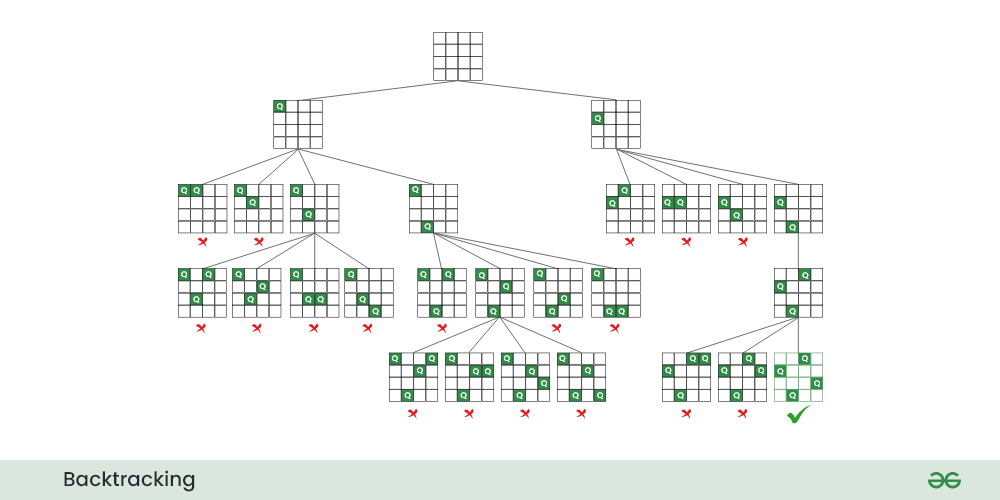
**N-Queens Problem**



1)Description

The N-Queens problem is a classic combinatorial problem that dates back to the 19th century. The problem is defined as follows:

**Problem Statement**: Place N chess queens on an N×N chessboard in such a way that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The N-Queens problem is a well-known problem in the field of computer science, artificial intelligence, and combinatorial optimization. It's often used as a benchmark for testing algorithms and problem-solving strategies.

The N-Queens problem is a classic example of an NP-complete problem. Its complexity grows rapidly with an increase in N, making it computationally challenging for larger board sizes.

Various algorithms have been proposed to solve the N-Queens problem. Common approaches include backtracking, constraint satisfaction problems (CSP), genetic algorithms, and more..

**Forward Checking**:

Time Complexity: O(N!)

Space Complexity: O(N)

Forward checking involves recursively searching through the solution space while considering constraints. The time complexity is exponential, mainly because the algorithm explores all possible configurations. The space complexity is linear, representing the depth of the recursion.

**Genetic Algorithm**:

Time Complexity: It's challenging to provide a precise Big O notation due to the stochastic nature of genetic algorithms.

Space Complexity: O(N \* population\_size)

**Reinforcement Learning(My principal purpose of the project)**

The reinforcement learning algorithms I used have a random element and can be tricky to control in different situations.

After running many experiments and tweaking various settings, the time it takes to find a solution varies a lot. The randomness in these algorithms, combined with how the agent interacts with the environment, makes it hard to predict exactly how long they'll take.

The results depend a lot on how complex the learning situation is and how well I adjust the settings. In practical terms, it's important to experiment and adjust the settings to get the best performance on specific problems.

2)Results & Analysis

As my initial intent was to create a neural network that takes as input a vector of target values and provides the reinforcement learning agent with Q-values.

These Q-values consist of n values, each representing the expected value for placing a queen in one of the n possible squares. The agent selects the maximum Q-value to determine the corresponding spot to place the queen.

The target values were obtained from a combination of the rewards given to the agent for the action he tried to do (so placing the queen in a spot) and the prediction of the model itself.

The neural network I used was the following:

model = keras.Sequential([

    keras.layers.Dense(16, activation='relu', input\_shape=(n\_queens,)),

    keras.layers.Dense(4, activation='softmax')

])

model.compile(optimizer='adam', loss='mse')

As visible, the metric used for the loss, is the MSE.

I encountered several difficulties in combining the predictions of the neural network with the actions I had to make the agent take. Finding the right rewards to feed into the neural network's input, to encourage the agent to interpret them correctly, was challenging from the very beginning.

I didn't want to force the agent by adding an 'if' condition, thus removing the possibility for it to place the queen in a column or row previously used for other queens. This goes against the principles of reinforcement learning. However, the issue was that the neural network, trained on one sample at a time (the target vector), couldn't adapt quickly enough, not keeping up with the decisions made by the agent. Online training was not the only option, but preserving various target value vectors to use micro-batches wasn't a very sensible choice, considering that in the meantime, the agent could only take random actions. Below, I illustrate how I struggled to make the agent receive rewards that would lead the neural network to produce suitable q\_values for the next actions.

