CP468 - Artificial Intelligence

Wilfrid Laurier University

AI Final Project

April 5th, 2024

M Queens Implementation

Authors:
Sanjae Suresh Kumar, 210523530
Pranav Verma, 193272030
Timson Liu, 200870060
Evan Ellig, 200431290

Index

1 Abstract

The project addresses the N-Queens problem, a classic example in the field of artificial intelligence. The challenge involves placing N queens on an N×N chessboard in such a manner that no two queens threaten each other. This means no two queens can be in the same row, column, or diagonal. The complexity of the problem exponentially increases with the size of the board, making it a significant test of algorithm efficiency and optimization. However, in our modified version of the problem, we aim to find solutions for a subset of the original problem, where the number of queens M is less than or equal to the size of the board N, and searches for the most optimal solution that takes least number of moves to reach from start state to the goal state.

To achieve this modification, we adjust the problem's constraints and search space accordingly. Instead of aiming to place N queens on an N×N board, we seek to find solutions for M queens on an N×N board, where M can be any integer less than or equal to N. This adjustment expands the problem's applicability, allowing us to explore scenarios with varying queen counts on boards of different sizes. Also, our implementation allows for a scenario where the initial placement of a subset of queens (M queens) is given, and it seeks to find a solution that minimizes the number of moves required to reach a conflict-free state, using a heuristic approach that integrates A* search.

A significant innovation in our M Queens approach is the introduction of a user-defined initial state, enabling the algorithm to commence from a specific configuration of queens on the board. Our algorithm then applies heuristic searches to iteratively navigate through solutions towards one with the least cost, quantified as the minimum number of moves required to reach a state where no queens threaten each other. This user-centric design not only enhances the algorithm's applicability to diverse problem instances but also showcases the adaptability and robustness of heuristic search techniques in solving complex combinatorial puzzles.

Utilizing A* search algorithms becomes instrumental in efficiently navigating through the vast search space of possible queen configurations. A* search combines the advantages of both breadth-first and depth-first search strategies while incorporating heuristic information to guide the exploration process. In the context of the N queens problem, A* search helps us efficiently traverse the solution space by prioritizing promising candidate configurations based on heuristic estimates of their potential to lead to a solution. By leveraging A* search, we can effectively identify and explore feasible solutions for the modified N queens problem within a reasonable computational time frame, enabling us to find optimal or near-optimal solutions for a given board size and queen count.

The M-Queens problem, with its combinatorial nature and exponential solution space growth with increasing board size, serves as an excellent benchmark for assessing algorithmic efficiency in constraint satisfaction problems. Our project underscores the significance of heuristic searches, particularly the A* algorithm, in overcoming the inherent challenges of this problem by efficiently pruning the search space and prioritizing promising solutions. Through this work, we contribute to the broader field of artificial intelligence by demonstrating the practical application and effectiveness of advanced search techniques in complex problem-solving scenarios.

1.1 Primary Algorithmic Approaches

A* Algorithm: This search algorithm is known for its efficiency in finding the shortest path between two points, making it an ideal choice for solving the M-Queens problem. The implementation uses a priority queue to explore successor states, updating scores based on the heuristic cost estimate. This heuristic includes considerations for conflicts (i.e., pairs of queens threatening each other) and the distance to the closest goal state, enabling the algorithm to efficiently navigate toward the solution.

Backtracking Method: Recognized for its utility in solving computational problems incrementally, the backtracking method systematically explores the search space by trying different queen positions on the board. It abandons a candidate solution and backtracks as soon as it becomes apparent that the current path will not lead to a viable solution. This depth-first search approach is particularly effective for the M-Queens problem as it allows for the gradual construction of solution candidates, discarding those that do not satisfy the problem's constraints.

1.2 GUI Overview

For tackling the M queens problem and providing a user-friendly interface, we implemented our solution using the Python programming language. Python's simplicity, readability, and extensive libraries make it an ideal choice for algorithmic problem-solving tasks like this. Leveraging Python's rich ecosystem of libraries, we utilized Tkinter, a standard GUI (Graphical User Interface) toolkit for Python, to develop the frontend of our application.

Tkinter offers a straightforward approach to building graphical interfaces in Python, making it accessible for developers to create interactive applications with buttons, labels, and other GUI elements. By employing Tkinter, we crafted an intuitive frontend that allows users to interact with the N queens problem solver effortlessly. Users can input parameters such as the size of the chessboard and the number of queens they want to place, and they can visualize the solutions generated by the algorithm in a user-friendly manner.

2 Differences from Prolog Solution

The exploration of algorithmic strategies to solve the N-Queens problem reveals a spectrum of approaches, each with its distinct focus and optimization criteria. Our project and the professor's implementation embody two such diverse methodologies, underlining the breadth of computational tactics deployable against this enduring puzzle. Here, we delve into the core differences between these approaches, particularly emphasizing our heuristic-based solution aimed at optimizing the path from an initial state to a solution, contrasted against the professor's exhaustive search for all possible solutions.

2.1 Implementation from Class

Our professor's approach, as demonstrated in Program 1 for the eight queens problem, employs a recursive backtracking algorithm that iterates through possible board configurations to identify all solutions where no queens attack each other. This method systematically explores the search space by placing a queen in a valid position, proceeding to the next position, and backtracking when a conflict is detected. The algorithm checks each candidate solution (a list of non-attacking queen positions) against the constraints of the problem:

- No two queens share the same row (implicitly ensured by the representation).
- No two queens share the same column.
- No two queens are placed on the same diagonal. This exhaustive search method is comprehensive in finding all possible solutions but does not prioritize the search based on solution quality or cost (in terms of moves required to reach a solution).

