details

2. Setup: the Observers are the timing package and the UI. They each need a class that implements Observer in order to receive notifications (via calls to method "update" in the Observer). The UI base class GameView itself implements Observer, and in its own initialization, calls g.addObserver(this), adding itself to the registered Observers of Game. The top-level timing class GameTimer also implements Observer, and also adds itself by g.addObserver(this). Each of these Observer classes have a method called update that is ready to handle a notification.

API Details

3. Operation: when a move is made via makeObservable in Game, Game code there calls setChanged() and notifyObservers(m), passing the move as the argument.
4. Operation: Each Observer gets a call to its update method, with the move object as an argument. It does the action desired: In GameTimer, it checks about switching which player to charge time to. In the UI, it shows the user the new board.
5. Operation, case 0: when the game is started, Game.initObservable calls notifyObservers(), no arg.
6. Operation, case 0: Each Observer gets a call to its update method, with move=null, and does any appropriate action for the start of the game.

Game Demo (Handout)

- TestGame prints out all the moves from the starting position of the game, and the game states after these exploratory moves.
- The legal moves are 0, 1, 2, and 3, resigning being legal in this game.
- After an initial move of 1, the game state is printed as follows by toString: "[1] [] (2 next)".
- This means Player1 has 1, Player 2 has nothing, and Player2 is next to play.
- In the next pass, using a move of 2, the game state is printed as follows: [2] [](2 next)

Game Demo

- Note how the move of 1 has been wiped out in effect, before move 2 is made.
- We are again considering an initial move from the starting position of the game.
- The move of 1 was done with a scratch copy of the game state, leaving the base game state alone to provide another good starting copy for the second move.

TestGame Software Structure

- TestGame uses Easy, which is a concrete class implementing Game.
- Easy has a private nested class EasyMove that implements Move and private inner class EasyMoveIterator that implements MoveIterator.

TestGame Structure

- Game game = new Easy(), game.getMoves() returns a Move Iterator object that is specifically an EasyMoveIterator, say mItr, and that iterator provides Move objects by mItr.next() that are specifically EasyMove objects.
- Just like container iterators, we don't need to know the specific types, since the API is given by the top-level interfaces or abstract classes.

TestGame Structure

- Inside Easy, there are two private TreeSets “moves1” and “moves2” to keep track of which spots each player now has.
- The toString method of Easy just uses TreeSet’s toString to print for example “[2, 3]” to represent one player’s holdings:
- String s = moves1 + “ ” + moves2 + “(”;
- s += isGameOver()? “done”: “” + nextPlayer + next)
- We can change TestGame to run another game just by changing the new statement, for example to “new Tictactoe()” or “new Nim()”

Running TestGame

- The classpath can also be specified as an argument to the javac command:
- javac -classpath c:\cs310\games\classes TestGame.java (Windows)
- classpath is also available for UNIX.
- Also a CLASSPATH env var works in both places.

Creating Games (and other objects) by String Name

- So far, we have been creating games by specific classes:
- Game game1 = new Easy();
- Game game2 = new Tictactoe();
- But Java has the power to instantiate a class given its string name:
- Game g = (Game (Class.forName(className).newInstance());

Creating Games

- or in steps:
- Class class = Class.forName(className); // Class object for game class
- Game g = (Game)class.newInstance(); // Easy or Tictactoe or object
- A method with this code and returning the Game object can be called a Game Factory method. Of course it can be even more abstract.
- This works for any className:
- Object o = Class.forName(className).newInstance();

Creating Games

- This code can be found in AbstractFactory.createObject(classname) in the games package.
- GameFactory.create(gamename)checks gamename against its known names, prepends "edu.umb.cs.game." to a good name, and then calls createObject(gamename), a call to its superclass, AbstractFactory.
- Thus we see that both of the following create new Game objects of subclass Easy.
- Game g1 = GameFactory.create("Easy");
- Game g2 = new Easy();

Player Object

- A full-fledged "player" object decides on moves.
- In our case, we have human players who think up moves, and computer players who search mechanically for moves.
- They are both players of the game, so this situation suggests inheritance like the following.
- Further, we want to be able to try different computer algorithms for findbest, so we keep ComputerPlayer generic and plug in various subclasses.
- Perhaps BackTrack should be called BackTrackingComputerPlayer, but that's a really long name.

Player Object

Player: findbest is abstract

HumanPlayer:

- uses brain for findbest,
- accessed via i/o with user (in gamesui package)

ComputerPlayer:

- uses search, findbest is abstract.
- Backtrack: specific algorithm for findbest uses minimax search for findbest action (you implement this)

Player Implementation

- Here the concrete classes are HumanPlayer and Backtrack, the backtracking ComputerPlayer.
- So Backtrack ISA ComputerPlayer ISA Player, and also HumanPlayer ISA Player.
- You are going to fill out Backtrack.java to make it do the minimax search.
- Player.java, HumanPlayer.java and ComputerPlayer.java are in the games package.
- Backtrack is not, since it is not fully implemented.

Player Implementation

- Note that the Game class only knows PlayerNumbers, not Players.
- Players and Games come together at game-playing time, and then Players are known by their PlayerNumbers in a particular Game instance.
- Recall that PlayerNumbers are really simple, just 5 particular values.
- An analogy for PlayerNumbers are tickets at the deli counter. You, a complicated object, are reduced to #57 while you wait for your turn and then interact with an equally complicated agent on the other side of the counter.

Running the Whole Thing

- PlayGame, factories for game and player objects:
- We need a main program to create objects and call PlayOneGame.
- PlayGame.java has this main program, and is in package cs310 as the app of interest for our project.
- You don't need to change it. The creation of Game objects uses the capability of Java to take a string name of a class and load that class for execution, covered in the last lecture.
- This way, we have extendibility in games, that is, we can add a game to the system after all the game-playing software is written.

Summary of Important Classes

- PlayGame – static method playGame creates view, game, players, and PlayOneGame object from them, calls PlayOneGame.go()
- AbstractPlayOneGame, PlayOneGame – consists of two Players and one Game. Loops over moves of one game: figures out which player has the turn and calls that player's version of findbest to get a move, use makeObservable to make the move.

Summary of Important Classes

- Player, ComputerPlayer, Backtrack or Dynamiccomputer player, you write Backtrack and Dynamic for pa5
- HumanPlayer – human player who inputs moves from keyboard. In package gamesui.
- Game, subclasses Easy or Tictactoe or Sticks – game class; Move; Move Iterator; PlayerNumber

Other Classes in Use

- Factories: AbstractFactory, subclasses GameFactory, PlayerFactory
- UI: Terminal, GameView, subclass ConsoleView,
- Exceptions: GameException, NoSuchMoveException, IllegalMoveException

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.