James Pine Has A Blog

- blog
- archives
- about

IBM Watson vs. Google Search
Losing Six Years of Photos and Recovering Them Five Years Later

Wednesday, March 9th, 2011…18:07

C O L O R F L O W

Jump to Comments

A some point last year I became infatuated with a game commonly known as Flood-It by a company called LabPixies. My obsession resulted in the creation of a clone I call C O L O R F L O W which addresses the major faults I found with the game, namely there was no way to:

- undo an errant move
- play the same board multiple times to improve on my own score
- know if my sequence of moves was optimal
- play the same board a friend had just played

First I needed a way to represent the state of the board and the game playing functionality. There are 6 colors randomly spread over a 14 x 14 grid. My first instinct was to use a list of lists with the value of each element being a number between 0 and 5 inclusive, but eventually I decided to use a single dimension 196 element list moving left to right and top to bottom e.g. in a 3 x 3 grid:

```
[0][1][2]
[3][4][5]
[6][7][8]
```

This allowed me to reference any element on the board with a single number instead of two (x, y) which made things a bit more efficient.

Changing the colors of squares on the board requires knowing which squares are currently “flooded” and which adjacent squares are of the new color which will become “flooded”. Given a square, 42, in a 14 x 14 grid it’s easy to see that square 28 is above it, 56 is below it, 43 is to the right and there is no square to the left. While computing those is trivial, for efficiency reasons, I decided to build up a dictionary of squares->neighbors. Those two data structures were all I needed in order to write the color changing functions. I later added a set to keep track of which squares had been “flooded”.

At this point I could play a sequence of colors and ask the board whether it was flooded or not, so I decided to write a program to flood a given board in the fewest moves. The Flood-It version of the game caps the number of color changes for a 14 x 14 board at 22. At any point in the game you have a choice of up to 5 colors for the next move, so that’s 5^22 possible color sequences(2.4 quadrillion). Even if my code only took 1 microsecond to play each sequence of colors, it could take more than 75 year of continuous processing time to find the best one. Yes, there are cases where the branching doesn’t multiply by 5 at every step if fewer colors are valid candidates and yes if I found a color sequence of length 16 that worked right off the bat I could stop evaluating every subsequent sequence after 15 moves, but it would still take a very long time, especially since my code takes more than a microsecond to play a sequence of color changes.

So brute force was not an option. I experimented with a genetic algorithm as a sequence of colors mapped very easily to the idea of a chromosome. However, I found that crossover between color sequences often times created invalid color sequences. The compensation tactics (repairing chromosomes or using very large populations) were CPU intensive. I wanted this solver to finish in less than 5 seconds and the GA wasn’t even coming close.

My next strategy was to use a constrained lookahead. The idea was to generate and score all possible color combinations up to a fixed number of moves ahead e.g. 7. Then pick the best sequence and repeat until the board was flooded. This way I never had more than 5^7 sequences to evaluate so instead of 2.4 quadrillion, I only had to evaluate (5^7)*3 [234,375] for any game that could be solved in 21 or fewer moves. So how does one determine the best sequence of colors part way through the game?

I had already written the basis of a scoring algorithm in the fitness function of my genetic algorithm as a series of checks:

First, if the board is flooded, return 1 + (1 / length of color sequence). This always yields a number > 1 with higher number denoting shorter color sequences which flood the board.

If the board isn’t flooded, then early in the game it’s important to maximize the number of endpoints/squares which are candidates to be flooded on the next move. It’s also an important characteristic towards the end of the game if we have not eliminated many colors. Another important metric at any point in the game is the number of currently flooded squares. Finally,