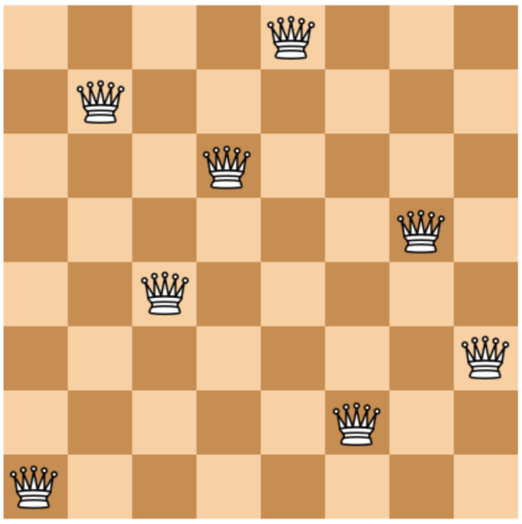
APL771 Project
N-Queens Problem using Genetic Algorithm

Parth Patel (2021CS10550)

November 15, 2023

§1 Problem statement

• Given an NxN chessboard, the goal is to find a placement of N queens such that there is no attacking pair of queens.
• The problem is a satisfaction problem which can be converted to an optimization problem by defining suitable fitness function.
• Note that the problem is NP-hard. Solving the problem using brute-force methods would take time which is exponential in N. We will use the genetic algorithm for optimization. Since genetic algorithm requires a fitness function which is to be maximized, we would appropriately define a fitness function.



§2 Program Logic

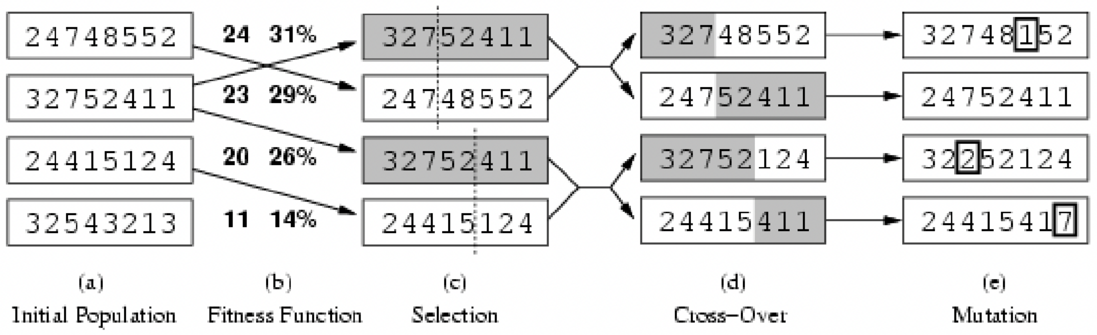
- The state of the chessboard is defined by N design variables. Each design variable represents a row, and the value of that variable denotes the column number on which the queen is placed.

- Note that the assumption that there will be exactly on queen in each row, since if it is not then one row will have two queens which will be attacking each other and hence that placement of queens on the chessboard would never be a solution to the N-queens problem.

- The fitness function for a given complete placement of N queens on the chessboard is the number of non-attacking pairs of queens. We want the board placement where the number of non-attacking pairs of queens is maximum i.e. $\binom{N}{2}$

§3 Genetic Algorithm Implementation

We will use string of N numbers to represent one design variable. Although the genetic algorithm discussed in class required the population to be binary numbers, we can always "club" the bits to get a natural number which wont change. So just using a string of bits in this case wont suffice and would lead to complications like representing a state of chessboard which never exists (column index out of bounds). Overview of the algorithm (with a sample example) -



4 states for 8-queens problem

2 pairs of 2 states randomly selected based on fitness. Random crossover points selected

