CS61C Spring 2014 Project 2: MapReduce

TA: Sung-Roa Yoon / Shreyas Chand
Part 1: Due 03/19 (WEDNESDAY, NOT SUNDAY) @ 23:59:59
Part 2: TBA

Updates

All updates to the project spec will be listed in this section.

- 03/13 : Project 2 released.
- 03/14 : Bug fix for MovesWritable.java! Please get the fresh copy of MovesWritable if you downloaded your version before the 15th!.

Clarifications/Reminders

- Start Early!
- **Make sure you read through the project specs before starting.**
- This project should be done on either the **hive machines** or the **the computers in 271,273,275, or 277 Soda**. These are the machines where we have hadoop installed and tested for your usage.
- It is suggested that you work with **one** partner for this project. Keep in mind this has to be a different person from the one you worked with on project 1. You may share code with your partner, but ONLY your partner. The submit command will ask you for your partner's name, so only one partner needs to submit. (It would be nice if only one partner submitted).

Goals

The goal of this project is to get you familar with the MapReduce programming model using the Hadoop framework. The first part of the project will give you an opportunity to turn minimax game tree search (an algorithm you're hopefully already familiar with) into a format that is compatible with the mapreduce framework. The second part will require you to run your implementation on a large cluster of Amazon Web Services Elastic Compute Cloud (AWS EC2) servers in order to crunch through a large problem. We hope that by doing this project you will gain an appreciation for the MapReduce programming model, and at the same time pick up a very marketable skill.

Part 1 (Due 3/19 @ 23:59:59)

Background

Connect N

The game we will try to completely solve in this project is a variation of Connect Four. Connect Four is essentially like Tic-Tac-Toe, but players are required to "drop" their piece into a column rather than arbitrary place it. Furthermore, instead of winning by placing three pieces in a row, they must place four. For a better idea of game mechanics, you can play the game here (or on any of a plethora of flash game websites) and read more about it on Wikipedia.

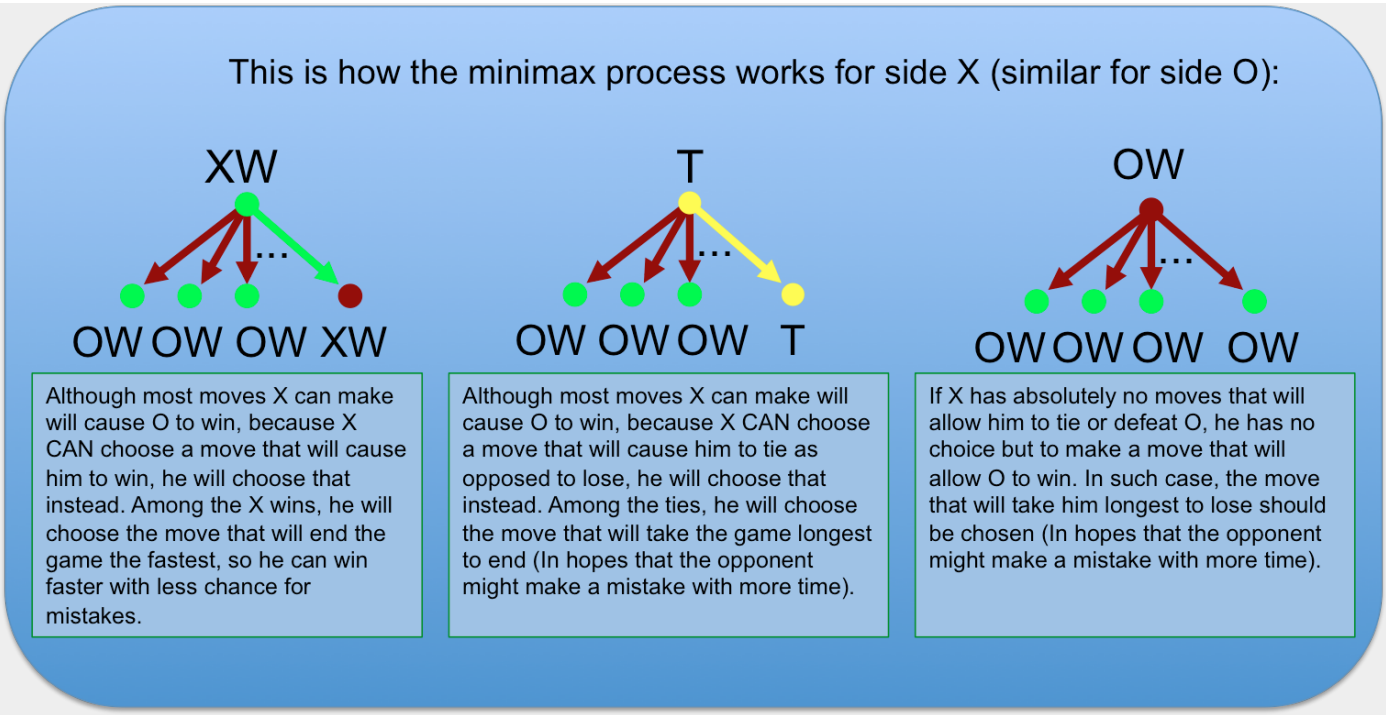
For the purpose of this project, we are going to be solving smaller versions of the complete game. Our game board will only be a maximum of 5 by 5 (instead of the standard 6 by 7). This is in order to limit the amount of time and space your code will require to run. In fact, for the first part we recommend you solve a game board of 3 by 3, with the win condition being three, instead of four, pieces in a row. With a smaller case, it should also be easier for you to verify the correctness of your output. Of course, you can use larger parameters if you have enough time, and space on your hard drive.

