Rachel Chiang

Project 2: 21-Queens

CS 420.01

Solving n-Queens ($n = 21$) Using Steepest-Ascent Hill Climbing Algorithm and Genetic Algorithm

The n-Queens problem involves an $n \times n$ board that is populated by n queens. Solving it requires all n queens to be placed on a board, situated such that none of them are attacking another. In this project, there will be 21 queens. More formally, this problem can be defined as follows:

Initial State	A board populated by 21 queens, with one queen per column, arbitrarily placed in rows.
Actions	Move a queen up or down the rows of its own column. In this case, we are ignoring horizontal (column) movements because they have already been assigned to their own column, which is closer to the problem solution than if they weren't in their own column. By extension, diagonal movements are ignored too.
Transition Model (Successor)	Any one state has 420 successor states because there are 21 queens to test. Since a queen is already situated in one square, it would only be able to move to the 20 other squares in its column. Thus, $(21\ queens) \times (20\ moves) = (420\ states)$.
Goal Test	For hill-climbing, we want $h = 0$, where h is the number of attacking queens. For genetic algorithm, we want $h = 20 + 19 + 18 + \cdots + 2 + 1 = \frac{20(20+1)}{2} = 210$, where h is the number of nonattacking pairs of queens.
Path Cost	For hill-climbing, +1 for a queen's movement. For genetic algorithm, +1 for every new population produced.

Problem Definition for 21-Queens.

This problem will be attempted by using the steepest-ascent hill climbing approach (HC) and the genetic algorithm approach (GA). Both are local search strategies, which implies that every solution found will be complete but not necessarily optimal. Even so, local search strategies shine in problems that require a lot of memory and that may have too vast of a search space, which makes them attractive for the n-Queens problem. The steepest-ascent hill climbing will begin with a single state and will move one randomly-selected queen at every iteration to a different location in its

column, prioritizing moves that reduce the overall h value. The genetic algorithm will begin with k states in a population and select two states (semi-randomly, since although the selection is random, the more fit individuals will have a higher probability to be picked) to breed and produce one child. The new population will contain k children and the process will repeat. In this project, the state of a board will be represented by a string of capital letters, such as "ABCDEFGHIJKLMNOPQRSTU". In terms of scalability, this is of course not ideal, because this means that n can only be at most 93 or 94, depending on whether or not the space should be valid or not. Since this is using capital letters of the alphabet from A to U, the ASCII offset is 65.

Type	Average Search Cost	Average Time [seconds]	Percentage of Optimal Solutions
HC	388.538	1.2081	99.8%
GA	3899.334	108.2796	6.0%

The average search cost (see path cost above for more details), average time in seconds, and percentage of optimal solutions generated for the hill-climbing (HC) and genetic algorithm (GA) strategies. Results were taken over 500 randomly-generated cases on each algorithm.

The steepest-ascent hill-climbing algorithm turned out to be extremely successful, whereas the genetic algorithm came out to be not even close to passable, and to make matters worse, each attempt takes an unpleasantly long time. Further details regarding the two will be given below, individually.

The steepest-ascent hill-climbing algorithm was both very simple to implement and very successful in finding an optimal solution. Out of curiosity, I forbade sideways movements (moving a queen to a square that yields the same h value as her current location) and resulted in absolutely no optimal solutions in ten iterations over m movements such that $m = \{1000 * i,\ where\ i \in [1,6]\}$, making a total of 60 attempts at solving a puzzle without sideways movements. However, allowing sideways movements improved the algorithm tremendously; the number of movements needed to find an optimal solution rarely exceeded 500, and the algorithm almost consistently would find an

optimal solution. This is expected because local searches have an issue of getting trapped in local minima. As seen above, 500 iterations resulted in a 99.8% success rate for optimality.

The genetic algorithm approach turned out to be generally unsuccessful, which was, admittedly, somewhat unexpected, so there will be more details provided regarding this one. The general implementation for the algorithm is fairly straightforward, but there are a few components to it that appear to influence the process and results in a significant way.

