The N-Queens Problem

Henry Torres - htorres15@csu.fullerton.edu
Ricky Martinez - rickym72@csu.fullerton.edu
Hammad Qureshi - qureshi434@csu.fullerton.edu

CPSC 481-05
Spring 2021
Tues./Thurs. (2:30pm-3:45pm)

Problem Statement:

The problem that we are trying to solve is the N Queens Puzzle. The user picks a number which will be represented as N and is to place N chess queens on an N×N chessboard so that no two queens threaten each other; i.e, a placement where no two queens share the same row, column, or diagonal.

The game is simple and uses queens from a chess board. Example, if the user picks N = 8, this would be 8 queens and every queen can move vertically, horizontally, and diagonally. Thus, no 2 pairs can conflict with each other. In addition, picking 8 queena would also create an 8x8 chess board as seen below.

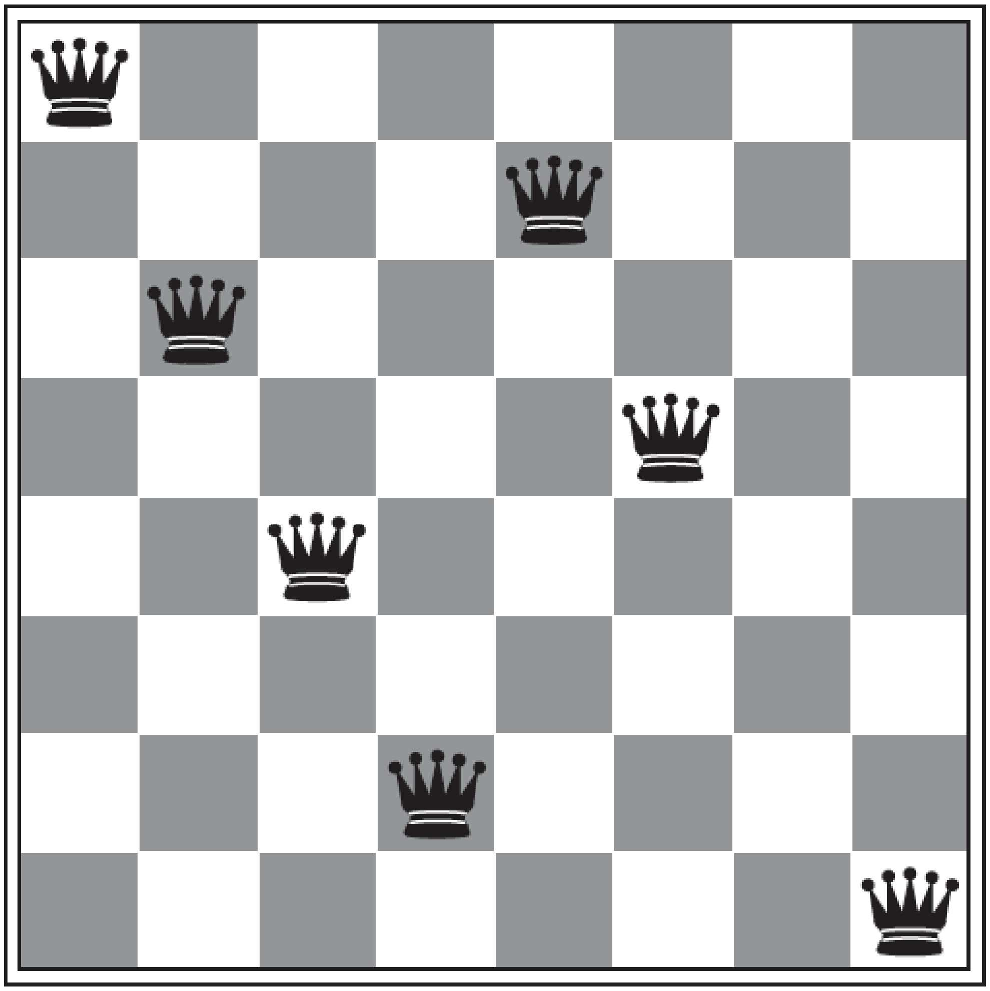


Figure 3.5 (“A.I.M.A.”, Textbook(pg. 72))

Approach:

The general approach of the N Queens Puzzle is implementing the genetic algorithm which was provided by Dr.Panangadan and the textbook "Artificial Intelligence: A Modern Approach".

