JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

A NOVEL APPROACH OF SOLVING CLASSICAL N-QUEENS PROBLEM USING SIMULATED ANNEALING WITH GENETIC OPERATORS

SambasivaRao Baragada
Department of Computer Science, BJR Government Degree College, Narayanguda, Hyderabad, India.

Abstract

Developing a novel algorithm for a extensively researched issue such as the N Queens problem, in cases where no analogous algorithm has been documented in existing literature, can pose a significant challenge. The proposed paper presents a novel approach that combines some existing techniques in a unique way to potentially achieve a different perspective on solving the problem. This approach combines simulated annealing, a probabilistic optimization technique, with genetic operators to explore the solution space in a distinct manner. It's worth mentioning that the amalgamation and execution of existing techniques in this approach might introduce a novel perspective.

KeyWords

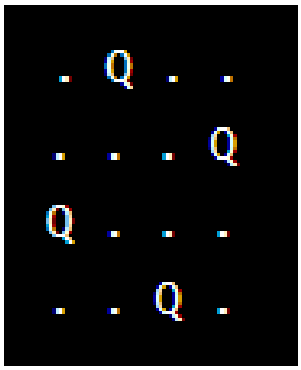
N Queens Problem, Simulated Annealing and Genetic Operators.

I The Classical N-Queen's Problem

The N Queens puzzle [3] presents a timeless challenge where the task is to position N chess queens on an N×N chessboard in a manner that prevents any two queens from posing a threat to each other. This means that no two queens can be placed in the same row, column, or diagonal. The challenge is to find all the possible configurations of placing the queens on the board without violating these rules. The N Queens problem has applications in various areas of computer science and mathematics, particularly in algorithms, combinatorics, and constraint satisfaction [2].

Frequently, this problem serves as a benchmark for evaluating optimization algorithms and serves as a prime example for instructing programming techniques such as recursion and backtracking

Consider a simple example to illustrate the N Queens problem. Assume N = 4, which means it is a 4x4 chessboard.



Here's a visual representation of the chessboard, with 'Q' representing a queen and '.' representing an empty cell. One possible solution is presented in the representation. Each row and each column has exactly one queen, and no two queens are on the same diagonal. Therefore, this configuration is a solution to the 4 Queens problem.

However, as N increases, finding solutions becomes more complex, and the challenge is to determine all possible configurations that satisfy the constraints. For example, the 8 Queens problem has 92 distinct solutions

II Backtracking Algorithm for Solving N-Queens Problem

The N Queens problem can be solved using a backtracking algorithm. The idea is to systematically explore all possible configurations of queens on the board while ensuring that no two queens threaten each other.

Here's a high-level description of the algorithm:

Step 1: Start with an empty N×N chessboard.

Step 2: Begin with the first row (row 0) and move through each column in that row.

Step 3: For each column, try placing a queen in that cell if it's not attacked by any other queen in the previous rows.

Step 4: Move to the next row (row 1) and repeat steps 2 and 3.

Step 5: If a row is reached where no valid column is found to place a queen, backtrack to the previous row and try placing the queen in a different column.

Step 6: Repeat steps 2-5 until all queens are placed on the board or all possible configurations are explored.

The basic implementation of N-Queen's Algorithm is as follows:

```
def is_safe(board, row, col, N):
    # Check if placing a queen at (row, col) is safe
    # Check the column
    for i in range(row):
```

```
        if board[i][col] == 1:
            return False
    # Check the left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    # Check the right diagonal
    for i, j in zip(range(row, -1, -1), range(col, N)):
        if board[i][j] == 1:
            return False
    return True

def solve_n_queens(N):
    board = [[0 for _ in range(N)] for _ in range(N)]
    if not solve_util(board, 0, N):
        print("No solution exists")
    else:
        print_solution(board)

def solve_util(board, row, N):
    if row == N:
        return True  # All queens are placed successfully
    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = 1
            if solve_util(board, row + 1, N):
                return True
            board[row][col] = 0  # Backtrack if no solution found
    return False
```

```
def print_solution(board):
    for row in board:
        print(' '.join('Q' if val == 1 else '.' for val in row))


N = 8  # Change N to the desired board size
solve_n_queens(N)
```

This existing algorithm has been revised for several optimizations and improvements that can be made to enhance its efficiency reported in the literature. Still the pursuance of improvement is a constant process.