New states after crossover

Random mutation applied

§3.1 Initial Population and first iteration

- We first initialize an initial random sample population of size `population_size` for N=`board_size`.

```
    // initializing initial population
    vector<vector<int> > initial_population;
    for (int _=0; _<population_size; _++){
        vector<int> board_sample;
        for (int _=0; _<board_size; _++){
            board_sample.push_back((rand()%board_size) + 1);
        }
        initial_population.push_back(board_sample);
    }
```

§3.2 Fitness function evaluation

- Implementation of the fitness function used -

```cpp
// given the state of board, return number of non-attacking
    pair of queens
int fitness_function(vector<int> board) {
    int eval = 0;
    for (int i=0; i<board.size(); i++) {
        int col = board[i];
        for (int j=i+1; j<board.size(); j++) {
            if ((board[j]==col+j-i) || (board[j]==col-j+i) ||
                board[j]==col) eval++;
        }
    }
    return (board.size()*(board.size()-1))/2 - eval;
}
```

- Below is the main loop for the program, `population_fitness` is the list of cumulative fitness values for the population. This is made for convenience in the next step, i.e. the random selection.

```cpp
    bool found_solution = false;
    int cumulative_fitness=0;
    for (int i=0; i<population_size; i++){
        int fitness = fitness_function(initial_population[i]);
        cout << fitness << "\n";
        print_board(initial_population[i]);
        cout << "\n";
        if (fitness==(board_size*(board_size-1))/2) {
            found_solution=true;
            solution = initial_population[i];
            break;
        }
        cumulative_fitness += fitness;
        population_fitness[i] = cumulative_fitness;
    }

    if (found_solution) break;
```

- **Termination :** The program is terminated if the fitness function of the sample reaches the optimal value which is N*(N-1)/2 (i.e. The maximum number of non-attacking pairs of queens)

§3.3 Random selection

- Based on the `cumulative_fitness` evaluated in the previous step, we will randomly select `population_size` number of members with a probability proportional to their fitness.

```cpp
    // selection
    for (int i=0; i<population_size; i++){
        int rand_sample = rand()%cumulative_fitness;
        for (int j=0; j<population_size; j++){
            if (rand_sample<population_fitness[j]){
                selected_population[i]=initial_population[j];
                break;
            }
        }
    }
```

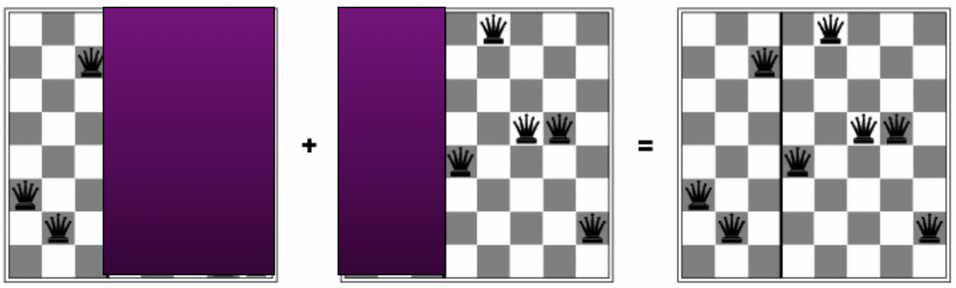


Figure 1: Crossing over for N=8

§3.4 Crossing over

- Implementation of the cross-over function (Comments make the code self-explanatory)-

```
pair<vector<int>, vector<int> > crossover(const vector<int>&
    parent1, const vector<int>& parent2) {
    // Assuming parent1 and parent2 have the same size
    int n = parent1.size();

    // Choose a random crossover point
    int crossover_point = rand() % n;

    // Create the first offspring by combining the first part of
        parent1 and the second part of parent2
    vector<int> offspring1(parent1.begin(), parent1.begin() +
        crossover_point);
    offspring1.insert(offspring1.end(), parent2.begin() +
        crossover_point, parent2.end());

    // Create the second offspring by combining the first part of
        parent2 and the second part of parent1
    vector<int> offspring2(parent2.begin(), parent2.begin() +
        crossover_point);
    offspring2.insert(offspring2.end(), parent1.begin() +
        crossover_point, parent1.end());

    return make_pair(offspring1, offspring2);
}
```