Description of the software:

The components of the software consists of 4 files: N-Queens.py, search.py, timer.py, and queenGame.py. The language the program was written in was Python 3.9. The libraries used were time, random, pygame, and sys.

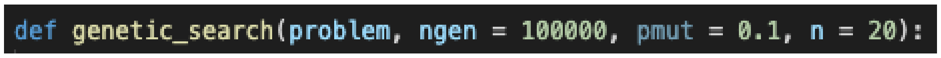
N-Queens.py: is the file that contains the main driver which will in turn call the search.py and timer.py file.

Search.py: contains the main algorithm that we will be using. This file also contains the class NQueensProblem(Problem) which will help initialize the input of the user. It is then followed by the function genetic_search() which creates the proper variables for the genetic_algorithm() ro run correctly.

Timer.py: this file contains an open source timer which counts the time program takes to run, this is measured in seconds.

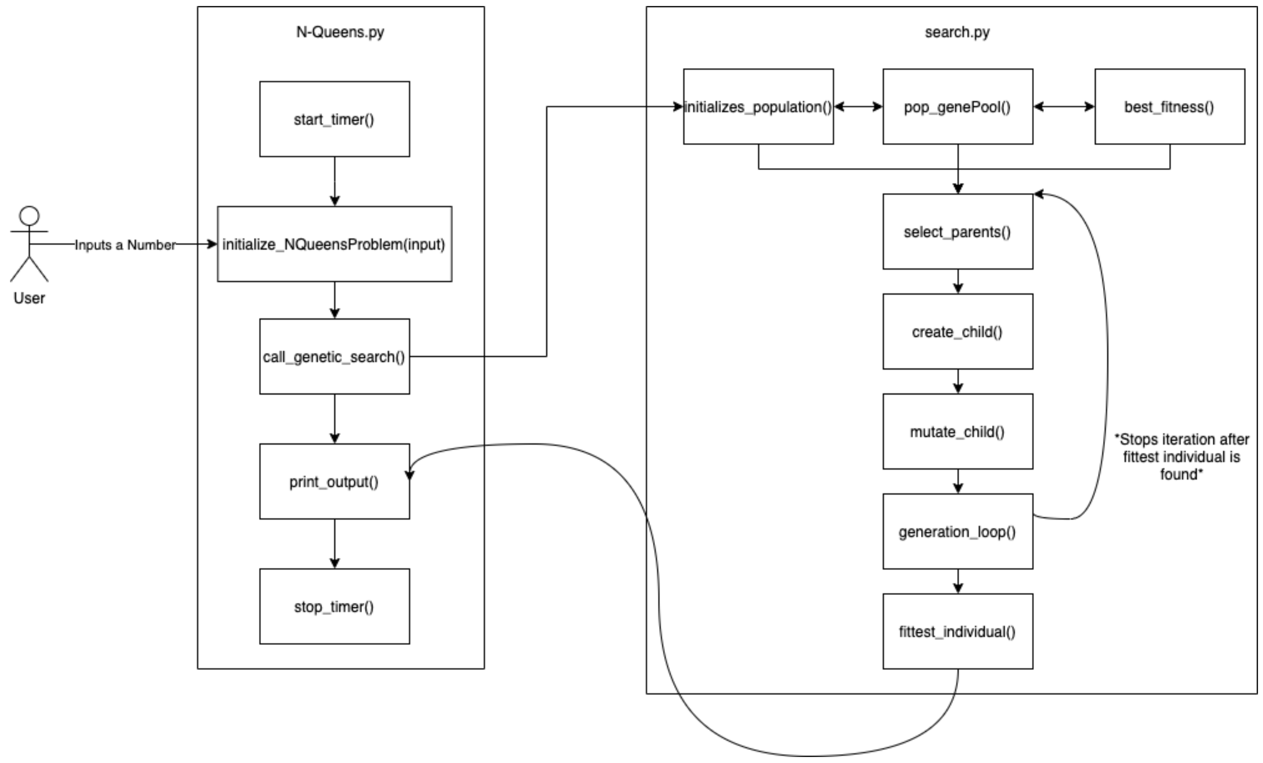
queenGame.py: this file contains the GUI for the Nqueens problem which also implements the search.py file and the genetic algorithm.