2.2 M Queens Approach

In contrast, our project adopts a modified N-Queens problem by focusing on finding an optimal solution for a specified number of queens ($M \leq N$) from an inputted initial state, using heuristic searches. Our implementation leverages the A* search algorithm, known for its efficiency in solving complex problems by combining the best features of depth-first and breadth-first searches with heuristics to guide the search towards the most promising solutions.
The key differences in our approach are:

- **Initial State Input**: Unlike the exhaustive search that starts from an empty board, our algorithm begins from a user-defined initial state. This feature is particularly useful for scenarios requiring optimization from a given configuration rather than starting from scratch.
- **Heuristic Evaluation**: We utilize heuristic cost estimates to evaluate the potential of each state, guiding the algorithm to prioritize moves that lead to the least cost solution. This heuristic considers the minimum moves required for each queen to reach a position where it does not threaten any other queen, efficiently pruning the search space.
- **Solution Cost Optimization**: Our focus is not merely on finding any solution but on identifying the solution that can be achieved with the least number of moves from the initial state. This approach adds a layer of complexity and practicality, as it aligns with real-world scenarios where the cost of reaching a solution is a critical factor.
- **Scope of Solutions**: While the professor's implementation seeks to find all possible solutions without regard to their starting point or efficiency, our project aims to find the best solution from a given starting point, emphasizing the algorithm's ability to find efficient paths through the search space.

3 Solution Implementation

3.1 Technologies Used

We implemented our solution in Python. Python was preferred because:

- **Simplicity and Readability**: Python is widely known for its simplicity and readability. It allows us as developers to express concepts in fewer lines of code compared to other languages, making the code easier to understand, maintain and update. This was crucial for our project which is based on N-Queens solver where clarity is essential for both developers and users.

- **Availability of Libraries**: Python has a vast ecosystem of libraries and frameworks, including Tkinter for GUI development and various data manipulation libraries. Utilizing these libraries simplifies the implementation of complex functionalities such as implementing search algorithms like A* as well as the graphical interface.
- **Prototype Development**: Python's rapid development capabilities make it ideal for prototyping and iterating on ideas quickly. Developing the N-Queens solver in Python allows for rapid prototyping of the GUI and algorithm implementation, enabling faster iterations and improvements.

3.2 A* Algorithm Implementation

A* search algorithm is one of the most popular techniques used in pathfinding and graph traversals. It is a searching algorithm that uses the shortest path between an initial and a final point, it is often used for map traversal to find the said shortest path that is to be taken.

3.2.1 a_star_search()

This wrapper function initiates the A* search process by putting the initial state into the priority queue. It maintains g_score and f_score dictionaries to keep track of the cost from the start node and the total estimated cost from the start node to the goal node passing through the current node respectively. It iteratively explores successor states, updates the scores, and adds them to the priority queue until a goal state is reached or the queue becomes empty.

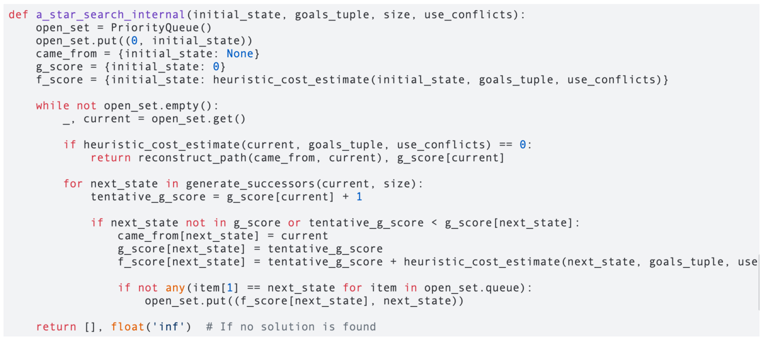
```python
def a_star_search(initial_state, goals, size):
    initial_state = tuple(sorted(initial_state))
    goals_tuple = tuple(map(tuple, goals))  # Make sure goals are a tuple of tuples

    # Run A* search to find the closest solution
    closest_solution_path, closest_cost = a_star_search_internal(initial_state, goals_tuple, size, use_conflic

    # Run A* search again to consider both conflicts and distance to closest solution
    combined_solution_path, combined_cost = a_star_search_internal(initial_state, goals_tuple, size, use_confl

    # Choose the better solution based on total cost
    if closest_cost < combined_cost:
        return closest_solution_path, closest_cost
    else:
        return combined_solution_path, combined_cost
```

3.2.2 a_star_internal_search()

This function is the internal implementation of the A* search algorithm. It considers whether to include conflicts in the heuristic cost estimate based on the use conflicts parameter. It handles the core logic of A* search including updating scores, managing the priority queue, and reconstructing the path to the goal state.