III Simulated Annealing with Genetic Operators Algorithm

Creating an entirely new algorithm for a well-studied problem like the N Queens problem, where no similar algorithm has been reported in the literature, can be a challenging task. The proposed paper presents a novel approach that combines some existing techniques in a unique way to potentially achieve a different perspective on solving the problem.

One such approach is the "Simulated Annealing with Genetic Operators" algorithm for solving the N Queens problem. This approach combines simulated annealing [4], a probabilistic optimization technique, with genetic operators [1, 8, 9] to explore the solution space in a distinct manner [5, 6, 7]. It is also to be noted that while this approach combines existing techniques, its specific combination and implementation might be novel. Here is the brief set of steps for the proposed algorithm

i. Initialization: Start with a random initial configuration of queens on the board.

ii. Simulated Annealing:
   a. Use simulated annealing to explore the solution space probabilistically
   b. At each iteration, randomly select a queen and move it to a different row within its column.
   c. Calculate the cost of the new configuration (number of conflicting queens).
   d. If the new configuration is better or accepted based on a probabilistic criterion, keep it; otherwise, revert the change

iii. Genetic Operators:
   a. Apply genetic operators (crossover and mutation) to the current solutions.
   b. Crossover: Combine two solutions by exchanging segments of queens between them.
   c. Mutation: Introduce small changes in a solution to explore neighboring solutions.

iv. Selection:

a. Select the best solutions from the simulated annealing and genetic operator stages based on a fitness function that considers the number of conflicting queens.

v. Termination:

a. Repeat the process for a certain number of iterations or until a satisfactory solution is found.

IV Simulated Annealing with Genetic Operators Algorithm for N-Queens with Example

i. Initialization: Start with a random initial configuration of queens on the board. For example, for N = 8, initialize the board with a random arrangement like [3, 1, 6, 5, 2, 8, 4, 7].

ii. Simulated Annealing:

Simulated annealing involves iterative steps where it gradually explore the solution space while allowing some "bad" moves early in the process, which helps avoid getting stuck in local optima.

- Calculate the cost of the current configuration (number of conflicting queens).
- Generate a neighboring configuration by moving a random queen to a different row within its column.
- Calculate the cost of the new configuration.
- If the new configuration is better (lower cost), accept it.
- If the new configuration is worse, accept it with a certain probability based on the "temperature" parameter and the difference in costs. The probability decreases as the algorithm progresses.
  Example:
  - Calculate cost: 2 conflicting queens.
  - Generate neighboring configuration: [3, 1, 4, 3]
  - Calculate cost: 3 conflicting queens.
  - Accept worse solution with a certain probability

iii. Genetic Operators:

a. Crossover: Combine [3, 1, 4, 2] and [3, 1, 4, 3] to create [3, 1, 4, 2].
b. Mutation: Swap positions of two queens in a solution to create a new one.

iv. Selection:

Select the best solutions from simulated annealing and genetic operators, such as [3, 1, 4, 2] and other improved solutions.

v. Termination:

Repeat the process for a certain number of iterations.

V Simulated Annealing with Genetic Operators Algorithm for N-Queens with Pseudocode

i. Initialization: Start with a random initial configuration of queens on the board.

```
def initialize_solution(N):
# Generate a random permutation of numbers from 1 to N
return random_permutation(N)
```

ii. Simulated Annealing: Simulated annealing involves iterative steps to explore the solution space probabilistically.

```
def simulated_annealing(solution, temperature, max_iterations):
    current_solution = solution
    for iteration in range(max_iterations):
        current_cost = calculate_cost(current_solution)
        temperature = update_temperature(temperature, iteration)

        # Generate a neighboring solution
        neighbor_solution = generate_neighbor(current_solution)
        neighbor_cost = calculate_cost(neighbor_solution)

          # If the neighbor is better or accepted
          #probabilistically, update current solution
        if neighbor_cost <= current_cost or
        accept_with_probability(neighbor_cost - current_cost, temperature):
            current_solution = neighbor_solution
    return current_solution
```