The purpose of the main driver is to initially communicate with the user and ask for an input, specifically an integer. This input will then be initialized in theNQueensProblem(Problem) located in the search.py file. Followed by the initialization of the genetic_search() which is also located in search.py. Next, genetic_search() should create the correct variable which will return the correct states, problem, gene_pool, fitness_thresh, and next_generation. From here the code will be executed calling the genetic_algorithm() which in turn will pick the best parents based on fitness and will combine those 2 parents and produce a child state and possibly mutate that child based on the percentage of pmut. An example where ngen is the amount of generations, pmut is the percentage of mutation. While n is the amount of states in a population.



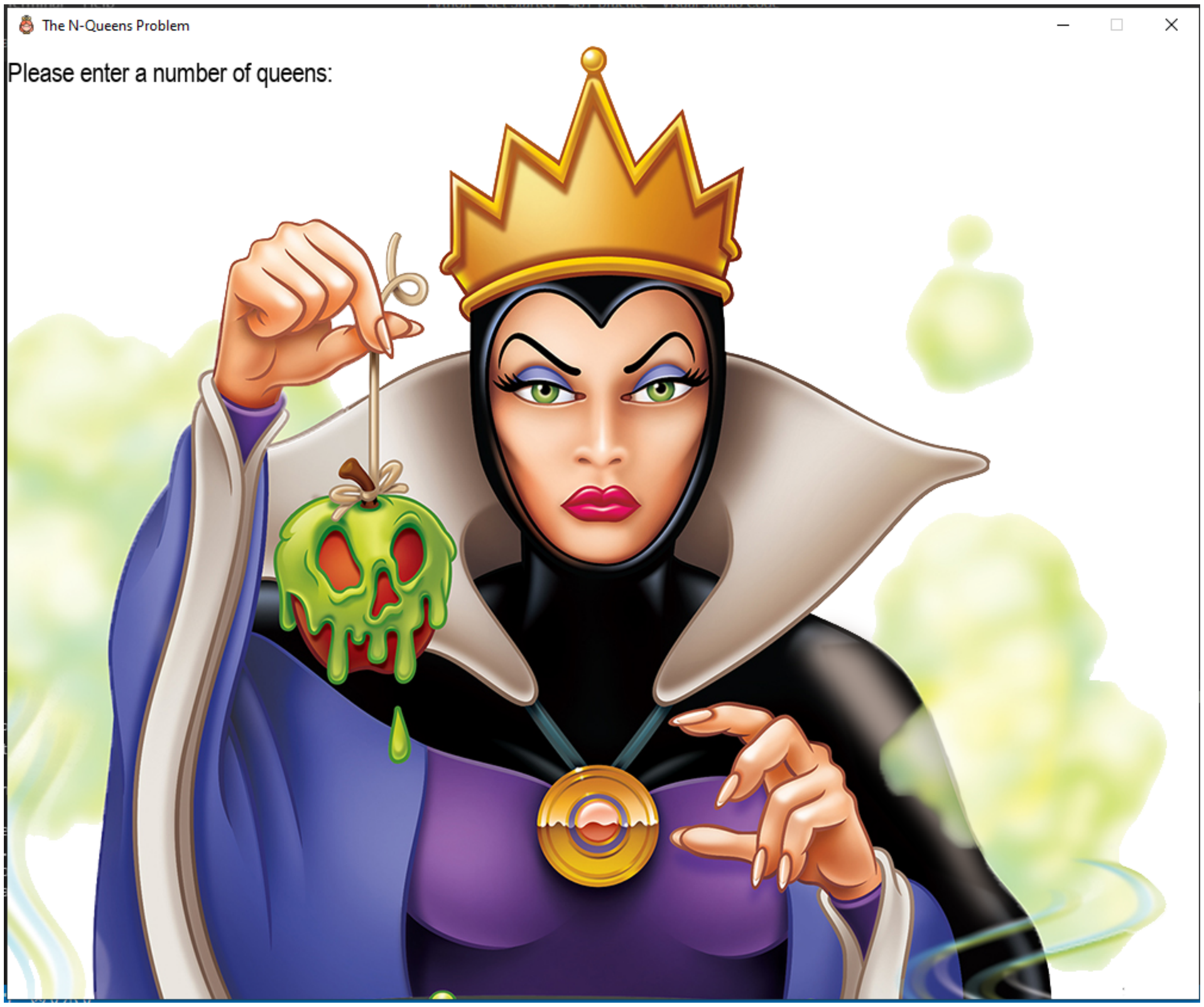
```python
def genetic_search(problem, ngen = 100000, pmut = 0.1, n = 20):
```