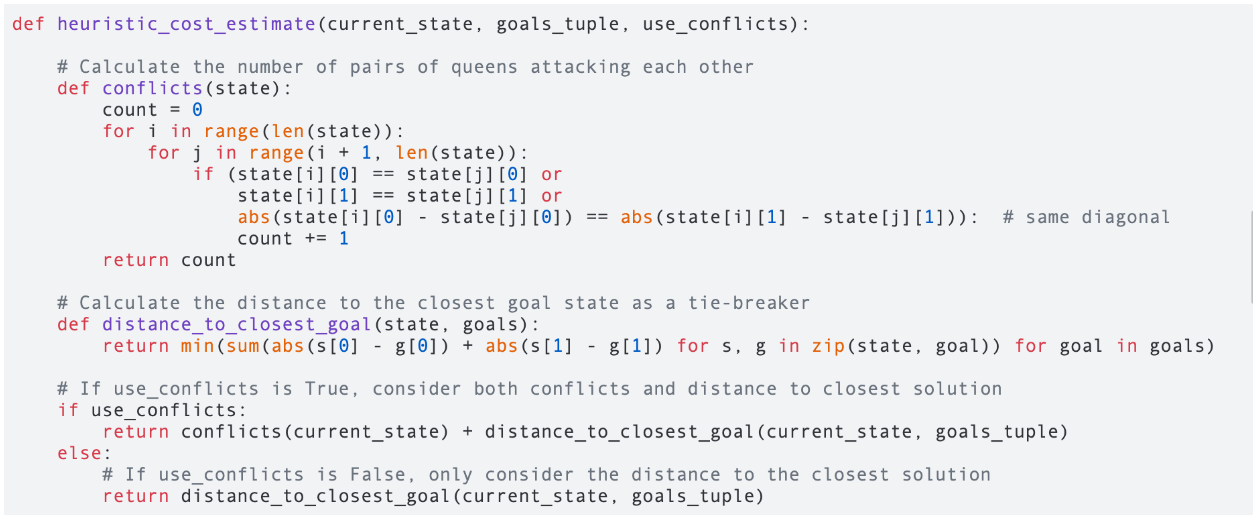
3.2.3 heuristic_cost_estimate()

The heuristic cost estimate in the heuristic_cost_estimate function plays a crucial role in guiding the search process. It considers both conflicts (pairs of queens attacking each other) and the distance to the closest



```python
def a_star_search_internal(initial_state, goals_tuple, size, use_conflicts):
    open_set = PriorityQueue()
    open_set.put((0, initial_state))
    came_from = {initial_state: None}
    g_score = {initial_state: 0}
    f_score = {initial_state: heuristic_cost_estimate(initial_state, goals_tuple, use_conflicts)}

    while not open_set.empty():
        _, current = open_set.get()

        if heuristic_cost_estimate(current, goals_tuple, use_conflicts) == 0:
            return reconstruct_path(came_from, current), g_score[current]

        for next_state in generate_successors(current, size):
            tentative_g_score = g_score[current] + 1

            if next_state not in g_score or tentative_g_score < g_score[next_state]:
                came_from[next_state] = current
                g_score[next_state] = tentative_g_score
                f_score[next_state] = tentative_g_score + heuristic_cost_estimate(next_state, goals_tuple, use

                if not any(item[1] == next_state for item in open_set.queue):
                    open_set.put((f_score[next_state], next_state))

    return [], float('inf')  # If no solution is found
```

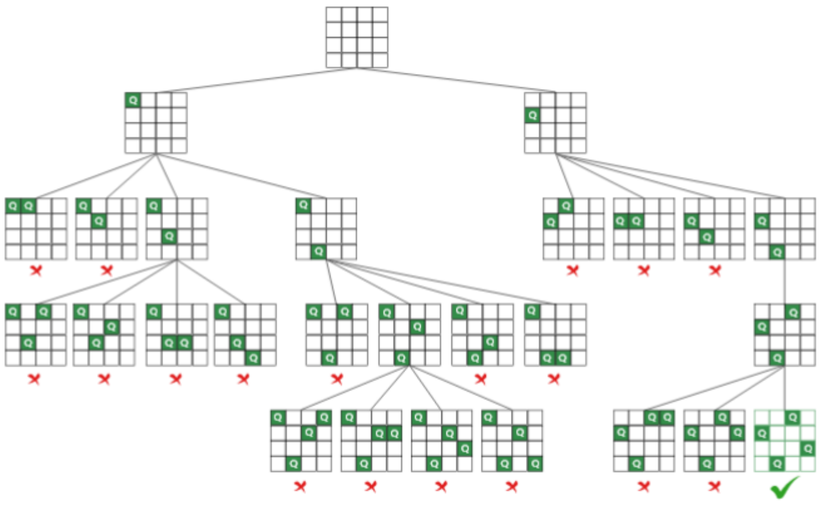
goal state. This heuristic helps A* search to efficiently navigate the search space towards the goal state. The heuristic cost (h(n)) is crucial for guiding the search toward the goal efficiently. It is designed to estimate the ”closeness” to a solution, taking into account both the minimum number of moves required to reach a state matching any pre-generated solution and the number of conflicts between queens.



```python
def heuristic_cost_estimate(current_state, goals_tuple, use_conflicts):

    # Calculate the number of pairs of queens attacking each other
    def conflicts(state):
        count = 0
        for i in range(len(state)):
            for j in range(i + 1, len(state)):
                if (state[i][0] == state[j][0] or
                    state[i][1] == state[j][1] or
                    abs(state[i][0] - state[j][0]) == abs(state[i][1] - state[j][1])):  # same diagonal
                    count += 1
        return count

    # Calculate the distance to the closest goal state as a tie-breaker
    def distance_to_closest_goal(state, goals):
        return min(sum(abs(s[0] - g[0]) + abs(s[1] - g[1]) for s, g in zip(state, goal)) for goal in goals)

    # If use_conflicts is True, consider both conflicts and distance to closest solution
    if use_conflicts:
        return conflicts(current_state) + distance_to_closest_goal(current_state, goals_tuple)
    else:
        # If use_conflicts is False, only consider the distance to the closest solution
        return distance_to_closest_goal(current_state, goals_tuple)
```

