George Matta

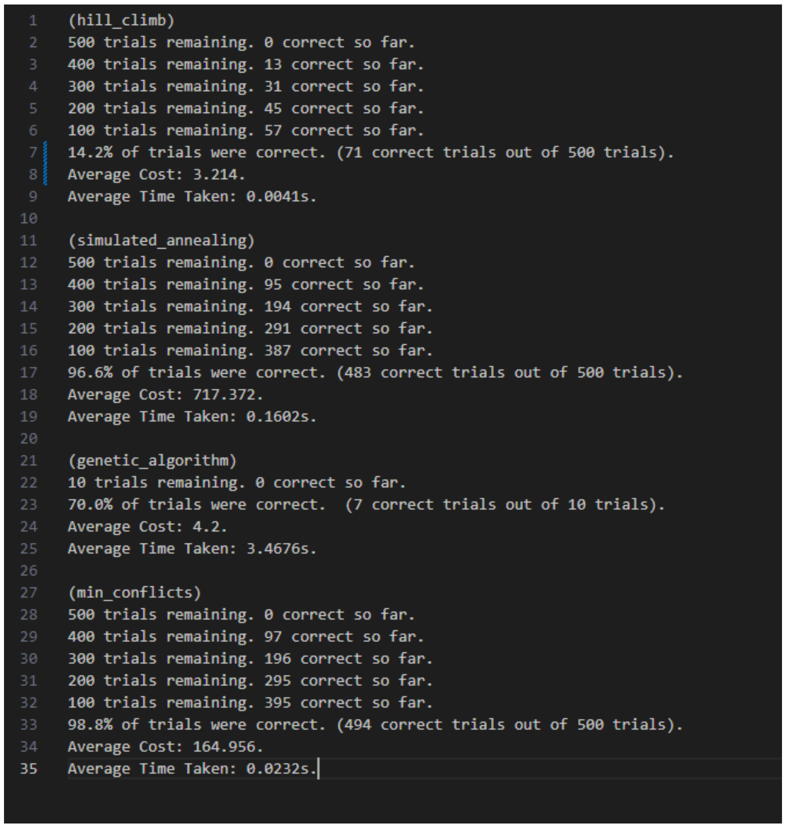
CS4200 - Project 1

Report

N-Queens Solvers: HC, GA, SA, MC

This report will explore a variety of possible solutions for the N-Queens problem (specifically, with 8 queens). Though the assignment specified the use of the Hill-Climbing algorithm, and one of Simulated Annealing, Genetic Algorithm, or Min-Conflicts, I decided to implement all four solvers to collect as much information as possible.

These results are shown here:



```
(hill_climb)
500 trials remaining. 0 correct so far.
400 trials remaining. 13 correct so far.
300 trials remaining. 31 correct so far.
200 trials remaining. 45 correct so far.
100 trials remaining. 57 correct so far.
14.2% of trials were correct. (71 correct trials out of 500 trials).
Average Cost: 3.214.
Average Time Taken: 0.0041s.

(simulated_annealing)
500 trials remaining. 0 correct so far.
400 trials remaining. 95 correct so far.
300 trials remaining. 194 correct so far.
200 trials remaining. 291 correct so far.
100 trials remaining. 387 correct so far.
96.6% of trials were correct. (483 correct trials out of 500 trials).
Average Cost: 717.372.
Average Time Taken: 0.1602s.

(genetic_algorithm)
10 trials remaining. 0 correct so far.
70.0% of trials were correct.  (7 correct trials out of 10 trials).
Average Cost: 4.2.
Average Time Taken: 3.4676s.

(min_conflicts)
500 trials remaining. 0 correct so far.
400 trials remaining. 97 correct so far.
300 trials remaining. 196 correct so far.
200 trials remaining. 295 correct so far.
100 trials remaining. 395 correct so far.
98.8% of trials were correct. (494 correct trials out of 500 trials).
Average Cost: 164.956.
Average Time Taken: 0.0232s.
```

Most prominently, the min-conflicts and simulated annealing have the highest success percentage out of the other solvers. To explore this further, we must first discuss the hill-climbing algorithm. Going into the project's development, I was aware that the HC solver would only be able to solve ~14% of problem instances. The hill-climbing algorithm works because it will procedurally choose the move that increases the value of the solution (in this case, decreases the number of attack pairs for the queens), and will keep doing that until all possible moves decrease the value. In essence, the hill-climbing algorithm will climb up to a local minimum and terminate. Because of this, the ability of the hill-climbing algorithm is strongly dependent on the input board given to it. If it were possible to graph every possible state and the number of attacks, as the input state is closer to the global maxima of the graph, it would be more likely to be solved by the hill-climbing algorithm. For this reason, the hill-climbing algorithm is stuck at around 14% success. To mitigate this, we must allow the solver to sometimes make bad choices (so that the board bounces around the value graph rather than being stuck at the closest local minima). This is where simulated annealing comes into play. SA works very similarly to hill-climbing; an input board is given, possible moves are ranked by their values, and moves with better values are guaranteed to be taken. Further, however, even if a move is of lower value, it has a chance of being taken anyway. This change is based on the "temperature". Simply put, the temperature starts at a relatively high value and slowly decreases every iteration. Through the equation e^(difference in value/temperature), a probability is determined. Effectively, as the temperature decreases over time, so does the probability of choosing bad moves. As we can see, this significantly improves the success rate of the N-Queens solver.

Another possible solver, the min-conflicts algorithm, works differently from hill-climbing and simulated annealing. Rather than choosing the next move out of all of the possible moves of the board, the min-conflicts algorithm will choose to only move a queen that is under attack. While this step relies on randomness (we randomly choose one of the queens under attack), the algorithm will deterministically choose the move that moves the queen out of danger. Meaning, that the move will be made that minimizes the number of attacks as much as possible on the randomly chosen queen that is under attack.

Finally, the genetic algorithm. Genetic Algorithms work by creating a random, initial, population, and then procedurally improving it over several generations. We do this by selecting two relatively strong individuals, creating children individuals based on those parents, and then randomly mutating those individuals to create the next population. Objectively speaking, genetic algorithms do not prove to be viable options for solving the N-Queens problem with a relatively low N value (like 8). The issue resides in the loss of genetic diversity for the solutions found. More on this will be explored later, but I will first discuss how I mitigated this issue. The genetic algorithm has a variety of parameters that can be adjusted and fine-tuned; mainly the number of individuals in a population, the number of generations, and the probability of mutation for the created children. For my use case, I used a relatively high number of individuals per population (750) with a relatively low number of generations (10). With N-Queens, there aren't as many possible moves as other genetic algorithm applications (TSP, for example) so it'll converge quickly and then be hard to improve past 1 attack for however many generations you have. With a bigger population, however, (with a not-that-low mutation probability), you end up with a lot of genetic diversity and hopefully a successful result because of it. This forces it to take considerably longer to terminate. Of course, the GA does end up succeeding in terms of solving at least 3 instances of the genetic algorithm (in most cases, it would get around 50% of the input cases), but it does take at least 3 times as long as other solvers.