There is a diagram below which will show how this code is computed.



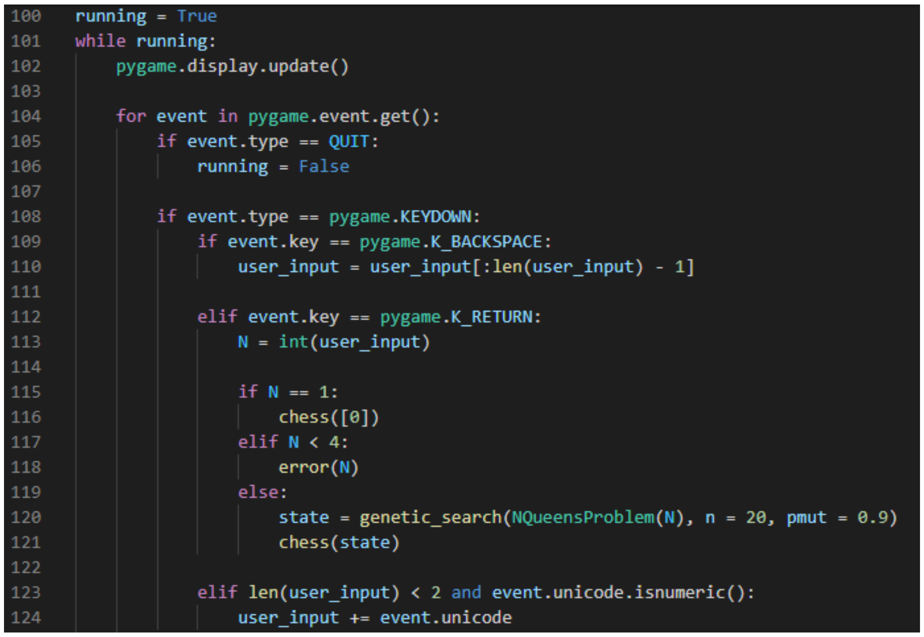
The queenGame.py is the GUI which is ideally the same thing as our driver and will call the search.py file which in turn will call the genetic_algorithm() the same way as N-Queens.py. The difference is we implemented images, texts, and shapes into the GUI to reproduce a chess board.

The start of the GUI will ask you to "Please enter a number of queens: " here the user will input a number, if 1 is chosen it will always return 0. If 2 or 3 is chosen it will display a no solution.

