Assignment 4

10 October 2012

Due Wednesday 17 October at 1am

The purpose of this assignment is to experiment with the Minimax game-tree algorithm, and develop your own heuristic for the game *Connect Four*.

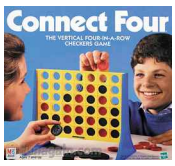

Figure 1:

Getting started

Open any file within your `cs162` folder in the editor, and then choose the "Sync with git" option from the Tools » External Tools menu.

Now you should have an `a4` folder, with several C++ files:

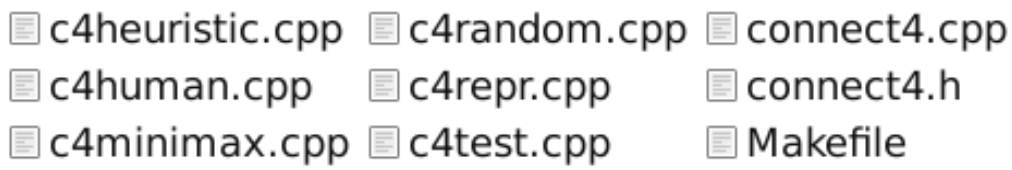

Figure 2:

You are welcome to browse through all of these, but your work will mainly be in `c4heuristic.cpp`. Open that file now, and then use Tools » External Tools » Run. After it builds successfully, you will be presented with a menu:

```
Welcome to
    --=| C O N N E C T   F O U R |=--
           (c)2012 Christopher League

  1. Human
  2. First available
  3. Random
  4. Minimax depth 2 null heuristic
  5. Minimax depth 4 null heuristic
  6. Minimax depth 6 null heuristic
```

```
  7. Minimax depth 2 your heuristic
  8. Minimax depth 4 your heuristic
  9. Minimax depth 6 your heuristic
Select Player 1:
```

You get to select which mechanism to use for each player, so you can play a game with two humans, or a human against a variety of computer algorithms, or two algorithms against each other.

- *First available* just chooses the left-most column that has space available.
- *Random* chooses a random column, as long as it has space available.
- *Minimax with null heuristic* searches to the specified depth, but if no end-of-game is found along a particular path, it just returns a score of zero. Thus, the minimax scores will be $\pm\infty$ (if one or the other player wins), or zero.
- *Minimax with your heuristic* – for now, this is the same as the null heuristic, but you have a chance to improve it by developing your own heuristic in `c4heuristic.cpp`. See below for details on how to do that.

Experiments

In this section, you can experiment with the minimax algorithm without writing any code. Write up what you find using gedit, and save it to `results.txt` in the `a4` folder.

1. Run a match with both players using the "first available" strategy. Who wins?
2. Run 10 matches with both players using the "random" strategy, and keep track of which player wins each time. Are they evenly matched? Might there be a first-move advantage?
3. Run 5 matches where random plays first against minimax depth 2 (null heuristic), and then 5 matches where minimax depth 2 play first against random. Does minimax always win?
4. Repeat the above experiment, but with random against minimax depth 6.
5. Play a few games as a human against minimax depth 6. Is it blocking your wins in simple cases? Was it able to win?
6. Run minimax depth 4 against minimax depth 6 – let each have a chance to be player 1. What are the results in each case, and are they as you expected?

Coding

Now it's your turn to try to develop a heuristic. You can start with a very simple one that will make a small improvement, and then get as fancy and intelligent as you like. After the due date, we'll try to have your heuristics compete against each other, and with mine, in a tournament.

Here is the function you will write in `c4heuristic.cpp`:

```
int heuristic(board b, color k, unsigned lastMove) {
    // Replace the following with your board evaluator.
    return 0;
}
```

You are given a representation of the board, `b`, and the identity of the current player (which we call a color), `k`. You should return an integer score where a positive number means that board is good for player `k`, and negative number means it's bad for player `k`.

The color is one of the constants `RED` (representing player 1) or `BLUE` (representing player 2). There's a convenient function `nextColor` for determining the opposite color, like this:

```
    color opponent = nextColor(k);
    cout << "I am " << colorName(k) << ", you are "
         << colorName(opponent) << '\n';
```

Thus if we're evaluating a board where `k==RED`, this will output "I am Player 1, you are Player 2," or the opposite if `k==BLUE`.

For investigating the board, you can use the constants `NUM_COLUMNS` (which equals 7) and `NUM_ROWS` (which equals 6), and this helper function:

```
color playerAt(board, unsigned c, unsigned r);
```

(If you weren't aware, `unsigned` is an integer that is non-negative.) The function call `playerAt(b,c,r)` is given board `b`, column `c`, row `r`, and returns `RED`, `BLUE`, or `EMPTY`. If you call it with a column or row that is out of range, it will abort the program with an error message. (Recall that we usually start from zero, so valid columns are 0–6, even though in the user interface, they are labeled 1–7.)

You probably noticed in your experiments that minimax prefers the right-most column. If it can't see to the end of the game, then all boards look equally good, because the heuristic for now is always zero, so it just picks the last (right-most) one that it tried.

```
Player 1 chose 7

  |- - - - - - -|  0x240000000000000
  |- - - - - - -|  h=3
  |- - - - - - -|
  |- - - - - - -|
  |- - - - - - -|
  |- - - - - ->1|
  +-+-+-+-+-+-+-+
   1 2 3 4 5 6 7
```

So let's start with a simple idea: that the middle column is more valuable than the right one! To make our heuristic do that, we can just award a point for each piece the player has in the middle column. Here's some code:

```
int heuristic(board b, color k, unsigned lastMove) {
    int score = 0;
    int middle = 3; // Labeled 4 in the user interface
    for(unsigned r = 0; r < NUM_ROWS; r++) {
        if(playerAt(b, middle, r) == k) {
            score++;
        }
    }
    return score;
}
```

With this heuristic, at depth 2, the computer will always prefer the middle. Once the middle is full, it will go back to its comfort zone on the right. (The heuristic is player 1 here.)

```
  |- - - 2 - - -|  0x240000570000000
  |- - - 1 - - -|  h=3
  |- - - 2 - - -|
  |- - - 1 - - -|
  |- - - 2 - - -|
  |- - - 1 - ->1|
  +-+-+-+-+-+-+-+
   1 2 3 4 5 6 7
```

Further ideas

There are lots of ideas you can try to implement as heuristics. One of the simplest is to come up with a weight map for each square on the grid. We investigated something like this in class, observing that the number of *horizontal* wins possible in each column produces:

```
1 2 3 4 3 2 1
```

You could iterate over the whole grid (a two-dimensional loop over `r` and `c`) and add up the weights for our pieces. Maybe also subtract the weight for our opponent's pieces. If you consider not just horizontal, but vertical and diagonal moves, the weights are more complex, and your heuristic is probably a better player.

A weight grid is just one possibility, but we can also directly count at all the possible wins that remain for each player. You'll have to be pretty careful with loop arithmetic to get that one right!

See what you can do, and enjoy! Don't forget to "sync with git" before the deadline.

Resources

While developing this, I was able to play my heuristic against some other AIs on the web, such as this one:

- http://www.mathsisfun.com/games/connect4.html

Just play Human-vs-computer on one system, and Computer-vs-human on the other, then copy their moves back and forth.