Artificial Intelligence

Programming Assignment 1 Report

Question 1:

Assumptions: I have denoted the blank space by the number 0 in my puzzle representation. I have assumed that the puzzle whose initial state and goal state is entered by the user is solvable.

a.) BFS

Methodology/Algorithm:

- Create 3 queue (1. queue for states, 2. queue for previous 0 position, 3. queue for current 0 position)

- Push initial state and respective 0 position in above queue

- Keep visited array, initially with initial state

- Until queue is empty, get the first state from queue and perform below task every time:

- Check if the first state from queue matches final_state, Exit if found.

- Check the possible operations from our current 0 position.

- Set the priority of possible operation (Such that the cell that matches final state cell is given low priority and others high)

- For all the possible operations check if the state achieved after the operation do not match already visited state, for all such found states we add such state to queue.

- Continue with our process of until queue is empty.

b.) DFS

Methodology/Algorithm:

- Create 3 stacks (1. stack for states, 2. stack for previous 0 position, 3. stack for current 0 position)

- Push initial state and respective 0 position in above stack

- Keep visited array, initially with initial state

- Until stack is empty, Pop the top state and perform below task every time:

- Check if the popped state matches final_state, Exit if found.

- Check the possible operations from our current 0 position.

- Push back the popped state from stack

- Set the priority of possible operation (Such that the cell that matches final state cell is given low priority and others high)

- For all the possible operations check if the state achieved after the operation do not match already visited state, if we found such state we push such state to stack else we pop the state pushed earlier.

- Continue with our process of until stack is empty.

c.) A*

Methodology/Algorithm:

- Create 4 array (1. array for states, 2. array for previous 0 position, 3. array for current 0 position, 4. array for the heuristics for the states)

- Add initial state, respective 0 position and heuristics of initial state

- Keep visited array, initially with initial state

- Until array is empty, get the state with minimum heuristics from the array and perform below tasks:

- Check if the state with minimum heuristics matches final_state, Exit if found.

- Remove the state with minimum heuristics from the above array.

- Check the possible operations from our current 0 position.

- Set the priority of possible operation (Such that the cell that matches final state cell is given low priority and others high)

- For all the possible operations check if the state achieved after the operation do not match already visited state, for all such found states we add such state to array along with there heuristics.

- Continue with our process of until array is empty.

d.) IDA*

- Set heuristics_threshold as the heuristics of initial state

- Continue the below process for different heuristics threshold set every time:

- Create 4 stacks (1. stack for states, 2. stack for previous 0 position, 3. stack for current 0 position, 4, stack for state heuristics)

- Push initial state and respective 0 position in above stack

- Keep visited array, initially with initial state

- Until stack is empty, Pop the top state and perform below task every time:

- If the popped state heuristics are greater heuristics_threshold (Use this state heuristics and check if it is less next_minimum_heuristics then assign this to next_minimum_heuristics), continue with the process of until stack is empty.

- Check if the popped state matches final_state, Exit if found.

- Check the possible operations from our current 0 position.

- Push back the popped state from stack

- Set the priority of possible operation (Such that the cell that matches final state cell is given low priority and others high)

- For all the possible operations check if the state achieved after the operation do not match already visited state, if we found such state we push such state to stack else we pop the state pushed earlier.

- Continue with our process of until stack is empty.

- After the stack is empty, assign next_minum_heuristics to minimum_heuristics and continue with the process of different heuristics thresholds.

Results/Analysis for Question 1

8 PUZZLE

If I take the initial state of 8 puzzle as :

1 3 4

8 6 2

0 7 5

And the goal state as:

1 2 3

8 0 4

7 6 5

BFS Result

No. of moves = 51

Moves are ['start', 'right', 'up', 'up', 'right', 'right', 'up', 'right', 'up', 'left', 'up', 'right', 'down', 'up', 'right', 'up', 'down', 'right', 'left', 'down', 'up', 'left', 'up', 'up', 'down', 'left', 'right', 'right', 'left', 'right', 'down', 'left', 'left', 'down', 'down', 'right', 'right', 'down', 'up', 'left', 'left', 'left', 'left', 'up', 'up', 'down', 'down', 'down', 'right', 'down', 'left', 'down']

Time taken to find the solution = 0.0069463253021240234 seconds

DFS Result

No. of moves = 6

Moves are ['start', 'right', 'up', 'right', 'up', 'left', 'down']

Time taken to find the solution = 0.0009968280792236328 seconds

A* Result

No. of moves = 6

Moves are ['start', 'right', 'up', 'right', 'up', 'left', 'down']

Time taken to find the solution = 0.0009958744049072266 seconds

IDA* Result

No. of moves = 6

Moves are ['start', 'right', 'up', 'right', 'up', 'left', 'down']

Time taken to find the solution = 0.0039517879486083984 seconds

15 PUZZLE

If I take initial state of a 15 puzzle as:

0 1 2 3

5 6 7 4

9 10 11 8

13 14 15 12

And the final state as:

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0

BFS Result

No. of moves = 95

Moves are ['start', 'right', 'down', 'right', 'down', 'right', 'down', 'right', 'down', 'right', 'down', 'left', 'up', 'right', 'down', 'right', 'down', 'down', 'right', 'down', 'left', 'right', 'up', 'down', 'right', 'down', 'left', 'down', 'up', 'right', 'left', 'right', 'up', 'down', 'right', 'down', 'left', 'right', 'down', 'up', 'right', 'down', 'left', 'down', 'up', 'right', 'down', 'left', 'down', 'left', 'up', 'down', 'up', 'right', 'left', 'right', 'down', 'left', 'right', 'down', 'up', 'right', 'left', 'down', 'up', 'right', 'down', 'right', 'right', 'down', 'down', 'down', 'up', 'right', 'left', 'right', 'down', 'left', 'right', 'down', 'up', 'right', 'left', 'up', 'down', 'right', 'down', 'up', 'right', 'left', 'left', 'up', 'right', 'right', 'up', 'down']

Time taken to find the solution = 0.00897216796875 seconds

DFS Result

No. of moves = 6

Moves are ['start', 'right', 'right', 'right', 'down', 'down', 'down']

Time taken to find the solution = 0.0009968280792236328 seconds

A* Result

No. of moves = 6

Moves are ['start', 'right', 'right', 'right', 'down', 'down', 'down']

Time taken to find the solution = 0.0009641647338867188 seconds

IDA* Result

No. of moves = 6

Moves are ['start', 'right', 'right', 'right', 'down', 'down', 'down']

Time taken to find the solution = 0.0019533634185791016 seconds

24 PUZZLE

Initial state of the puzzle:

1 2 3 4 5

6 7 8 9 10

11 12 13 14 15

16 17 18 19 20

21 22 23 0 24

Final state of the puzzle:

1 2 3 4 5

6 7 8 9 10

11 12 13 14 15

16 17 18 19 20

21 22 23 24 0

BFS Result:

No. of moves = 1

Moves are ['start', 'right']

DFS Result:

No. of moves = 1

Moves are ['start', 'right']

A* Result:

No. of moves = 1

Moves are ['start', 'right']

IDA* Result:

No. of moves = 1

Moves are ['start', 'right']

Analysis: We can see that A* and IDA* always give the optimal solution i.e in minimum no. of moves.

While DFS gives the solution nearest to the root.

BFS explores more no. of states.

Question 2:

Assumptions:

I have assumed the population size to be 100. I have used the no. of attacking pairs as the fitness value. So, for 8- queen the fitness value of a chromosome varies from 0 to 28. So, the best individual in the population will have its fitness value as 0.

I have set the limit for no. of generations for which my program runs as 10000. So, if the algorithm does not find the fittest individual in 10000 generations then it prints no solution found.

Methodology/Algorithm:

- I generate the initial population size of 100 and the population has genes selected randomly in the range 1 to 8 for n-queen and 1 to N for N-Queen.
- Now I calculate the fitness value of each of these chromosomes.
- Now I sort the chromosomes in the ascending order on the basis of fitness value of the chromosome.
- Now I pick the top 80 parents and do crossover among them by randomly selecting 2 parents from the pool of 80 parents and do cross-over to generate 100 chromosomes from them
- Now, we have 100 chromosomes.
- Check the whether any of the chromosomes has a fitness value equal to 0. If found any such chromosome then exit and print the chromosome as it is the solution to the N-queen problem and also print the generation number. Also, I plot a graph containing generation numbers on the X-axis and the minimum fitness value for that generation on Y-axis.
- If no chromosome generated after crossover has a fitness value equal to 0 we proceed to mutation. The way to do mutation is:
- For each of the chromosomes, we generate a random number 0 or 1.

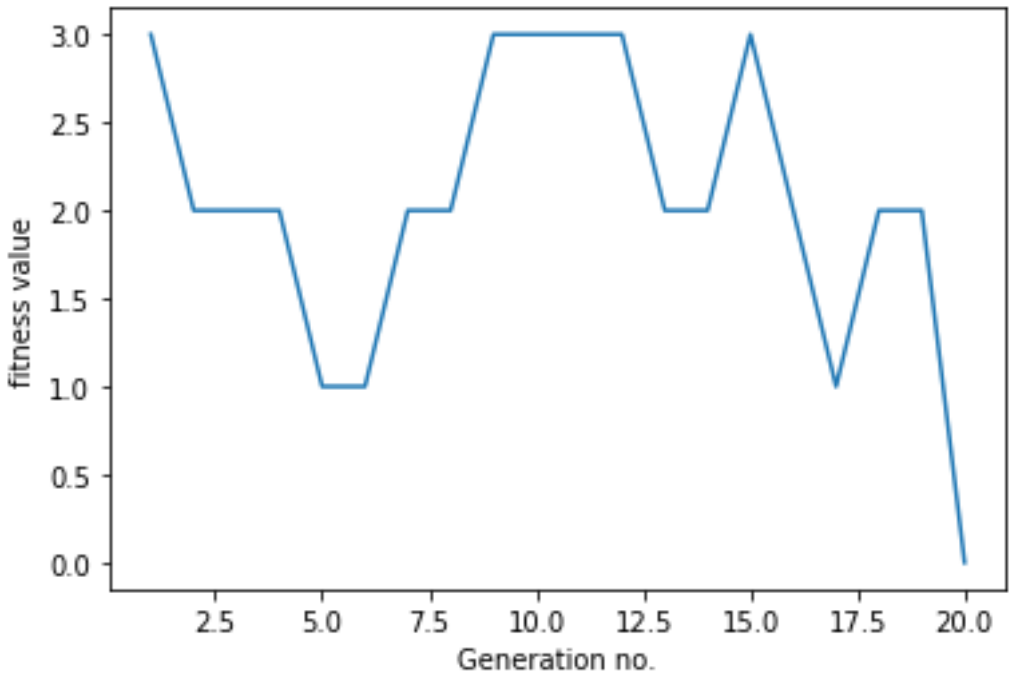
- If the number is 1 then we first randomly select a column number i.e from 1 to N of the chromosome and then change the value stored at that column to some random value in the range 1 to N.
- **So, actually what we are doing is we are placing the queen at in some random column at some random row with a small random probability**
- Now, check if any of the chromosomes generated after mutation have a fitness value equal to 0. If true then exit and print the chromosome as well as the generation no.
- Else repeat all the above steps till an individual with fitness value equal to 0 is found or the generation count reaches 10000.