(max): 0

state:[3 0 2 0]

queens placed:4

I tried to assign action 0 to row 3 but it failed the admissibility test

REWARD:0

COUNTER:77

1/1 [==============================] - 0s 17ms/step

q\_values:        [ 1.6873366   0.49810103 -0.34256077  0.12105239]

(max): 0

state:[3 0 2 0]

queens placed:4

I tried to assign action 0 to row 3 but it failed the admissibility test

REWARD:0

COUNTER:78

1/1 [==============================] - 0s 17ms/step

q\_values:        [ 1.6873166   0.49810103 -0.34256017  0.12105139]

(max): 0

state:[3 0 2 0]

queens placed:4

I tried to assign action 0 to row 3 but it failed the admissibility test

REWARD:0

COUNTER:79

1/1 [==============================] - 0s 17ms/step

q\_values:        [ 1.6873066   0.49910103 -0.34256007  0.12105939]

[from version\_5.py]

As the counter indicates, these are the 77th, 78th, and 79th times that the agent tries to place the last queen in column 0 of the last row.

The current state is [3, 0, 2, x], and the agent is attempting to assign the last queen to column 0. However, in this position, the queen will be threatened by the queen of the second row, making this spot inadmissible.

The Q-value for the 0th column keeps decreasing, but with a very small factor.

Meanwhile, the neural network is not able to properly increase the other Q-values

At this point, after a week of unsuccesable tries, I decided to not use anymore the Neural Network.

The following results are for the problem of 4,5,6,7,8- queens, with the code “v\_13\_RL\_emp.py”, based on Reinforcement Learning agent, with the function get\_reward just shown previously.

It is important to emphasize that the agent's performance can be highly variable. In each script (I have just under a dozen codes, each representing different versions of the script that I have modified over time), the agent can make decisions either randomly or based on the Q-values (the maximum of the weights vector).

Specifically, the agent may opt for a random decision if, upon generating a random number using the NumPy library (using the rand method from the random module), this number is not greater than a certain epsilon threshold. This epsilon threshold is set to a very low value as I want the agent to learn how to navigate the environment.

In the following pictures I show the final state, the number of steps required and the average distant between the queens placed for each step.

4-Queens:

Average Distance of the current state: 2.54

state:[1 3 0 2]

queens placed:4

Final state:

. Q . .

. . . Q

Q . . .

. . Q .

Steps required: 45

Final Average Distance(average between each step): 0.57

Elapsed Time: 0.030826807022094727 seconds

5-Queens:

Average Distance of the current state: 3.01

state:[0 2 4 1 3]

queens placed:5

Final state:

Q . . . .

. . Q . .

. . . . Q

. Q . . .

. . . Q .

Steps required: 13

Final Average Distance(average between each step): 2.06

Elapsed Time: 0.010783672332763672 seconds

6-Queens:

Average Distance of the current state: 3.58

state:[2 5 1 4 0 3]

queens placed:6

Final state:

. . Q . . .

. . . . . Q

. Q . . . .

. . . . Q .

Q . . . . .

. . . Q . .

Steps required: 230

Final Average Distance(average between each step): 1.31

Elapsed Time: 0.20709943771362305 seconds

7-Queens:

Average Distance of the current state: 4.10

state:[1 6 4 2 0 5 3]

queens placed:7

Final state:

. Q . . . . .

. . . . . . Q

. . . . Q . .

. . Q . . . .

Q . . . . . .

. . . . . Q .

. . . Q . . .

Steps required: 103

Final Average Distance(average between each step): 1.65

Elapsed Time: 0.1057744026184082 seconds

8-Queens:

Average Distance of the current state: 4.58

state:[3 1 6 2 5 7 4 0]

queens placed:8

Final state:

. . . Q . . . .

. Q . . . . . .

. . . . . . Q .

. . Q . . . . .

. . . . . Q . .

. . . . . . . Q

. . . . Q . . .

Q . . . . . . .

Steps required: 1582

Final Average Distance(average between each step): 1.60

Elapsed Time: 1.7892706394195557 seconds

For providing a temporal comparison between the implementation of the final resolution using reinforcement learning and other types of solutions, I have also implemented algorithms for Forward Checking and Genetic Algorithm.

