Joshua Camacho

Professor Dominick Atanasio

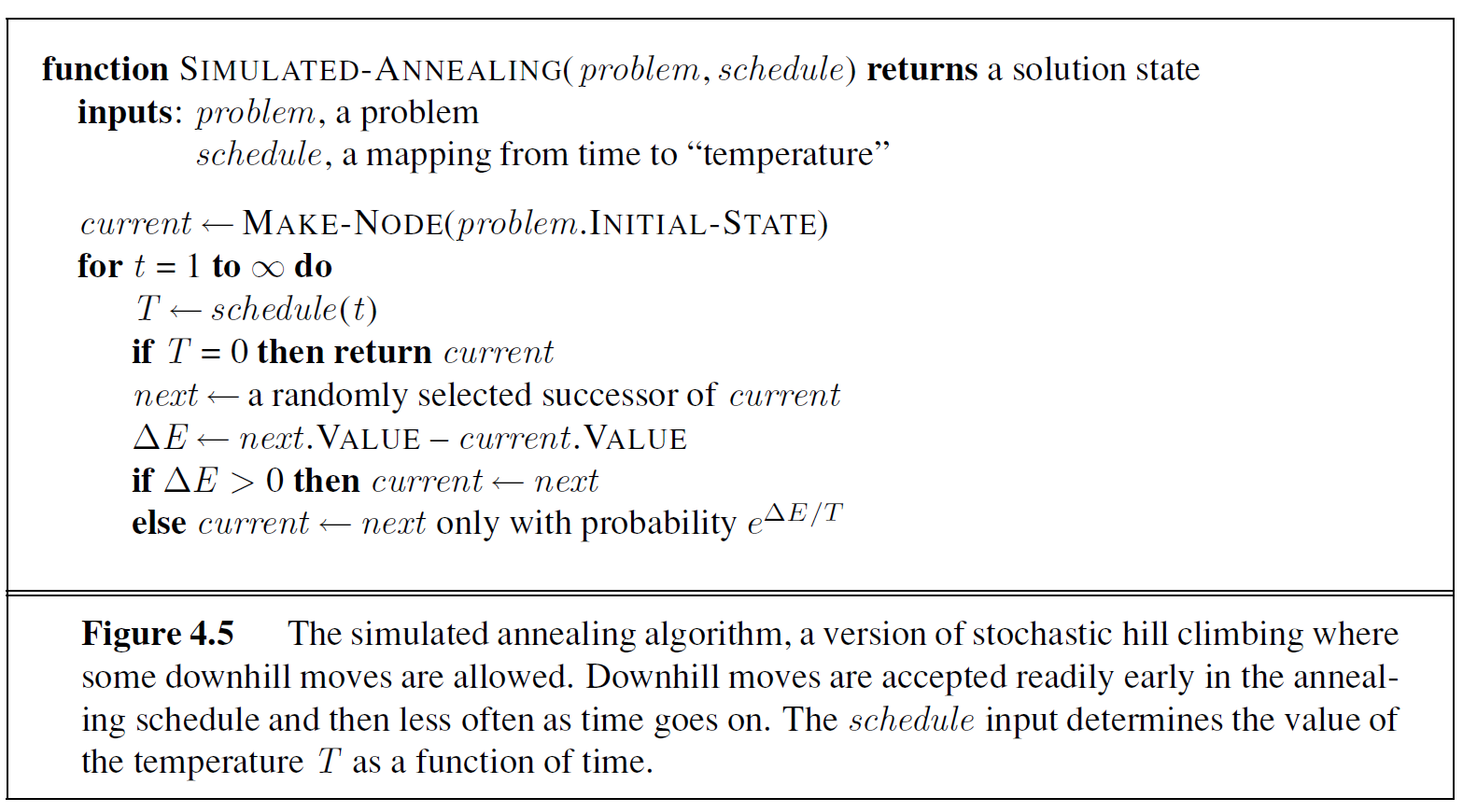
CS420 – Artificial Intelligence

5/6/18

Nqueen With Simulated Annealing and a Genetic Algorithm

For this project I used n=21 for the size of my n-queen board. Two local search methods were used, Simulated Annealing and a Genetic Algorithm.

For simulated annealing I used the algorithm provided by Suart Russel and Peter Norvig in their book, “Artifical Intelligence: A Modern Approach Third Edition”. Psuedocode is as follows:



This algorithm uses a modified version of hill climbing with a random chance to select a non-optimal successor state. The probability of making a non-optimal move depends on time through a scheduling function and the delta between the cost of the current and successor state. The schedule function was what I had the most difficulty with, given its crucial role in the algorithm to determine the probability of choosing a random successor.