First, to go over this project's implementation: the selection process was determined by the fitness function, which was decided as $h = nonattacking\ queens$. As stated previously, the selection is semi-random; random parents that are non-identical (in this case, meaning that the parents' memory locations were distinct) were selected, but the probability to pass for selection as a parent was dependent on a survival rate, which was dynamic according to the current population. The relationship is as follows: $s_i = \frac{f_i}{\sum_{j=0}^{k} f_j}$, or, in English, the survival rate of an individual i (s_i) is equal to its individual fitness (f_i) divided by the sum of the fitness of each individual (f_j) in the population of k states. In the crossover phase, a random index would be selected to slice the two parents. For the results given above, the mutation probability was set to 0.05 (or 5%), the generations (corresponding to the creation of new subsequent populations) to 4000, and the population size to 50. Mostly, the genetic algorithm on 21-Queens took a while, so limited preliminary testing was conducted before locking in certain parameters (see Appendix, Table 1 for exact values). Adjusting any one parameter alone generally did not yield any significant changes. Increasing the population size is risky, since it increases both the space and runtime complexities, but the latter posed as a bigger issue. To counterbalance this, adjusting the generations is one possibility, but GA requires time to explore many possible solutions since it relies so heavily on randomness and probability. Increasing the

mutation probability to be more than 0.05 tended to prevent the algorithm from converging on one solution, thusly reducing optimality in the solution.

Investigation revealed that the population loses diversity very early on in the genetic algorithm (the population became fairly uniform after a mere 1000 generations), which is extremely problematic because a faster convergence means that the algorithm will explore fewer solutions, opening up the possibility of stagnating at a local maximum, thereby wasting computation time by mating nearly-identical parents. There are some adjustments that could be considered to avoid this early convergence, which are reminiscent of those made to improve hill-climbing in that they try to allow nonoptimal decisions to the mix. It may be worthy to explore changing one or some combination of the following options: the population size, the selection procedure, the crossover procedure, or the mutation probability. I explored two of them; I adjusted the mutation probability and increased the population size. For the first attempt, I made the mutation probability change linearly. It decreased over time according to a counter that corresponds to the number of allowed total generations. The counter is initialized to be 4000 and decreases by one with each new population; in other words, the mutation rate starts out at 100% but drops over time. This proved to be effective at slowing the rate of convergence (see Appendix, Graph 1), which in turn allowed for wider exploration, but this change alone was insufficient. The percentage of optimal solutions ended up much lower than the original version with the static 0.05 mutation probability. For the second attempt, I maintained this linearly-changing mutation probability and doubled the population size, which, although nearly doubling the average search time, significantly improved the success rate of generating optimal solutions, as seen below. Despite such a rudimentary change to the mutation probability and a simple adjustment to the population size, the results turned out to be much better.

Attempt	Population Size	Mutation Probability	Average Search Cost [Generations]	Average Time [seconds]	Percentage of Optimal Solutions
1	50	$\frac{g_i}{(g_0/100)}$	3996.923	43.76054	0.5%
2	100	$\frac{g_i}{(g_0/100)}$	3195.515	81.40092	38.5%

Attempt 1 was over 400 cases and attempt 2 was over 122 cases. g_i is the current generation counter and g_0 is the original generation counter's value, which was set as 4000.

Even so, the optimal percentage of this flavor of the genetic algorithm can hardly compete with the steepest-ascent hill climbing algorithm. Further fine-tunings of the parameters mentioned above would probably further improve the genetic algorithm, but given the scope of this project and the amount of time needed both to find new approaches and to test said approaches, they will be left unexplored in this project. However, in particular, I think my next attempts would be, in no particular order, to increase the population size (again) and maybe even allow some parents to persist then prune according to fitness; to make the crossover split the parents into more than two pieces; to produce two children instead of one; or to sometimes add randomly-generated children to the population throughout the procedure, or maybe only if two parents are too identical, produce a fully-randomly-generated child so we are not wasting resources exploring the same solution.