Below, I present the results in terms of time:

note:

RL= Last Reinforcement learning version, FC= Forward Checking, GA= Genetic Algorithm

4-size:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **RL** | **FC** | **GA** |
| Time(s) | 0,02362 | 0,00021 | 1,38270 |
| Avg Dist | 2,33 | 1,67 | 1,67 |

5-size:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **RL** | **FC** | **GA** |
| Time(s) | 0,02135 | 0,00025 | 1,19760 |
| Avg Dist | 2,23 | 2 | 2 |

6-size:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **RL** | **FC** | **GA** |
| Time(s) | 0,251 | 0,00045 | 6,35615 |
| Avg Dist | 2,87 | 2,33 | 2,33 |

7-size:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **RL** | **FC** | **GA** |
| Time(s) | 0,170 | 0,00035 | 5,7019 |
| Avg Dist | 3,35 | 2,67 | 2,67 |

8-size:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **RL** | **FC** | **GA** |
| Time(s) | 0,2680 | 0,0015 | 82,939 |
| Avg Dist | 3.67 | 3 | 3 |

\*\*The execution time for the 8-size queen problem for the GA is not considered in the graph, because of its extraordinary (in negative way) deviation from all the other times

As we can see, Forward Checking is by far the most effective solving method. However, my reinforcement learning prototype is not as slow, especially when compared to the genetic algorithm.

Of course, I was aware that for a problem of this nature, using reinforcement learning as a resolution strategy would be excessively and unnecessarily complex. It has been a rewarding experience for me to engage in the initial and intriguing execution of an agent, tackling a rather complex environment such as the n-queens problem.

The other metric I utilized is the average distance between queens at each step of the agent simulation. This metric was employed to impose a penalty on the agent when attempting to place a queen in a non-admissible spot.

As we can see in the following graph, GA (Genetic Algorithm) and FC (Fitness Function) consistently exhibit the same values for the queens' distance, whereas my agent tends to place queens in a manner that maximizes their separation. This tendency is, of course, influenced by the rewards assigned thans to the distance itself.

\*\*The graph of GA and FC are the same, there is overlapping on the curves

3)Flow Chart

As mentioned earlier, my work on reinforcement learning had to scale down, requiring the removal of the neural network as the provider of Q-values for the agent. Below, I present two flowcharts, one depicting the initial idea, and the other representing the final structure of my code.

The following is the flowchart for the initial version (v3.py). It represents one episode of the agent simulation. The simulation consists of a #number\_of\_episodes (hyperparameter). When the agent finishes the episode (finding a valid solution), there is a backpropagation phase in which the model is trained on the experience just gained.

Immagine che contiene testo, diagramma, schermata, Parallelo

Descrizione generata automaticamente

Then, eliminating the NN model, this is the flowchart for the final version(v\_13\_RL\_emp.py)

Immagine che contiene testo, diagramma, schermata, Parallelo

Descrizione generata automaticamente

In this final version:

Initially the current state is given by:

# Function to generate initial state

def generate\_initial\_state():

    return np.full((n\_queens,), -1)

The new state is the state + action only if this lead to a valid state(admissable state). And this is verified by:

def is\_valid\_solution(state):

    queens\_placed = np.count\_nonzero(state != -1)

    print(f"state:{state}")

    print(f"queens placed:{queens\_placed}")

    if queens\_placed == 1:

        return True

    if queens\_placed == 2:

        if state[0] == state[1] or abs(state[0] - state[1]) == abs(0 - 1):

            return False

        else:

            return True

    else:

        for i in range(queens\_placed):

            for j in range(i + 1, queens\_placed):

                if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

                    return False

        return True

If is\_valid\_solution return false: New\_state=Current\_state

The agent takes the rewards from the following defintion of the method get\_reward:

def get\_reward(state,prec\_state,current\_average\_distance):

    queens\_placed = np.count\_nonzero(state != -1)

    queens\_placed\_before=np.count\_nonzero(prec\_state != -1)

    if queens\_placed == n\_queens and is\_valid\_solution(state):

        return 1.0

    else:# queens\_placed-queens\_placed\_before>0:

        return 0.05\*current\_average\_distance

The current\_average\_distance is the average distance of the queens placed in the current state.

Average distance measured as it follows:

def calculate\_average\_distance(state):

    queens\_placed = np.count\_nonzero(state != -1)

    if queens\_placed < 2:

        return 0.0  #Not enough quenns placed

    total\_distance = 0

    pair\_count = 0

    for i in range(queens\_placed):

        for j in range(i + 1, queens\_placed):

            col\_diff = abs(state[i] - state[j])

            row\_diff = abs(i - j)

            total\_distance += np.sqrt(col\_diff\*\*2 + row\_diff\*\*2)

            pair\_count += 1

    return total\_distance / pair\_count

The q\_values are calculated as it follows.

 q\_values[action]-=reward

Only the index corresponding to the action that the agent took or intended to take is updated in the Q-values vector).

4)Code

Instead of attaching all the codes I made (I used several files), I have created a Google Drive folder where you can find every script I used. The folder is open to anyone with the link, so you should not encounter any issues accessing it. Below is the link.

https://drive.google.com/drive/folders/184jOKp\_6Ha5G5zFfO6r-nXyf4WmcH4FD?usp=drive\_link