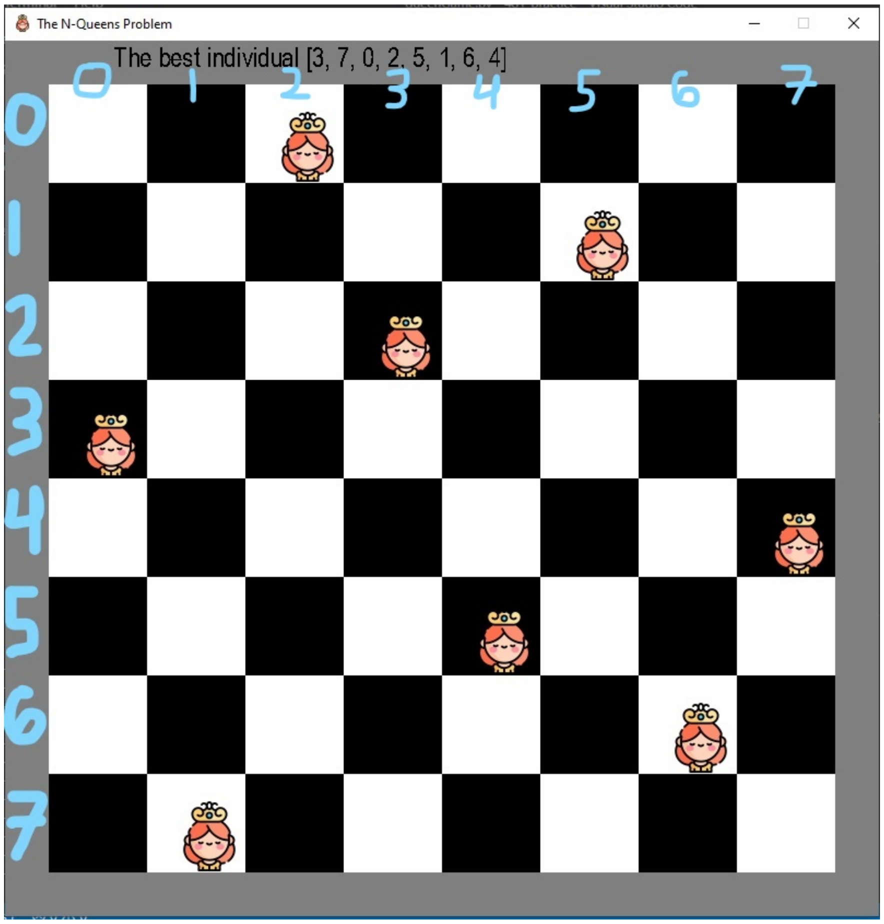


Please enter a number of queens:

In the image below the chessboard is displayed by calling a while loop and drawing a shape multiple times depending on the number of queens implemented.