3.3 Backtracking Algorithm Implementation

Backtracking is a general algorithm for solving computational problems, most notably constraint satisfaction problems, that incrementally builds candidates to the solutions and abandons a candidate's backtracks as soon as it determines that the candidate cannot be completed to a reasonable solution. The backtracking algorithm is used in various applications, including the N-queen problem, the knight tour problem, maze solving problems, and the search for all Hamilton paths in a graph. It is a brute force algorithm to find all solutions to a problem. It entails gradually compiling a set of all possible solutions. Because a problem will have constraints, solutions that do not meet the said constraints are removed. It finds solutions by building a solution step by step, increasing levels over time by using recursive calling. Backtracking implements depth-first search methods. The algorithm begins to explore the solutions, the abounding function is applied so that the algorithm can determine whether the proposed solution satisfies the constraints. If it does, it will keep looking, if not, the branch is removed, and the algorithm returns to the previous level.



3.3.1 generate_queens_for_m_queens

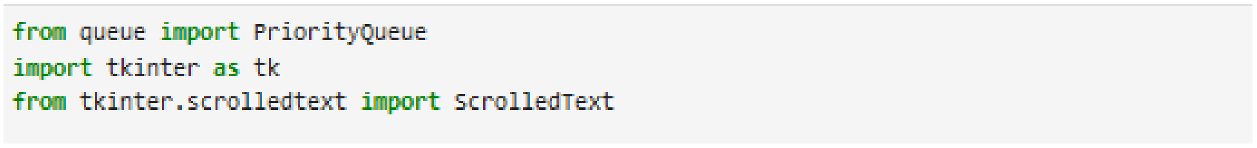
Implements the backtracking method to generate solutions for the N-Queens problem. It explores the search space by recursively trying out different queen positions and backtracking when a solution cannot be found with the current configuration. Adjustments are made for cases where the number of queens (m) is different from the board size (n).

```python
def generate_solutions_for_m_queens(n, m):
    solutions = []

    # Adjusted is_safe to match the dynamic version's parameters but work with 0-index internally
    def is_safe(queen, queens):
        for q in queens:
            if q[0] == queen[0] or q[1] == queen[1] or abs(q[0] - queen[0]) == abs(q[1] - queen[1]):
                return False
        return True

    # For M = N, use the standard backtracking method
    def backtrack_standard(queens, row=0):
        if len(queens) == m:
            solutions.append(sorted([(r+1, c+1) for r, c in queens]))
            return
        for col in range(n):
            # Adjusted to pass parameters in a manner consistent with the dynamic approach
            if is_safe((row+1, col+1), [(q_row+1, q_col+1) for q_row, q_col in queens]):
                queens.append((row, col))
                backtrack_standard(queens, row + 1)
                queens.pop()

    # For M != N, explore more dynamically
    def backtrack_dynamic(queens=[], start=0):
        if len(queens) == m:
            solutions.append(sorted([(r+1, c+1) for r, c in queens]))
            return
        for i in range(start, n * n):
            row, col = divmod(i, n)
            if is_safe((row+1, col+1), [(q_row+1, q_col+1) for q_row, q_col in queens]):
                queens.append((row, col))
                backtrack_dynamic(queens, i + 1)
                queens.pop()

    # Decide which method to use based on M and N
    if m == n:
        backtrack_standard([])
    else:
        backtrack_dynamic([])

    return solutions
```

4 Code Overview

4.1 Header files

- PriorityQueue from the queue module: This module was used to implement the open set in the A* search algorithm.
- tkinter as tk: We used the standard Python interface to the Tk GUI toolkit and was used to create the GUI components.
- ScrolledText from tkinter.scrolledtext: This is a text widget with a vertical scroll bar attached, allowing for displaying large amounts of text.



```
from queue import PriorityQueue
import tkinter as tk
from tkinter.scrolledtext import ScrolledText
```

4.2 Class

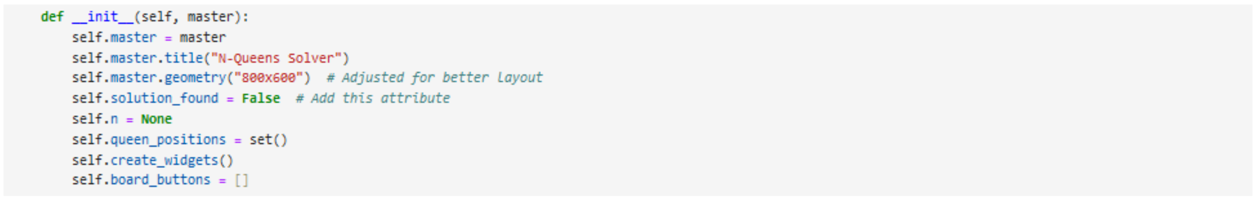
Class NQueensApp represents the main application window.

