CS4200

Project 2

Ryan Skinner

Both of my algorithms had a 99% success rate. The reason they aren't always solved is because I put a cap on how many iterations to try before giving up. The trade off here is, is it worth running more iterations to solve such a small number of puzzles? Realistically I could remove this cap and I could still find a solution in less than a second.

My results are the following:

Algorithm	Iterations	Solved	Success rate	Average Time	Average Iterations
Annealing	100,000	99936	99.94%	94	13951
Genetic	100,000	99927	99.93%	51	14344

Optimizing

I wanted to keep my success rate above 99% while keeping the program running as fast as possible. Since failures take longer than success, these went hand in had.

For simulated annealing I used exponential decay to lower the temperature, as using linear was ridiculously slow. To optimize this algorithm I needed to find a balance of a fast decay rate and threshold, while keep solution times high. To achieve this I found the lowest ratio of iterations to success rate, where iterations is $\log_{decay}(threshold)$. I also increased the speed a lot by not generating $e^{\Delta/t}$ unless it is needed, when before I would always generate it.

For the genetic algorithm I needed to find the optimal values for 4 different variables: mutation rate, population size, survival rate, and generations. Since the run time of this algorithm is populations*generations, I searched for a combination of the 4 that produced the lowest ratio of iterations to success rate. I also switched away from a random parent selection, to evenly iterating over the top 4. Since I needed to randomly pick 2 of 4 parents to generate 48 children I figured it would probably be roughly the same breeding patterns, this method also eliminated asexual reproduction. This cut my average time down quite a bit. I also could have changed my sort method to be a modified merge sort, a normal merge until pivot point was 4, that way I would only completely sort the first 4 elements in the array instead of all of them. I'm not sure how much of a boost this would give my program so I left this modification out.

However the best values here didn't always yield high results, they were optimized successful runs per iteration, not success rate of the iteration. This was easier to fix with the genetic algorithm because I just had to raise generational limit. With simulated annealing I had to tweak both the decay rate and the cut off value.

In the end I ended up with two similarly performing functions in terms of iterations but annealing was about half as fast.

Finally a note on Ryzen processors, which this was tested on. They claim to have machine learning branch prediction, which seems to be apparent, the more times I ran the test the better my numbers got. An average over 1,000 iterations would yield time in milliseconds while 100,000 would bring that down to tens of nanoseconds. This caused an issue when I wanted to quickly test which of two values obtained better results. The second one would have better results (if they are comparable values) because of the slight speed increased caused by this new technology. I could remove this by running a few thousand tests first, and then start timing them but I didn't think this project needed numbers that accurate.