Minimax Game Tree Traversal

The actual algorithm we want you to implement consists of two main parts: 1) Constructing a full game tree for connect four, and 2) Solving the game by running minimax to label each game state with a win, loss or tie value.

In the first part, you will begin from an empty board, generate all possible subsequent game board configurations and link each of the new states to the original parent. This process will be repeated for each of the generated states until you reach a game state that is a winning state for either one of the players.

In the second part you will walk through the tree you have generated from the bottom up and propagate up the win/loss/tie value according to minimax. Essentially, there are three different scenarios (as detailed in the diagram below) and in each case you simply have to choose the optimal option to update your value. We will record the win/loss/tie state, and the number of moves it will take to get to that state from the current state.



Getting started

Copy the files in the directory `~cs61c/proj/02` to your `proj2` directory, by entering the following command:

```
$ mkdir ~/proj2
$ cp -r ~cs61c/proj/02/* ~/proj2
```

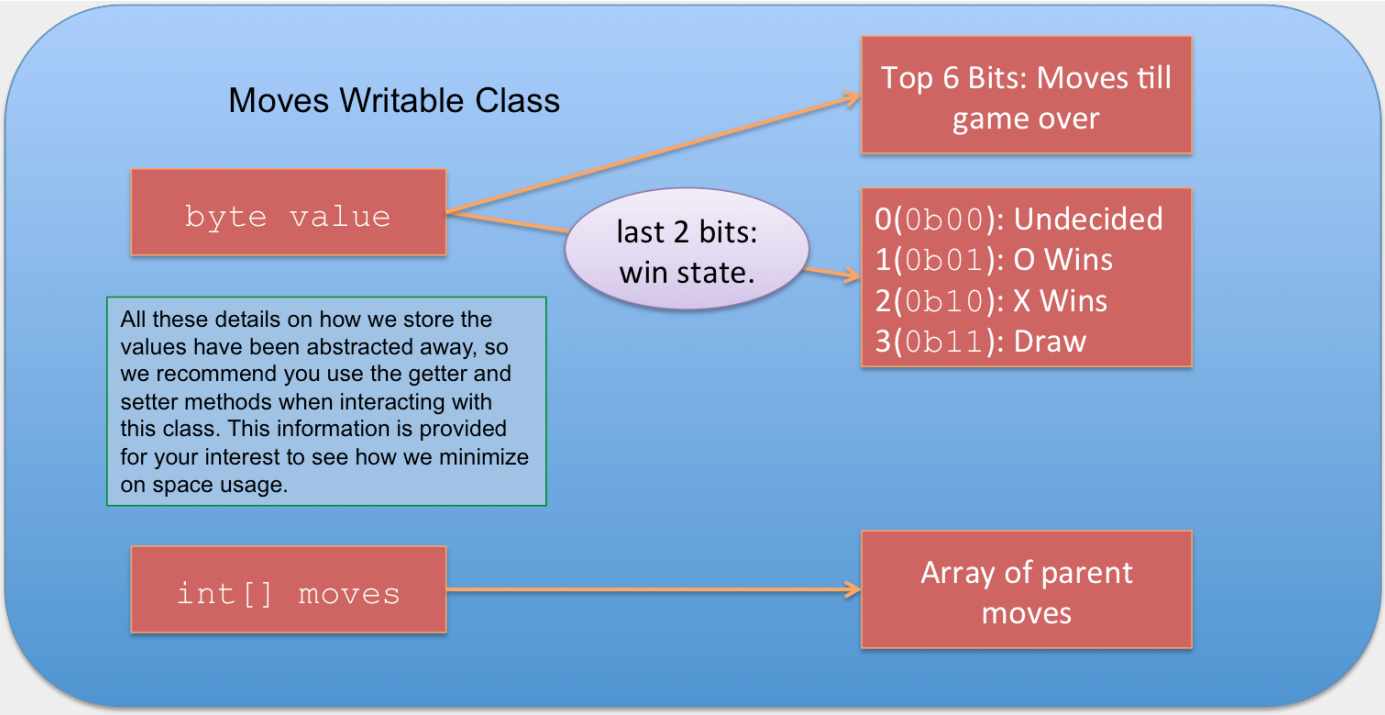
The files you will need to modify and submit are:

- `PossibleMoves.java` contains the code for the second MapReduce job that produces the full game tree. The first run of this job receives as input the output produced by the MapReduce job in `InitFirst.java`.
- `SolveMoves.java` contains the code for the third MapReduce job that traverses each layer of the full game tree and performs minimax selection to score each gamestate in the tree.

You are free to define and implement additional helper functions, but if you want them to be run when we grade your project, you must include them in the above classes. **Changes you make to any other files will be overwritten when we grade your project.**

The rest of the files are part of the framework and should not be modified. However, you will need to look at them as well. The particularly interesting/useful classes are bolded.

- `Makefile` defines the configurations for the run commands including the size of the game board and how many pieces in a row are required to win
- `Proj2.java` contains `main`, which sets up the four MapReduce jobs to be run
- `InitFirst.java` The first MapReduce job that creates the initial input (empty game board) to `PossibleMoves.java`
- `FinalMoves.java` contains the code that cleans up the output from the final run of `SolveMoves.java` to create the table of mappings from game state to moves until win/loss/tie.
- **`TUI.py`** contains a TUI (Text User Interface) for trying the game out with your solved solution! You run the TUI by typing `python TUI.py WIDTH HEIGHT CONNECTION_TO_WIN`, and place pieces on the board by typing the column where you want the piece dropped. (Zero indexed).
- **`Proj2Util.java`** contains the hasher and unhasher (the hash is invertible to allow for this), which you need to generate keys for most map reduce outputs, and also contains a game finished check that lets you know if either side won the match or not.
- **`MovesWritable.java`** is a `Writable` class that can be used to store win/loss/tie state, the number of moves to get to that state and an array of all the parent game states associated a particular game state.