Appendix

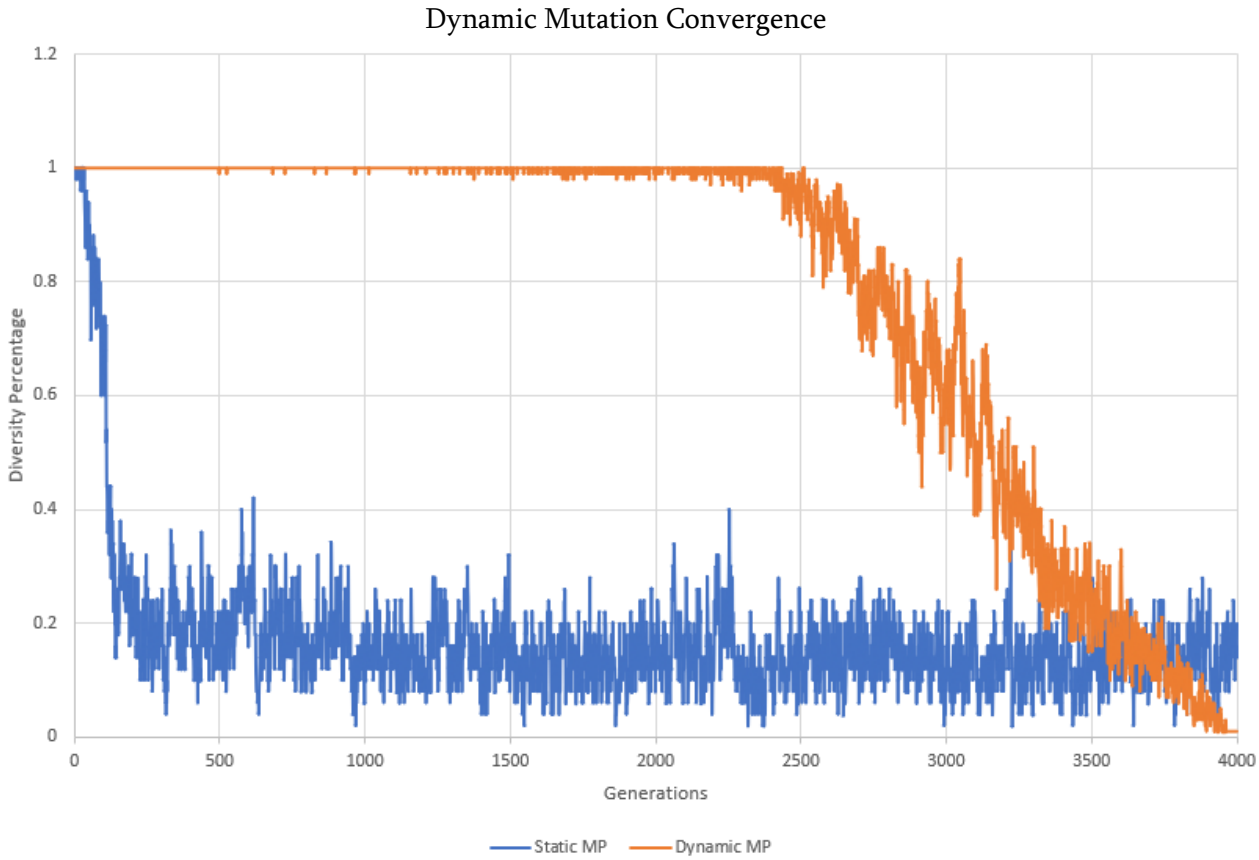
Genetic Algorithm Tables

Below are tables of information gathered when attempting to experiment with different parameters and methods in implementing the genetic algorithm.

Preliminary Parameters Testing

Population Size	Generations	Mutation Probability [%]	Time [mm:ss]	Nonattacking Queens
50	5000	5	00:22	**210**
50	6000	5	01:11	209
50	6000	5	01:12	209
50	5000	5	01:00	209
50	3000	5	00:34	209
100	3000	5	01:10	209
50	5000	5	00:56	209
50	4000	5	00:45	209
50	3000	1	00:34	208
50	3000	10	00:34	208
50	3000	5	00:37	208
100	3000	2	01:14	208
20	10000	5	00:50	208
50	3000	2	00:34	207
50	2000	20	00:24	207
20	3000	5	00:14	206
20	10000	20	00:46	206
50	2000	50	00:22	205
2000	100	5	00:52	204
1000	100	10	00:26	204
1000	100	30	00:26	204
800	1000	5	03:31	203
60	5000	5	01:10	203
40	6000	5	00:56	202
1000	100	5	00:28	202
1000	100	50	00:26	202
1000	100	2	00:26	202
75	5000	5	01:28	201
40	1000	5	00:09	198

Table 1. Testing various population sizes, generations, mutation probabilities, and generations.



Graph 1. One example of the effect of changing the mutation probability mechanism. The percentage refers simply to how much of the population represents a unique state.

$$Diversity\ Percentage = \frac{number\ of\ unique\ strings}{total\ population}$$

You can view two sample populations in the SampleOutputs/GAPopulations directory.