Project 2: Dynamic programming

COT 4400, Summer 2016

Due July 17, 2016

1 Overview

For this project, you will develop an unbeatable AI for the Three-Pile Subtraction game. Designing and implementing this solution will require you to model the problem using dynamic programming, then understand and implement your model.

You are only allowed to consult the class slides, the textbook, the TAs, and the professor. In particular, you are not allowed to use the Internet. This is a group project. The only people you can work with on this project are your group members. This policy is strictly enforced.

In addition to the group submission, you will also evaluate your teammates' cooperation and contribution. These evaluations will form a major part of your grade on this project, so be sure that you respond to messages promptly, communicate effectively, and contribute substantially to your group's solution. Details for your team evaluations are in Section 10.2. You will submit the peer evaluations to another assignment on Canvas, labelled "Project 2 (individual)."

A word of warning: this project is team-based, but it is quite extensive and a nontrivial task. You are highly encouraged to start working on (and start asking questions about) this project early; teams who wait to start until the week before the due date may find themselves unable to complete it in time.

2 Problem Description

The rules to the Three-Pile Subtraction game are as follows:

1. The game begins with some number of stones distributed into three different piles.
2. Two players take turns removing stones from one of the piles, where a move consist of removing the number of stones in one pile from another pile (i.e., subtracting one pile from another).
3. A player may not remove more stones from a pile than the pile contains; i.e., you must always subtract smaller numbers from larger numbers.
4. A player must remove at least one stone on their turn; a pile of zero stones is effectively out of the game.
5. Play continues until no valid moves remain, and the last player to make a move wins. This will happen when there are two empty piles of stones.

For example, when the three piles contain 1, 2, and 3 stones, the valid moves are: removing 1 stone from the pile of 2, removing 1 stone from the pile of 3, and removing 2 stones from the pile of 3.

Like many variants of the game of Nim, the Three-Pile Subtraction game exhibits an optimal strategy, where games can be classified into two categories: *winning games*, which will always be won by the player who makes the next move (assuming they know the optimal strategy), and *losing games*, which will always be lost by the player who makes the next move (assuming their opponent knows the optimal strategy). In this project, you will uncover the optimal strategy for the Three-Pile Subtraction game and create an unbeatable AI player. In addition, you will write a report describing the algorithmic design decisions you made in implementing your AI.

3 Anatomy of an unstoppable AI

Your AI program will have three main functions for its core logic. One function, `getMoves`, will accept the current state of the game and return the list (and count) of potential moves that a player can make. The second function, `isWinner`, will accept a given game state and classify the game as a *winning game* or *losing game*, using the `getMoves` function. The third function, `bestMove`, will take a game state and compute the optimal move using `getMoves` and `isWinner`. The requirements for these functions are detailed in Sections 4, 5, and 6, respectively. The AI program will begin the game by asking the player how many stones to put in the three piles, deciding whether it wants to go first, and then defeating the player in the game. The input format for the AI program is detailed in Section 7.

As you begin working on the three functions, there are four important questions you need to answer. You will want to discuss these questions as a team, as they impact more than one function.

1. How will you represent the game state (i.e., the three piles of stones) in memory?
2. How can you classify a game as a winning or losing game based on the classification of the possible moves from this state?
3. Why is dynamic programming a good algorithmic strategy for this problem?
4. What data structure should you use to store the answers to the various subproblems?

There are multiple valid ways to answer the first question; e.g., an object, a struct, an array, etc. However, whichever representation you choose, you are *strongly encouraged* to sort the piles of stones, as the order of the piles doesn't make a difference in the moves available or the optimal strategy (i.e., the game with piles of 1, 2, and 3 stones is "the same" as the game with piles of 2, 3, and 1 stones). You may wish to write a helper function to sort the piles of stones.