Since the schedule function is not clearly defined in the pseudocode provided by the authors, I am under the assumption that it varies based on what specific problem the simulated annealing algorithm is applied to. I had many failed attempts in making a schedule function but stumbled into a decent one with a simple 1/t function as given below:

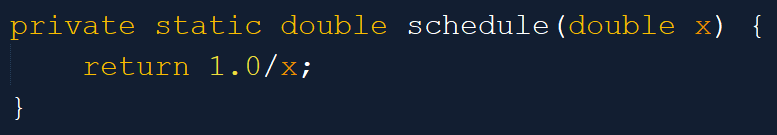


Figure 1: My schedule function

I also modified the annealing process slightly. Instead of entering the inside loop for t=1 to infinity, I made an outer loop which continually restarted the annealing process at t=1 to 1000. Between this an my scheduling function, it produced quite a strange set of probabilities that actually worked quite well for this problem. A graph of my schedule over one iteration:

Figure 2: Probability Mapped by Schedule Function

It simply bounced between a 0% chance to accept a non-optimal node and a 100% chance to accept a non-optimal node, but less frequently accepted towards the tail end of the annealing iteration processes. Note that this cycle would then repeat.

The results of my simulated annealing algorithm were excellent. I consistently solved 100% of randomly generated cases with my annealing algorithm, with average search costs, after 1000 runs, of around 5778 iterations.

My genetic algorithm had more problems than my simulated annealing. With many different unsuccessful iterations, my eventual algorithm was as follows:

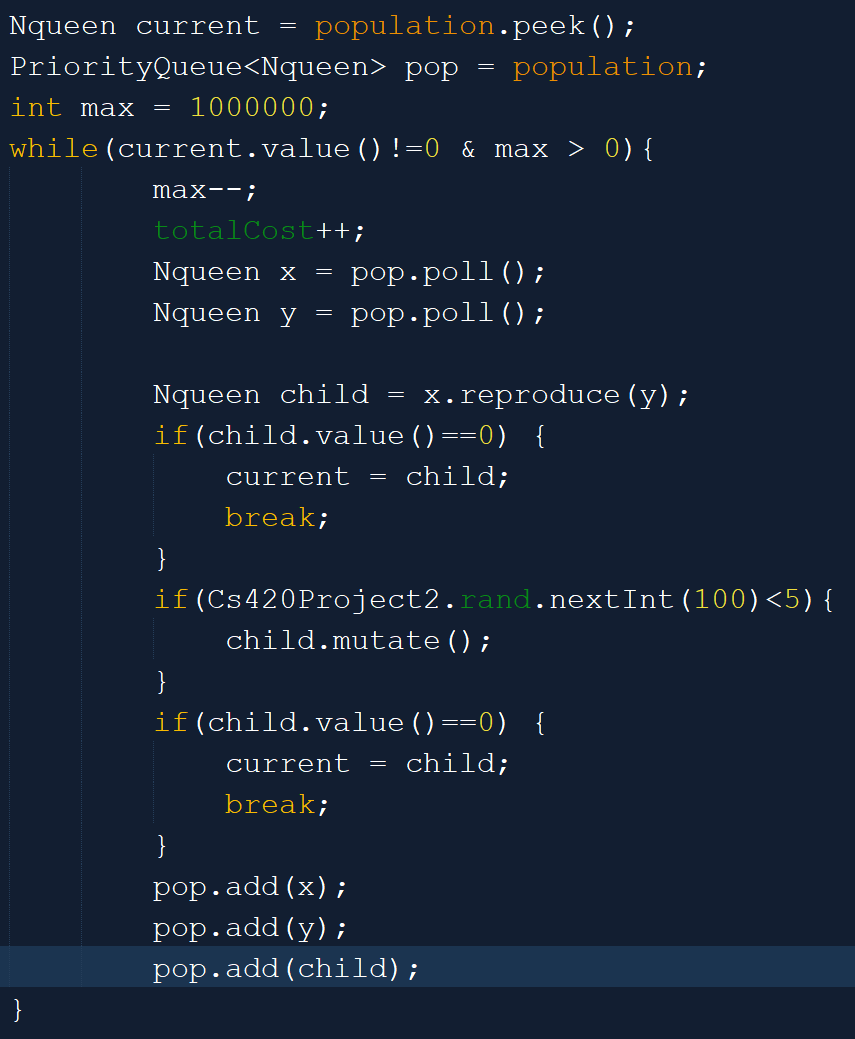


Figure 3: My genetic algorithm

I used a priority queue to easily mate the top individuals in the population. In order to achieve a consistent 93% success rate, I needed to add both parents and the newly created child back to the population. This meant that the population would continuously grow, which was somewhat taxing on RAM usage. In order to prevent the same states from continuously being selected from the priority queue every time, I added a “stale” counter which would increment while the state was in queue and would make it so states of the same value that were in the queue longer have a higher priority. I tested out many mutation rates, but generally higher rates produced worse results. Despite all my efforts, around 6-8% of populations tested would plateau and fail to produce an answer after 1 million iterations with an average iteration for the successful populations of 217538.

Comparing simulated annealing to my genetic algorithm, the simulated annealing was above and beyond the winner.

Figure : Search Cost Comparison

Further optimizations would have to be applied to the genetic algorithm to attempt to increase efficiency.