HW 2

Arnab Dey
Student ID: 5563169
Email: dey00011@umn.edu

State Representation

The full state of the board is represented by a tuple of length equal to number of queens. For N number of queens, index i of the state tuple represents the row position of i^{th} queen. For example, for our 8 queen problem, the state $(1, 0, 3, 6, 7, 4, 1, 5)$ corresponds to a board where the queens are in the following coordinates: $(0, 1), (0, 2), (2, 3), (3, 6), (4, 7), (5, 4), (6, 1), (7, 5)$.

Function Descriptions

Generation of random states

In *utils.py*, I have a function *generate_n_queens_states()* which generates random initial states to start with. It takes two arguments: (1) *n_queens*: number of queens in the problem, which is in our case 8, (2) *n_states*: total number of random states to be generated.

Hill climb steepest ascent

All the search algorithms are implemented in *search.py* file. My *hillclimb_sa()* function takes one argument which is an instance of *Problem* class. The implementation of this class is derived from the book. This function returns a state and the number of iterations to reach that state. Note that, for hill climbing with steepest ascent, the returned state might not always be the solution state, because hill climbing with steepest ascent can get stuck in local maxima (minima).

Hill climb first choice

My *hillclimb_fc()* function takes one argument which is an instance of *Problem* class. The implementation of this class is derived from the book. The generation of a random successor can be done in several ways. I have implemented total three variants to do so. One can randomly pick one of the queens and generate a random move for that queen creating a new state. If this state has fewer conflicts compared to the current state, we make the move to the new state, otherwise we try to generate another random state. We continue generating random successors until we find a better state or hit the maximum number of retrial allowed. I have kept a bound of 200 for the same.

Another way could be to generate all possible successors of the current state and then randomly pick one. I have it written in *get_random_successor_2()* function. It takes the current state as the argument and returns a randomly chosen successors.

Another way could be to shuffle all the queens first and iterate over them. For each queen, we can make random move and retain the first better move. I have it written in *get_random_better_successor()* function. It takes the current state as an argument and returns a better state if there is any. If there is no better state, it returns the current state. Note that, if this implementation is used, it is not required to put a bound on maximum number of trials to find a better successor. This implementation guarantees to find a better successor if there is any.

Simulated Annealing

My *sim_anneal()* function takes one argument which is an instance of *Problem* class. The implementation of this class is derived from the book. The schedule function that I have used is given below:

$$\text{schedule}(t) = Ke^{-\alpha t},$$

where I have taken $K = 20$ and $\alpha = 0.005$. The requirement of schedule function is to reduce the temperature gradually. That is why I have taken a exponentially decreasing schedule function. Note that, ideally this function reaches 0 only at ∞. Therefore, I am using another parameter $\epsilon = 1e^{-4}$ which is being compared with the temperature returned from the schedule function and if the temperature is below ϵ, temperature is considered to be zero. α decides how fast cooling is done. K is decided based on trial and error. Basically, the when we start simulated annealing, K plays an important role. So, we want K to be designed in such a way that when t is small, it generates probability close to 1 if there are large number of queen conflicts on the board. The probability of move is generated by comparing $e^{-\Delta E/T}$ with a sample drawn from uniform distribution between 0 and 1. Here ΔE is a non-negative number denoting difference in number of conflicts between current state and randomly chosen successor state and T denotes the temperature.

Running the Code

main.py file has the necessary code snippets to run the above search algorithms. Each search function described above returns the solution state and number of iterations to reach to that state. We then check if the returned state is a goal state (a state with no conflicts) using *Problem.goal_test()* function which takes a state as argument and returns True if the argument state is a goal state. Please change the variable *n_init_states* in *main.py* to change number of random initial states to test with.

Results

I have generated 1000 random initial states and Table 1 tabulates the results. Accuracy is calculated as follows:

$$\text{Accuracy} = \frac{\text{Number of times a goal state is reached}}{\text{Total number of random initial states}} = \frac{\text{Number of times a goal state is reached}}{1000}.$$

Table 1: Local Search Result Summary

Algorithm	Accuracy(%)	Average Number of Steps
Hill Climb Steepest Ascent	16%	4
Hill Climb First Choice	12.1%	6
Simulated Annealing	90.4%	1052

It can be seen that the accuracy of simulated annealing is much higher compared to steepest ascent and first choice hill climb at the cost of higher number of steps to reach a goal. Hill climbs are relatively faster compared to simulated annealing, however, they suffer from local maxima (minima) which lowers the accuracy. Also, I observed that three different implementations to generate random successors, as described above, results into different accuracy level. It also depends on maximum number of trials allowed to find a better successor (which I have set to 200 in my code). Higher the allowed number of trials, higher would be the accuracy but the search process would be slower.