4.3 Functions

4.3.1 def __init__()

The __init__ method initializes the application:

- self.master: Used as reference to the Tkinter root window.
- self.master.title(): Used to set the title of the window.
- self.master.geometry(): Used to sets the initial size of the window.
- self.solution_found: Used to flag a track if a solution has been found
- self.n: Variable that stores the size of the chessboard.
- self.queen_positions: Variable that stores the positions of the queens on the board.
- self.create_widgets(): It calls the method to create GUI widgets.
- self.board_buttons: List to hold references to buttons representing squares on the chess- board.

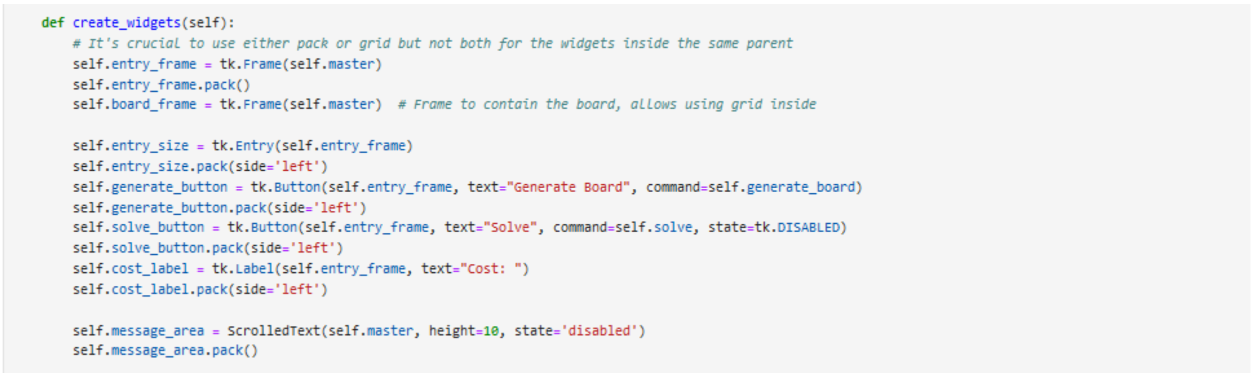


```
    def __init__(self, master):
        self.master = master
        self.master.title("N-Queens Solver")
        self.master.geometry("800x600")  # Adjusted for better layout
        self.solution_found = False  # Add this attribute
        self.n = None
        self.queen_positions = set()
        self.create_widgets()
        self.board_buttons = []
```

4.3.2 def create_widget()

This method creates the widgets for the GUI:

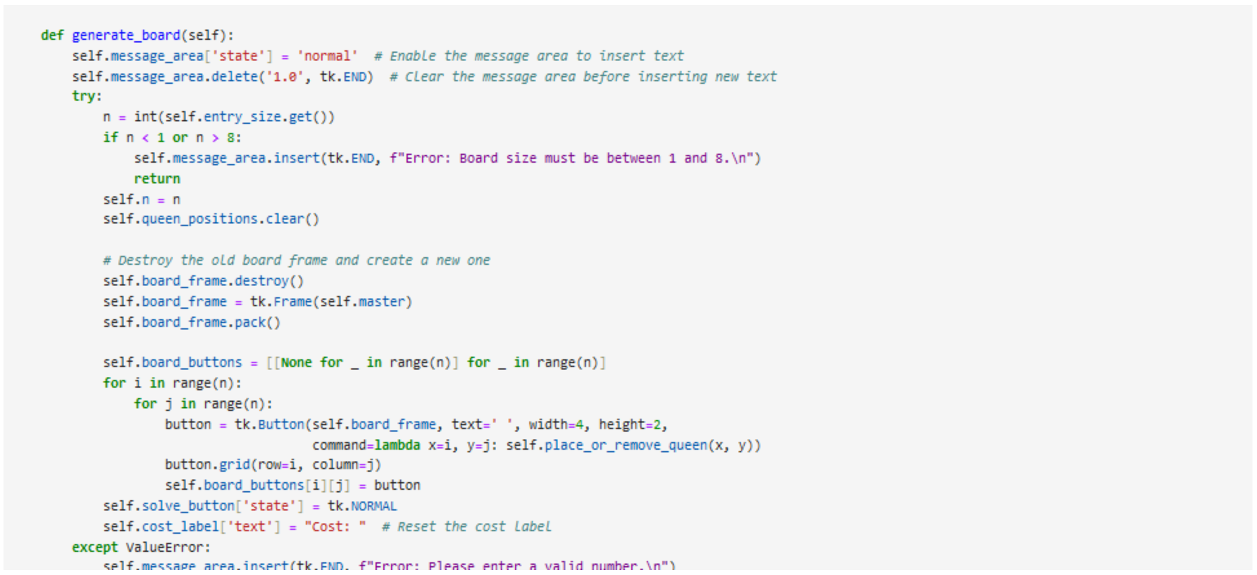
- self.entry_frame: Provides the frame to contain the entry widgets.
- self.board_frame: Provides frame to contain the chessboard buttons.
- self.entry_size: Provides the input widget for entering the size of the chessboard.
- self.generate_button: Provide the buttons to generate the chessboard.
- self.solve_button: Provides a button to solve the N- Queens problem.
- self.cost_label: Label to display the cost of the solution.
- self.message_area: ScrolledText widget to dis- play messages and logs.



```python
def create_widgets(self):
    # It's crucial to use either pack or grid but not both for the widgets inside the same parent
    self.entry_frame = tk.Frame(self.master)
    self.entry_frame.pack()
    self.board_frame = tk.Frame(self.master)  # Frame to contain the board, allows using grid inside

    self.entry_size = tk.Entry(self.entry_frame)
    self.entry_size.pack(side='left')
    self.generate_button = tk.Button(self.entry_frame, text="Generate Board", command=self.generate_board)
    self.generate_button.pack(side='left')
    self.solve_button = tk.Button(self.entry_frame, text="Solve", command=self.solve, state=tk.DISABLED)
    self.solve_button.pack(side='left')
    self.cost_label = tk.Label(self.entry_frame, text="Cost: ")
    self.cost_label.pack(side='left')

    self.message_area = ScrolledText(self.master, height=10, state='disabled')
    self.message_area.pack()
```