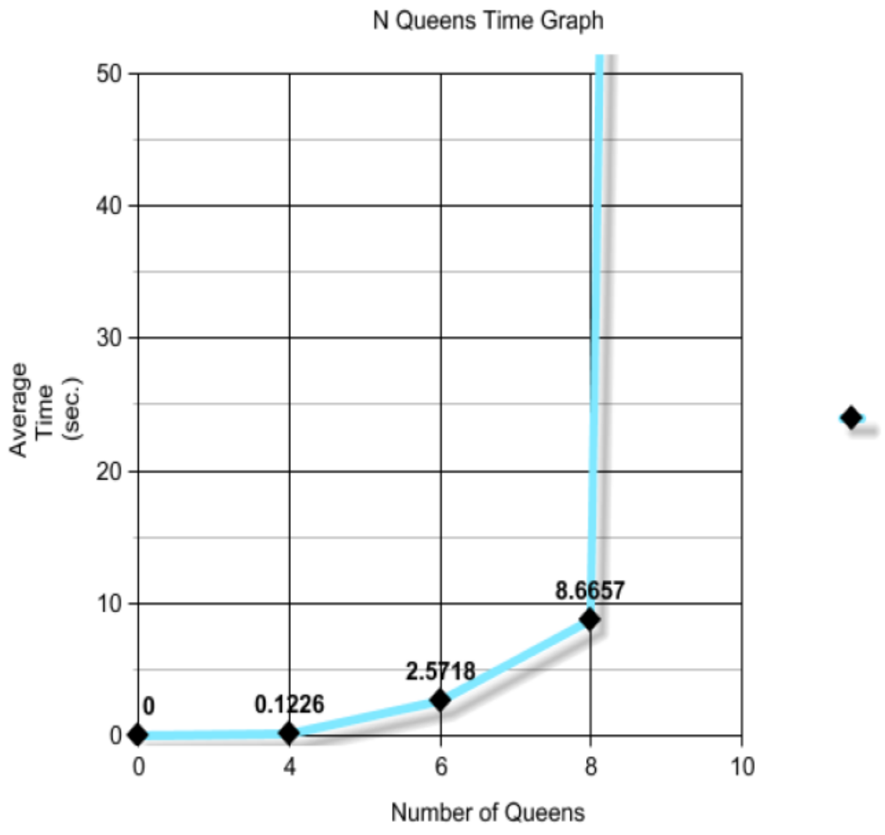
```python
running = True
while running:
    pygame.display.update()

    for event in pygame.event.get():
        if event.type == QUIT:
            running = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_BACKSPACE:
                user_input = user_input[:len(user_input) - 1]

            elif event.key == pygame.K_RETURN:
                N = int(user_input)

                if N == 1:
                    chess([0])
                elif N < 4:
                    error(N)
                else:
                    state = genetic_search(NQueensProblem(N), n = 20, pmut = 0.9)
                    chess(state)

            elif len(user_input) < 2 and event.unicode.isnumeric():
                user_input += event.unicode
```



```
The best individual [3, 7, 0, 2, 5, 1, 6, 4]
```

Furthermore, in pygame the display is initiated on the top left of the screen which means the state [3, 7, 0, 2, 5, 1, 6, 4] starts on the top left. 0 is the top left most square then followed by 1,2,3,4,5,6,7.

Evaluation: In respect to the percentage of proper solutions we need to look at the parameters passed. For queens passed in as N=8 or below we are able to produce a solution 100% of the time. However, when we begin to increase the queens to N= 8-10 our solution is only accurate 90% of the time and drops thereafter.



	0	4	6	8	10
	0 (0%)	0.12 (0%)	2.57 (0%)	8.67 (1%)	720.96 (100%)

In the image above we calculated the average time it would take to run the algorithm with 0, 4, 6, 8, and 10 queens. We tested the algorithm on average 8 times to get a reasonable result. We concluded anything higher than 8 queens would exceed the amount of time we wanted it to be efficient. At 10 queens it would sometimes find the solutions or it would exceed the amount of generations.

However, since we had multiple solutions(brute force) and the current optimization, we are able to see there was a drastic increase in speed even with lower proper working number pairs. In all, we are extremely content with solutions produced and the number of generations required to give us this.

Conclusion:

The N-Queens problem has become a staple for all undergraduate students and even those interviewing for a job. Although there is a multitude of ways to solve this problem type, we were able to take the given algorithm and utilities to finish the task. It can be seen with the implementation of the genetic algorithm coupled by mutation we can see how quickly or slowly generations begin to populate by the given queen size inputted. Since we were testing the way in which we could create non attacking pairs of queens the only user interaction with the AI is the population size of the queen. We were able to produce a solution with a brute force manner and re-evaluate our work to begin to optimize our space time complexity to produce fewer generations and return a solution. With a much more generous time frame we would be able to increase our queen size input and it's proper solution. In conclusion, this really shined light for us on a new set of problems and how they could be solved.

References:

- Pygame video:
  - https://www.youtube.com/watch?v=FfWpgLFMI7w&t=1838s
- Pygame:
  - https://www.pygame.org/docs/
- Search.py:
  - https://github.com/aimacode/aima-python/blob/master/search.py#L908
- Wikipedia:

    - https://en.wikipedia.org/wiki/Eight_queens_puzzle
- GA Video:
    - https://www.youtube.com/watch?v=qt85_CinKwo&t=4s
- Images:
    - https://www.flaticon.com/
- Timer.py:
    - https://realpython.com/python-timer/