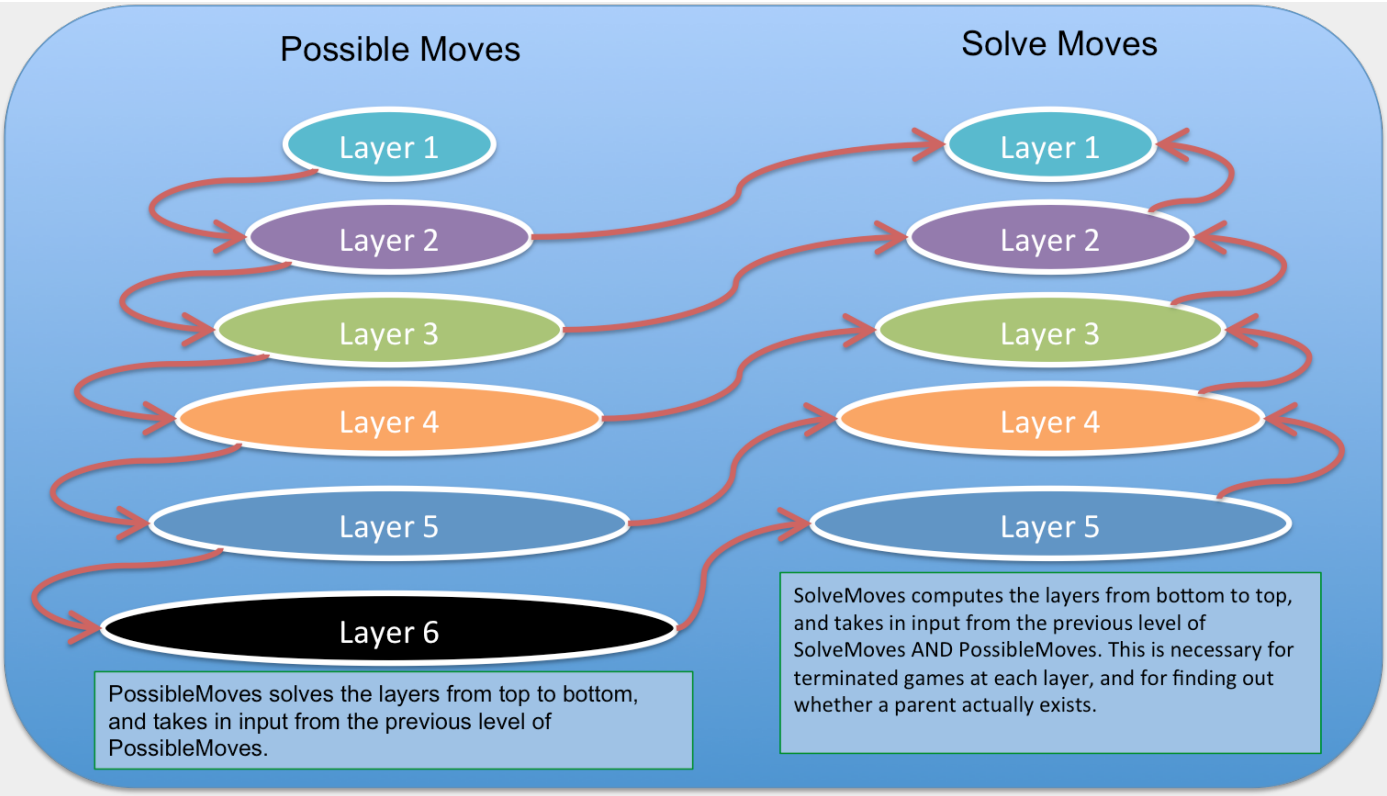


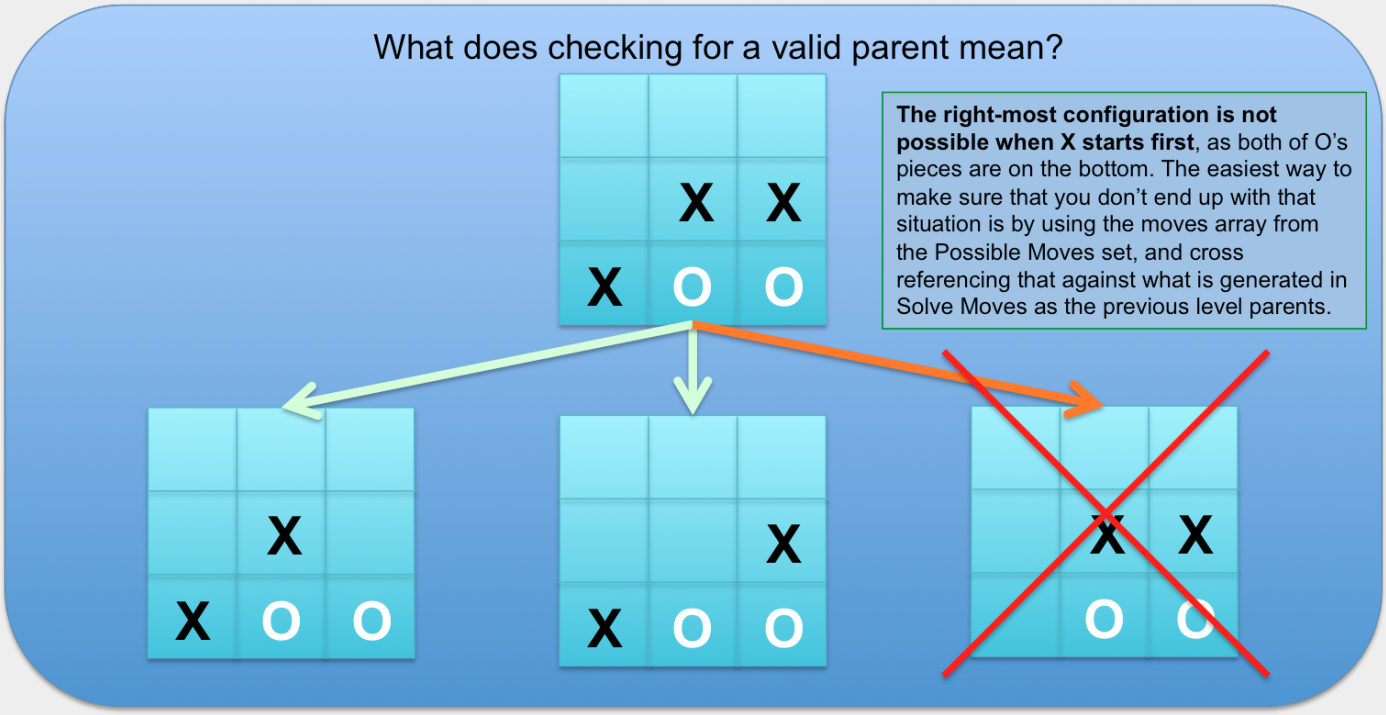
Your Job

Task A (Due 3/19): Your task will be to implement the two MapReduce jobs.

The first mapreduce job you have to write is in `PossibleMoves.java`. This job is called once per level in the full game tree, with each call to map getting as input the output of the previous map phase. The first run of the job gets the output from the `InitFirst` mapreduce job (this basically gives it an empty board representation). Therefore, your map function should take the game state represented by the key (a game state that has been hashed into an `int` using the `hash` method in the `Proj2Util` class). The reducer for this job sets up the actual game tree by emitting a `MovesWritable` object associated with every passed in child gamestate. The `MovesWritable` object will at this point encapsulate all the possible parents and a win/loss/tie value for this state. The parents is simply a combined list of all corresponding values for the provided key. The win/loss/tie value needs to be determined by you by scanning through the game state representation. If there are `connectWin` pieces in a row somewhere in the game board, you should determine this and set the correct win value (`0b01` for a `O` win, and `0b10` for a `X` win). If this is the `lastRound` and you don't find a win, then you should set the status to a tie (`0b11`). If none of these conditions exist, then the value for this state is undecided (`0b00`).

The second mapreduce job you have to write is in `SolveMoves.java`. This job is called once per level in the almost full game tree (not the bottommost level), working bottom up instead of top down like `PossibleMoves`. The first run of the job gets the output from `PossilbeMoves` exclusively, while all of the remaining levels of it gets the output from `PossibleMoves` from the layer below as well as the output from `SolveMoves` from the layer below. Therefore, your mapper should receive the key of the current board state, and the value of a MovesWritable class, which contains the board state, moves to end, and parents of the current board state. From here, your mapper should pass each instances of the parent state as output key, and the value of the current board as the output value. That means that your reducer will receive all of the parents which contain the values from their children, so you can generate the next level up. You should use minimax to determine which value should be used, with the help of the diagram above, and you should also generate the parents of this state. It is okay if you don't generate the parents perfectly, as long as you AT LEAST generate all of the parents. (As in you can generate the impossible parents here, as long as you can filter them out properly). The filtering should be aided by the outputs of `PossibleMoves` which should have distinctly different values from the outputs of `SolveMoves`. Think about what's different about the two, and write the code accordingly! Remember, if you have impossible board states in your final output, you will be deducted points!