4.3.3 def Generate_board()

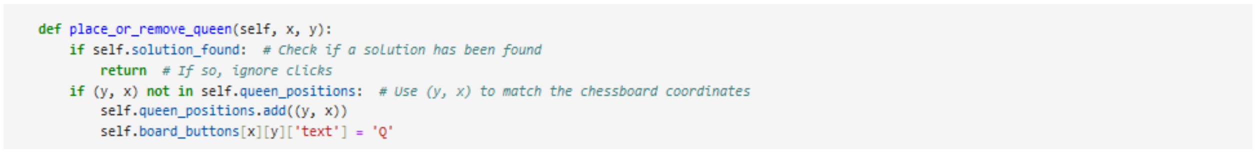
This method is called when the ”Generate Board” button is clicked. It reads the size of the board entered by the user. It checks the size of the board if the size is valid, it clears the previous board and creates a new one. Each square chessboard is represented by a button and by clicking a button we can place or remove a queen on that square. If an invalid size is entered or an invalid number is encountered, an error message is displayed.



```python
def generate_board(self):
    self.message_area['state'] = 'normal'  # Enable the message area to insert text
    self.message_area.delete('1.0', tk.END)  # Clear the message area before inserting new text
    try:
        n = int(self.entry_size.get())
        if n < 1 or n > 8:
            self.message_area.insert(tk.END, f"Error: Board size must be between 1 and 8.\n")
            return
        self.n = n
        self.queen_positions.clear()

        # Destroy the old board frame and create a new one
        self.board_frame.destroy()
        self.board_frame = tk.Frame(self.master)
        self.board_frame.pack()

        self.board_buttons = [[None for _ in range(n)] for _ in range(n)]
        for i in range(n):
            for j in range(n):
                button = tk.Button(self.board_frame, text=' ', width=4, height=2,
                                   command=lambda x=i, y=j: self.place_or_remove_queen(x, y))
                button.grid(row=i, column=j)
                self.board_buttons[i][j] = button
        self.solve_button['state'] = tk.NORMAL
        self.cost_label['text'] = "Cost: "  # Reset the cost label
    except ValueError:
        self.message_area.insert(tk.END, f"Error: Please enter a valid number.\n")
```

4.3.4 def place_or_remove_queen()