The second question is one of the most conceptually difficult parts of the assignment. The main observation to make is that every move you make from one game state creates another game state that is then passed off to the other player. Since each move results in a game with fewer total stones, we may classify these games as *winning* or *losing* recursively. We can then classify the current state according to the classification of the possible moves. Some questions you may wish to ask when making the classification appear below:

1. What if one of my moves leads to a winning game?
2. What if one of my moves leads to a losing game?

3. What if all of my moves lead to winning games?

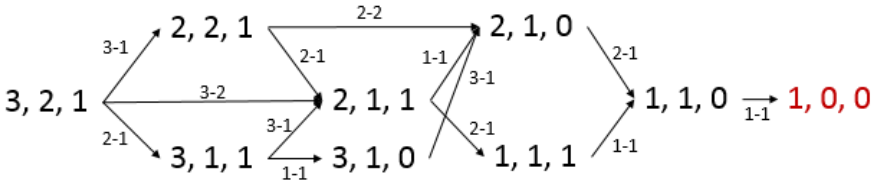
4. What if all of my moves lead to losing games?

Note that a game with no legal moves is a losing game by definition.

The third question is self-explanatory. For the fourth question, the data structure you use for this problem should be clear, based on what we've discussed in class. The data type for the subproblem solutions, though, should be either an enumerated type (`true`, `false`, or `unknown`) or a pair of Boolean values (`isKnown` and `isWinner`).

3.1 Example game

The diagram below includes all of the possible moves resulting from the game that starts with three stones in one pile, two stones in another, and one stone in the third pile:



This game may take 4–5 turns to complete, though if Player 1 is clever in choosing their first move, they can guarantee that the game will take 5 moves, ensuring a win. In other words, this game is a *winning game*.

4 Study your options

The `getMoves` function should accept the game state (i.e., how many stones are in each pile), and return the number of valid moves from that state, as well as the list of game states that result from those moves.

You will then need to design a pseudocode algorithm for `getMoves`. This pseudocode will go in the report that you will submit for this project (described in more detail in Section 8). In your pseudocode, you may adopt any reasonable convention for accessing the size of the stone piles from your game state (input) variable; e.g., `pile[1]`, `game.bigPile`, `game.getPile(1)`, etc. If you do implement a helper function for sorting the stone piles, you do *not* need to include the pseudocode for that function in your report. For simplicity's sake, your function *does not* need to remove duplicates in its output; if two moves result in the same game state (e.g., subtracting two different piles with the same size), you may return both.

In your implementation, you may use the fact that a player will only have up to three valid moves at any given time. You may also wish to reuse a single, static array that you pass into the function to avoid allocating and deallocating memory every time you invoke `getMoves`. Lastly, you *do not* need to use memoization when implementing `getMoves`, though you may do so if you wish.

5 Choose wisely

The `isWinner` function accepts a game state (see Section 4) and outputs `true` if the game is a *winning game* or `false` if it is a *losing game*. Note that you do *not* need to have finished the

`getMoves` function in order to design the `isWinner` function, though the `getMoves` implementation needs to be finished and debugged before `isWinner` can be tested.

Once you have developed a strategy for classifying games and decided on your dynamic programming data structure, you should write up your strategy as a pseudocode algorithm, which you should include in the project report (see Section 8). You may wish to implement a memoized solution for your function, though an iterative dynamic programming strategy would work as well. (Non-dynamic programming solutions will be very slow and will receive major penalties to your group grade.) You may also wish to use a global variable to store and access your dynamic programming data structure.

6 Make your move

The `bestMove` function accepts the current game state, uses the `getMoves` and `isWinner` functions to identify the optimal move, and it returns the game state after this optimal move is made. In the case where there is more than one optimal move, you may return the result of any such move. (In particular, if the game is a *losing game* that the AI cannot expect to win, you may return the result of any valid move.) Note that you do not need to implement the `getMoves` and `isWinner` function to implement `bestMove`; however, `bestMove` is strongly related to `isWinner`, so you may wish to consult with your teammates when designing `bestMove`. As with `isWinner`, you may wish to use a global variable to store and access your dynamic programming data structure.

7 Play the game

Lastly, you will need to implement code that takes input from the player and plays the game.