- The main loop of the program -

```
        // cross-over
        for (int i=0; i<population_size; i+=2){
            pair<vector<int>, vector<int> > children =
                crossover(selected_population[i],selected_population[i+1]);
            offsprings[i]=children.first;
            offsprings[i+1]=children.second;
        }
```

§3.5 Mutation

Introduce atmost one mutation in a sample with a probability of 0.5

```
        // mutation
        for (int i=0; i<population_size; i++) {
            if (dis(gen)>0.5){
                int random_row = rand()%board_size;
                int new_col = rand()%board_size + 1;
                mutated_offsprings[i][random_row] = new_col;
            }
        }
```

§3.6 Ending the iteration by updating the initial population

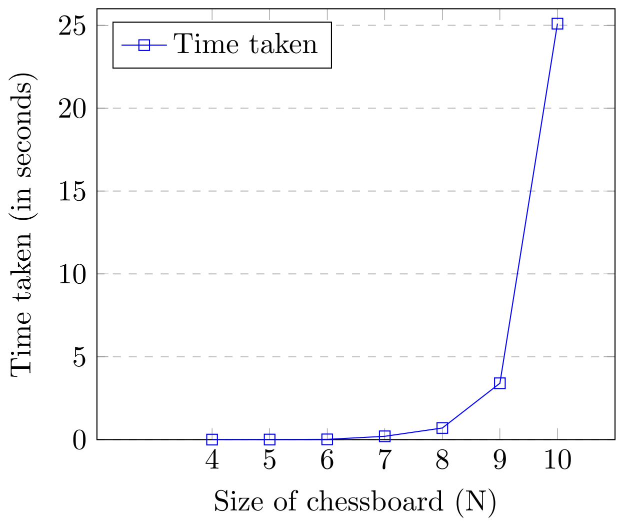
```
        // for next iteration, update initial_population
        initial_population=mutated_offsprings;
```

§4 Results and Analysis

For `population_size = 16`, we got the following results. Note that since this is a randomized algorithm, the time to converge to a solution can vary significantly.

N	Time taken to solve N-queens problem (in s)
4	0.001
5	0.004
6	0.015
7	0.2
8	0.7
9	3.4
10	25.1

Time taken for different total number of lines



- By increasing the `population_size`, we observed difference in the performance of the algorithm for different values of N. For every N, there exist an optimal `population_size` such that the algorithm converges to the optimal quickly. `Note that multiple possible optimal solutions/configurations are possible for the problem, the GA will output one of them`

- Various possible adjustments can be done to the parameters like `population_size`, number of cross-overs and number of mutations. This may improve the performance of the GA.
- Many improvements and tries can be done to improve the algorithm even further.
- One such idea is to introduce *champions*. We can fix some percentage of initial population which has a very high fitness and hence wont modify/change them.
- Another idea is to make a database of bad positions. Such databases can be precomputed and put up in file. Whenever such *bad* samples occur, we can replace them by any random permutation.
- We can change the mutation parameters. One can induce more than one mutation also as opposed to our implementation where atmost one mutation occurs.
- Crossing over step may involve more than one cross-overs as opposed to out implementation where only one cross-over occurs.
- Other methods like simulated annealing can also give better results for the problem, and may converge faster for higher values of N.

§5 Running the Code

1. Compile the project.cpp using `g++ project.cpp` in the same directory as project.cpp
2. Run the executable with `./a.out`
3. The program will prompt the user to enter the size of the board (N). Enter N between 4 to 9 (since N queens problem has no solution for $N \leq 3$ and will take much more time for $N \geq 10$)
4. The program will run and will output the solution. 'X' denotes the position of Queen on the board.

§6 Declaration

All the code implementation, problem formulation and this report is my own work and I have **not** used any resource or any Large Language Models (like ChatGPT) available online.