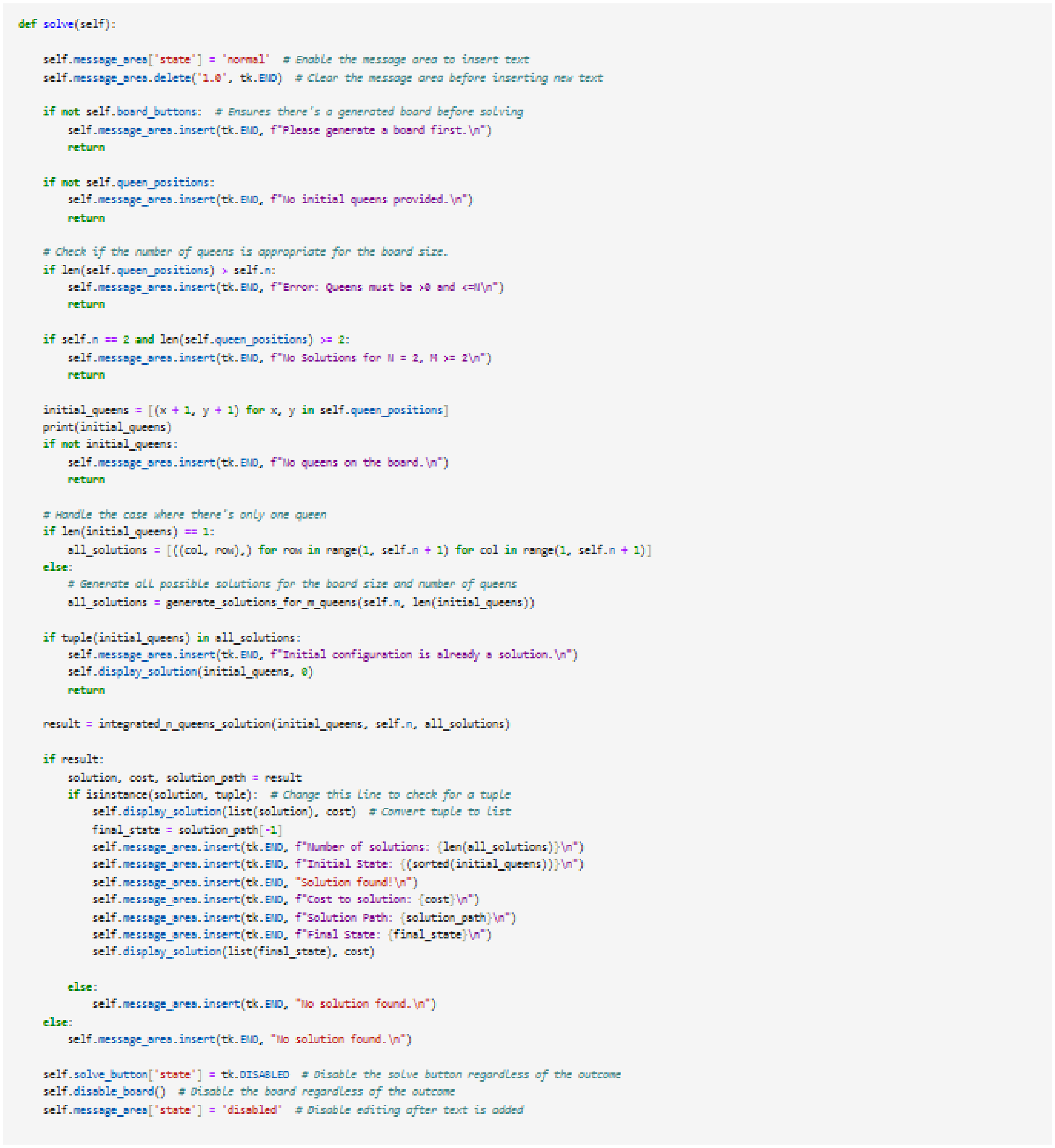
This method is called when a button on the chessboard is clicked and allows users to place or remove a queen on a square of the chessboard. It prevents further queen placements if a solution has already been found.



```python
def place_or_remove_queen(self, x, y):
    if self.solution_found:  # Check if a solution has been found
        return  # If so, ignore clicks
    if (y, x) not in self.queen_positions:  # Use (y, x) to match the chessboard coordinates
        self.queen_positions.add((y, x))
        self.board_buttons[x][y]['text'] = 'Q'
```

4.3.5 def solve()

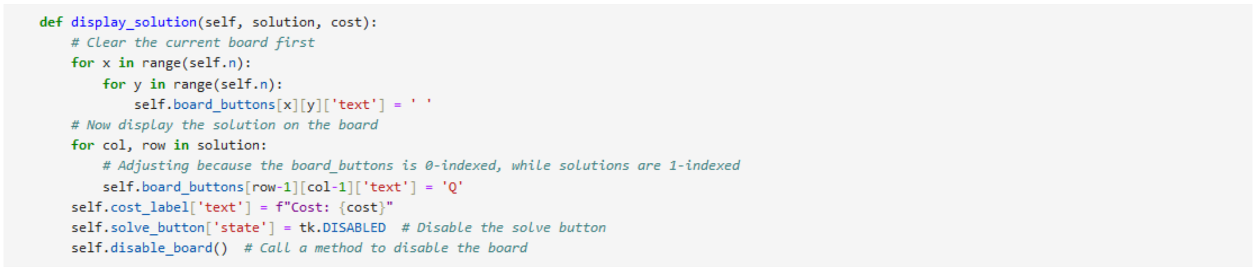
This method is called when the ”Solve” button is clicked. It performs various checks to ensure that a solution can be found. It then generates the initial configuration of queens and checks if it’s already a solution. If it is solved it prints statements like “no solution found” or “already solved”. It then calls the integrated_n_queens_solution() function to solve the N-Queens problem. If a solution is found, it displays information about the solution and updates the GUI accordingly.



```python
def solve(self):

    self.message_area['state'] = 'normal'  # Enable the message area to insert text
    self.message_area.delete('1.0', tk.END)  # Clear the message area before inserting new text

    if not self.board_buttons:  # Ensures there's a generated board before solving
        self.message_area.insert(tk.END, f"Please generate a board first.\n")
        return

    if not self.queen_positions:
        self.message_area.insert(tk.END, f"No initial queens provided.\n")
        return

    # Check if the number of queens is appropriate for the board size.
    if len(self.queen_positions) > self.n:
        self.message_area.insert(tk.END, f"Error: Queens must be >0 and <=N\n")
        return

    if self.n == 2 and len(self.queen_positions) >= 2:
        self.message_area.insert(tk.END, f"No Solutions for N = 2, M >= 2\n")
        return

    initial_queens = [(x + 1, y + 1) for x, y in self.queen_positions]
    print(initial_queens)
    if not initial_queens:
        self.message_area.insert(tk.END, f"No queens on the board.\n")
        return

    # Handle the case where there's only one queen
    if len(initial_queens) == 1:
        all_solutions = [((col, row),) for row in range(1, self.n + 1) for col in range(1, self.n + 1)]
    else:
        # Generate all possible solutions for the board size and number of queens
        all_solutions = generate_solutions_for_m_queens(self.n, len(initial_queens))

    if tuple(initial_queens) in all_solutions:
        self.message_area.insert(tk.END, f"Initial configuration is already a solution.\n")
        self.display_solution(initial_queens, 0)
        return

    result = integrated_n_queens_solution(initial_queens, self.n, all_solutions)

    if result:
        solution, cost, solution_path = result
        if isinstance(solution, tuple):  # Change this line to check for a tuple
            self.display_solution(list(solution), cost)  # Convert tuple to list
            final_state = solution_path[-1]
            self.message_area.insert(tk.END, f"Number of solutions: {len(all_solutions)}\n")
            self.message_area.insert(tk.END, f"Initial State: {(sorted(initial_queens))}\n")
            self.message_area.insert(tk.END, "Solution found!\n")
            self.message_area.insert(tk.END, f"Cost to solution: {cost}\n")
            self.message_area.insert(tk.END, f"Solution Path: {solution_path}\n")
            self.message_area.insert(tk.END, f"Final State: {final_state}\n")
            self.display_solution(list(final_state), cost)

        else:
            self.message_area.insert(tk.END, "No solution found.\n")
    else:
        self.message_area.insert(tk.END, "No solution found.\n")

    self.solve_button['state'] = tk.DISABLED  # Disable the solve button regardless of the outcome
    self.disable_board()  # Disable the board regardless of the outcome
    self.message_area['state'] = 'disabled'  # Disable editing after text is added
```