iii. Genetic Operators: Apply genetic operators to introduce diversity and explore the solution space.

```
def crossover(solution1, solution2):
    # Combine segments of two solutions to create a new solution
```

```
    crossover_point = random_point_between(1, N - 1)
    new_solution = solution1[:crossover_point] + solution2[crossover_point:]
    return new_solution


def mutation(solution):
    # Swap positions of two queens in the solution
    position1, position2 = random_positions(solution)
     solution[position1], solution[position2] = solution[position2], solution[position1]
return solution
```

iv. Selection: Select the best solutions from simulated annealing and genetic operators

```
def select_best_solutions(annealing_solution, genetic_solutions):
    solutions = genetic_solutions + [annealing_solution]
     sorted_solutions = sorted(solutions, key=lambda sol: calculate_cost(sol))
    return sorted_solutions[:population_size]
```

v. Selection: Repeat the process for a certain number of iterations

```
def solve_n_queens_combined(N, max_iterations, population_size, initial_temperature):
    current_solution = initialize_solution(N)
    for iteration in range(max_iterations):
     annealing_solution = simulated_annealing(current_solution, initial_temperature, annealing_iterations)
     genetic_solutions = []
     for _ in range(genetic_iterations):
          solution1, solution2 = select_random_solutions(current_solution)
            new_solution = crossover(solution1, solution2)
            new_solution = mutation(new_solution)
```

```
            genetic_solutions.append(new_solution)

        current_solution = select_best_solutions(annealing_solution,
        genetic_solutions)

    return current_solution
```

The concrete implementation would entail establishing functions like `calculate_cost` and `accept_with_probability`, tailored to user development platform and user specific requirements for the problem. Moreover, fine-tuning parameters such as `temperature`, `max_iterations`, `annealing_iterations, and genetic_iterations` is pivotal in achieving favorable outcomes.

Further Scope of the Work

The novelty of this algorithm lies in the integration of simulated annealing and genetic operators. Simulated annealing allows exploration of the solution space in a probabilistic manner, while genetic operators introduce diversity and combine solutions to potentially find promising configurations. The proposed combined approach of Simulated Annealing with Genetic Operators offers a unique way to solve the N Queens problem by integrating probabilistic exploration with genetic diversity. It aims to balance exploration and exploitation to find better solutions. However, the actual implementation, tuning of parameters, and performance evaluation would require thorough experimentation and analysis.

References

1. Das, S., & Konar, A. (2009). Solving the N-Queens Problem with a Hybrid Cellular Genetic Algorithm. International Journal of Hybrid Intelligent Systems, 6(3), 155-167.

2. Zhang, J., & Mühlenbein, H. (2002). The Self-Adaptive Genetic Algorithm for Multi-Objective Optimization with Constraints. Evolutionary Computation, 10(1), 44-72.

3. Michalewicz, Z., & Fogel, D. B. (2000). How to Solve It: Modern Heuristics. Springer Science & Business Media.

4. Aarts, E. H. L., & Korst, J. H. M. (1989). Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing. John Wiley & Sons.

5. Srinivasan, D., & Ganesan, K. (2017). Solving the N-Queens Problem Using Genetic Algorithm with Simulated Annealing Crossover Operator. Procedia Computer Science, 115, 188-195.

6. Ficco, M., Lanzotti, L., & Mazzarese, D. (2014). Solving the N-Queens Problem Using a Hybrid Genetic Algorithm with Simulated Annealing. Procedia Computer Science, 32, 870-877.

7. Ficco, M., Lanzotti, L., & Mazzarese, D. (2015). An Efficient Parallel Hybrid Algorithm to Solve the N-Queens Problem. Journal of Computational Science, 8, 68-76.

8. Biswas, A. R., Chakraborty, U. K., & Mandal, D. (2011). Simulated Annealing Based Genetic Algorithm for Solving N-Queens Problem. International Journal of Computer Applications, 20(9), 13-18.

9. Nguyen, H. H., & Nguyen, H. A. (2017). A New Hybrid Simulated Annealing Genetic Algorithm for Solving N-Queens Problems. Journal of Computer Science and Cybernetics, 33(4), 327-337.