Here are some tips and reminders:

- Look at `Proj2.java` to understand how each of the MapReduce jobs will interact with each other. We will be calling each of the two mapreduce jobs you write multiple times in a loop. They should be able to handle this.
- Use the `MovesWritable` class to link together game states and hold win/loss/tie data about them.
  - The `value` field in this class actually holds two different pieces of information. The two low order bits are used to signify the game state status. The bits are `0b01` if the corresponding state will lead to a win for `O`, `0b10` if it will lead to a win for `X`, `0b11` if it will lead to a tie, or `0b00` if the state has not yet been decided.
  - The remaining 6 bits of the byte represent the unisgned number of moves it would take to reach the state signified by the first two bits.
  - The `moves` field holds a list of hashed game states that can lead to this game state.
  - The `counter` field is the length of the moves field. It is used to help us when restoring a `MovesWritable` instance from the output context.
- The `FinalMoves` mapreduce jobs runs at the end, after all your code has executed to clean up any leftover intermediate key value pairs from the output of `SolveMoves`.
- The first run of `PossibleMoves` will need the empty board representation to work correctly. The `InitFirst` mapreduce job is provided to give the correct input format. Take a look at it to see exactly what the beginning key-value pair provided to your code will be.
- The `Proj2Util` class is extremely inportant as it provides the hashing function. They allow us to compress a game state to fit in a 4 byte integer. While you don't have to fully understand how it works, be sure to understand the input and output formats for both the `hasher` and `unHasher` functions. Essentially, the string representation should be as long as the area of the gameboard with which we are working. The slots of each column are grouped together, with the bottom slot of the first column being the left most character, and the top slot of the last column being the right most

character.

Debugging and Testing

The provided `Makefile` gives you the ability to easily compile and run your code using variable arguments.

```
$ make // Compiles your code

$ make proj2 // Runs your code with the default 3 by 3 board and connect 3

$ make proj2 WIDTH=n HEIGHT=m CONNECT=i // Make sure to actually type the all-caps letters,
                                        //and generate your output with board width of n, board height of m, and connect to win of i.

$ make proj2 WIDTH=4 HEIGHT=5 CONNECT=3 // For example, this would run a four by five board, with connect 3.
```

The provided `TUI.py` will help you test your output, as you can use it to play against your AI, or see how the AI plays optimal moves on both sides.

```
$ python TUI.py 4 5 3 // This runs the TUI.py with your output, with board width of 4, board height of 5, and connect 3 to win.
```

In the `ref_out` directory, we have also provided you with the correct output we expect for two different game types, 2 by 2 Connect 2 and 3 by 3 Connect 3. You can use this to verify whether or not your program is working for at least the small case.

While you are working on the project, we encourage you to keep your code under version control. **However, if you are using a hosted repository (i.e. GitHub, BitBucket, etc), please make sure that it is private, or else it will be viewed as a violation of academic integrity.**

If you mistakenly break parts of the skeleton, you can always grab a new copy of the file by running the following command, but BE VERY CAREFUL not to overwrite all your work.

```
$ cp ~cs61c/proj/02/<NAME OF FILE> ~/proj2
```

Before you submit, **make sure you test your code on the Hive machines.** We will be grading your code there. **IF YOUR PROGRAM FAILS TO COMPILE, YOU WILL AUTOMATICALLY GET A ZERO FOR THAT PORTION OF THE ASSIGNMENT** There is no excuse not to test your code.

How your project will be graded:

- First, it will be graded on the correctness of the produced solution. This will be done by comparing your output file containing the solved values with the file that the staff solution outputs. Make sure that your file does not have any game states that cannot exist. (For example a board with moves played after a win state, or a board containing illegal moves like floating pieces or repeated turns by a particular side.)
- Second, we will be comparing the performance of your code against the staff solution. This includes both run time and disk space usage.
    - As a reference, running `make proj2 WIDTH=4 HEIGHT=5 CONNECT=3`, should take somewhere between a system time of a minute and a minute and a half.
    - The above command should also produce an output somewhere between 15-20 MB. You can verify this by running `du -h`, and checking to see that your `all_data` directory is no larger thatn 20MB.

Submission

The full proj2-1 is due Wednesday (3/19/2014), to account for the exam on the previous Wednesday. To submit proj2-1, enter in the following. You should be turning in `PossibleMoves.java`, and `SolveMoves.java`.

```
$ cd ~/proj2
$ submit proj2-1
```