7.1 Input requirements

Initially, the player should enter three integers, separated by new lines or other whitespace, to describe the initial setup for the game.

On each of the player's turns, they should enter two integers 1–3 to indicate their move. The first number entered should represent the pile to remove stones from, while the second number should represent the pile containing the number of stones to remove. The number 1 should always represent the pile with the most stones, 3 the smallest, and 2 the middle. As a result, the only valid moves will be "1 2", "1 3", and "2 3", unless there are piles with equal numbers of stones.

Once the game is over, the program should terminate. *Do not* prompt the player to play again.

For the purposes of evaluating your code, you may assume that player's input will always be valid (e.g., all values are of the correct type and in the correct range, with no extraneous characters), but you may choose to employ input validation, as this is good practice and may make your testing easier.

7.2 Output requirements

The only required outputs are:

1. After the player decides on the initial number of stones, the program should print "I'll go

first" or "You go first" to indicate who will make the first move.

2. After each move the player or the AI makes, the program should print out the new game state. The game state should be displayed as the sizes of the three stone piles, separated by commas, in decreasing order. E.g., if the three piles have 1, 2, and 3 stones, the game state would be printed as "3, 2, 1".

All other output (e.g., printing a welcome message, prompting the player to enter their next move, etc.) is optional.

8 Project report

In your project report, you should include brief answers to four questions. The first three of these are questions 1, 3, and 4 in Section 3, which are detailed further in that section:

1. How will you represent the game state (i.e., the three piles of stones) in memory?
3. Why is dynamic programming a good algorithmic strategy for this problem?
4. What data structure should you use to store the answers to the various subproblems?

You should also include pseudocode descriptions of all three functions, `getMoves`, `isWinner`, and `bestMove`. *For full credit, your pseudocode must be clear enough that any competent programmer will understand how your algorithm works and could implement your algorithm in their preferred programming language.*

9 Coding your solutions

Your code may be in C++ or Java. If you implement your code in C++, your code should compile and run on the C4 Linux Lab machines. If compiling your code cannot be accomplished by the command

```
g++ -o threepile *.cpp
```

you should include a Makefile that capable of compiling the code via the `make` command.

If you choose to implement your code in Java, you should submit an executable jar file with your source. In either case, your source code may be split into any number of files.

10 Submission

Your submission for this project will be in two parts, the group submission and your individual peer evaluations.

10.1 Group submission

The submission for your group should be a zip archive containing 1) your report (described in Section 8) as a PDF document, and 2) your code (described in Section 9). If your code requires

more than a simple command to compile and run then you must also provide a Makefile and/or shell script. You should submit this zip archive to the "Project 2 (group)" assignment on Canvas.

Be aware that your project report and code will be checked for plagiarism.

10.2 Teamwork evaluation

The second part of your project grade will be determined by a peer evaluation. Your peer evaluation should be a text file that includes 1) the names of all of your teammates (including yourself), 2) the team member responsibilities, 3) whether or not your teammates were cooperative, 4) a numeric rating indicating the proportional amount of effort each of you put into the project, and 5) other issues we should be aware of when evaluating your and your teammates' relative contribution. The numeric ratings must be integers that sum to 30.

It's important that you be honest in your evaluation of your peers. In addition to letting your team members whether they do (or do not) need to work on their teamwork and communication skills, we will also evaluate your group submission in light of your team evaluations. For example, a team in which one member refused to contribute would be assessed differently than a team with three functioning members.

You should submit your peer evaluation to the "Project 2 (individual)" assignment on Canvas.

11 Grading

Report	**50 points**
Questions	20
`getMoves` pseudocode	10
`isWinner` pseudocode	10
`bestMove` pseudocode	10
Code	**20 points**
Compiles and is correct	15
Good coding style	5
Teamwork	**30 points**

Note that if your algorithm is inefficient, you may lose points for both your pseudocode and your submission. Also, in extreme cases, the teamwork portion of your grade may become negative or greater than 30. In particular, if you do not contribute to your group's solution at all, you can expect to receive a 0 overall.