Results/Analysis

For 8 queen

I plotted a graph for 8 queens keeping the generation no. on the X-axis and the fitness value on the Y-axis:

The graph I get is:



Here I got the output as:

1 7 5 8 2 4 6 3

Generation = 20

So, here we get the individual with fitness value equal to 0 at generation number 20.

From the graph we can analyse that in the starting i.e generation 1 we have fitness value equal to 3. The fitness value fluctuates continuously and then finally becomes 0 at generation number 20. Now no pair of queens on the 8x8 chess board are attacking each other.

Here I got the output as : 1 7 5 8 2 4 6 3

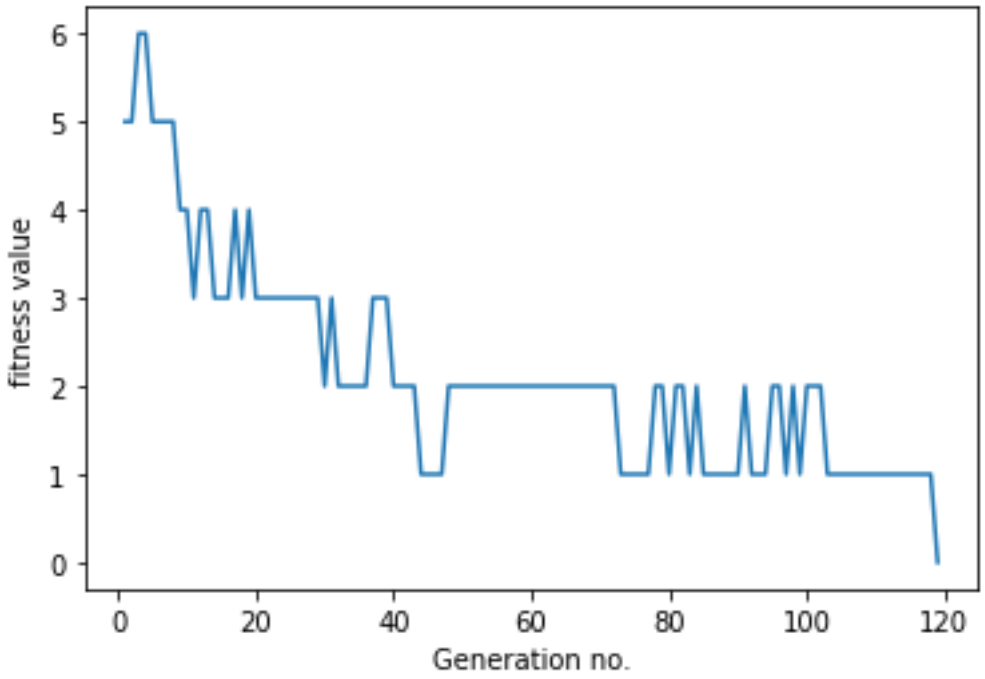
Here we have 8 values. This represents the row no. of each of the columns of the chess board where we need to place the queens. Here the row numbers are counted from the bottom of the chess board.

For N queens:

I plotted a graph for N queen keeping the generation no. on the X-axis and the the fitness value on the Y-axis:

Here I kept the value of N as 12

The graph I got is:



Here I got the output as :

12

2 9 6 8 3 1 12 10 5 7 4 11

Generation = 119

So, here we get the individual with fitness value equal to 0 at generation number 119.

From the graph we can analyse that in the starting i.e generation 1 we have fitness value equal to 5. The fitness value fluctuates continuously and then finally becomes 0 at generation number 119. Now no pair of queen on the 12x12 chess board are attacking each other.

Here I got the output as : 2 9 6 8 3 1 12 10 5 7 4 11

Here we have 12 values. This represents the row no. of each of the columns of the chess board where we need to place the queens. Here the row numbers are counted from the bottom of the chess board.

So, applying genetic algorithm we have solved the N-queen problem.