4.3.6 def display_solution()

This method displays the solution on the chessboard. It clears the previous state of the board. It also places queens on the squares according to the solution found and updates the cost label and disables further

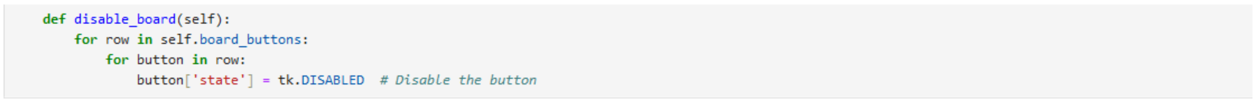


```python
def display_solution(self, solution, cost):
    # Clear the current board first
    for x in range(self.n):
        for y in range(self.n):
            self.board_buttons[x][y]['text'] = ' '
    # Now display the solution on the board
    for col, row in solution:
        # Adjusting because the board_buttons is 0-indexed, while solutions are 1-indexed
        self.board_buttons[row-1][col-1]['text'] = 'Q'
    self.cost_label['text'] = f"Cost: {cost}"
    self.solve_button['state'] = tk.DISABLED  # Disable the solve button
    self.disable_board()  # Call a method to disable the board
```

interaction with the board.

4.3.7 def disable_board()

This method disables interaction with the chessboard, it disables all the buttons on the chessboard, preventing users from modifying the board after a solution has been found.



```python
def disable_board(self):
    for row in self.board_buttons:
        for button in row:
            button['state'] = tk.DISABLED  # Disable the button
```

5 Debugging Methods

5.1 Error Handling

In our code, error handling mechanisms have been incorporated to ensure that inputs provided by users are valid and fall within acceptable bounds. For instance, when users input values for parameters such as the size of the chessboard or the number of queens to place, the code verifies that these values are integers and within reasonable ranges. If an invalid input is detected, appropriate error messages are displayed to inform users about the nature of the error and how to rectify it. These error messages guide users in providing correct inputs, thus enhancing the usability and robustness of the application.

5.2 Debugging Information

We have integrated print statements and message displays strategically throughout the code to provide valuable debugging information during execution. For instance, when the algorithm starts, it prints the initial state of the chessboard, showing the placement of any existing queens. As the algorithm progresses, it may display the solution path it explores, allowing developers to trace the algorithm's steps and identify any potential issues. Additionally, the code may output the cost associated with each solution path or other relevant metrics, aiding in the evaluation of algorithm performance. These debugging messages offer insights into the program's execution flow, facilitating the identification and resolution of any bugs or unexpected behaviors encountered during development or testing.

6 Future Improvements

To improve the M-Queens Solver program, several optimizations can be considered, with a primary focus on enhancing efficiency. First and foremost, eliminating intermediary functions that serve as bridges between the user interface and the algorithm can streamline the code structure, making it more readable and maintainable. This simplification can also contribute to efficiency by reducing unnecessary computational overhead associated with function calls and data processing. Additionally, adopting descriptive variable names and organizing the code into modular components can enhance its understandability, facilitating future updates and modifications aimed at further improving efficiency.

Furthermore, exploring different algorithms, such as genetic algorithms or simulated annealing, holds promise for significantly enhancing the program's efficiency. These alternative approaches can offer novel ways to search the solution space more effectively, potentially yielding better solutions in a shorter amount of time. By leveraging advanced algorithms tailored to the specific characteristics of the N-Queens problem, the program can achieve superior performance, especially when dealing with larger board sizes or complex scenarios.

In summary, prioritizing efficiency enhancements in the N-Queens Solver program is crucial for its effectiveness and scalability. By optimizing code structure, adopting clearer naming conventions, and integrating advanced algorithms, the program can evolve into a more robust and efficient tool for solving the classic N-Queens problem. These improvements not only enhance the program's practical utility but also contribute to its broader applicability in tackling increasingly challenging instances of the N-Queens problem.

7 References

- https://www.durangobill.com/N_Queens.html
- https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/
- https://github.com/romel309/8_puzzle_solver_prolog/tree/master
- item https://www.geeksforgeeks.org/a-search-algorithm/
- https://www.geeksforgeeks.org/a-search-algorith