LinkedIn Puzzle (Queens) Solver Using A star

Bob Kunanda - 13523086
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: bobkunanda@gmail.com , 13523086@std.stei.itb.ac.id

***Abstract*— This paper presents an implementation of the Queens puzzle using the A* search algorithm. The puzzle is a variation of the classic n-queens problem with additional constraints: queens must not be placed within one cell of each other, no two queens can share the same row or column, and each irregularly shaped area on the board must contain exactly one queen. The board is represented using a 2D array for placements and an area map. A* is used to explore possible queen placements by expanding valid configurations, helped by a heuristic function based on placement progress. Although the A* algorithm successfully finds a solution, this paper concludes that it may not be the most suitable method due to the puzzle's nature as a constraint satisfaction problem rather than a pathfinding one. A backtracking approach may be more efficient and better aligned with the problem structure. This project serves as an exercise in applying algorithmic techniques and problem modeling to non-traditional search problems.**

Keywords—queens puzzle; A search; constraint satisfaction; backtracking; algorithm design

I. Introduction

The Queens puzzle is a modern logic game featured on LinkedIn, inspired by the classic 8 Queens puzzle but with a unique twist. While the traditional puzzle requires players to place eight queens on a chessboard such that no two attack each other, including diagonally the Queens puzzle removes the diagonal restriction and introduces additional constraints like jigsaw sudoku. Players must fill the board entirely with queens, ensuring that no two queens are in the same row or column, that no queen is placed within one-block radius of another, and that each irregularly shaped area on the board contains only one queen.

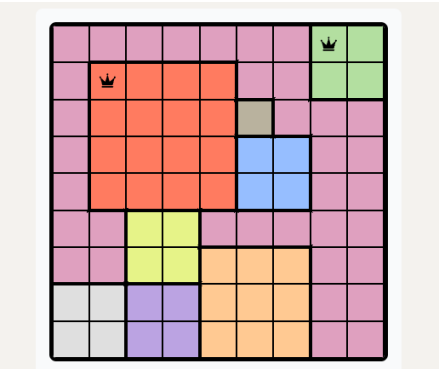

Image 1. Example of Queens Puzzle, taken from [1]

The purpose of this paper is to explore the logic and constraints of the Queens puzzle and to apply algorithmic knowledge to implement one or more possible solutions. By approaching the puzzle from a computational perspective, this study aims to deepen understanding of constraint-based problem-solving and demonstrate how algorithm design can be used to efficiently solve complex board-based logic challenges.

II. Theoretical Foundation

A. A Algorithm*

The A* algorithm is a widely used pathfinding and graph traversal technique that finds the shortest path between two nodes in a weighted graph. It combines the strengths of UCS algorithm and greedy best-first search by considering both the actual cost to reach a node and the estimated cost to reach the goal from that node. This is achieved using a heuristic function, making A* both complete and optimally efficient under certain conditions.

Formally, A* uses the following evaluation function for each node n:

$$f(n) = g(n) + h(n)$$

Where:

$g(n)$ = is the actual cost from the starting node to the current node n,

$h(n)$ = is the heuristic estimate of the cost from n to the goal node,

$f(n)$ = is the total estimated cost of the cheapest solution through n.

In the context of this implementation, each board state is treated as a node. The cost g(n) is represented by the variable step, which indicates how many queen placements have been made so far. The heuristic h(n) is represented by filled_count, which tracks how many cells have been marked) to estimate progress toward completing the board.

The custom priority logic is implemented in the __lt__() method of the Board_state class as follows:

```
def __lt__(self, other):
    """Comparison rule for prioqueue"""
    return self.filled_count + self.step > other.filled_count + other.step
```

Image 2. Heuristic for Prioqueue Function, taken from [2]

This function defines the comparison rule used by Python's heapq module, which manages the open set as a priority queue. Since heapq is a min-heap, the comparison is inverted using > to simulate a maximizing function giving higher priority to states with a higher combined score of progress and path cost. This inversion ensures that board states that are closer to a complete and valid configuration are expanded earlier.

By defining f(n) in this way, the algorithm effectively prioritizes board states that are not only closer to the goal but also progressing steadily through legal placements. While the heuristic is simple, it is admissible and consistent, which maintains the correctness of the A* algorithm in this context.

III. Related Work

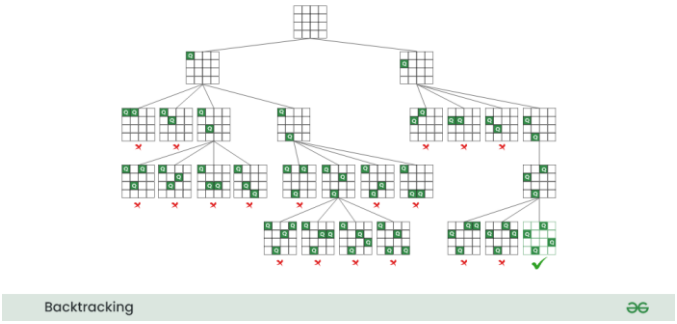

Image 3. N-Queens Problem , taken from [3]

The Queens puzzle presented in this paper is inspired by the classical N-Queens problem, a well-known combinatorial challenge in computer science and artificial intelligence. The original N-Queens problem requires placing n queens on an $n \times n$ chessboard such that no two queens threaten each other. This includes avoiding conflicts in rows, columns, and both diagonals. Numerous solutions have been proposed for the N-Queens problem, most notably backtracking algorithms, which systematically explore the solution space and prune invalid configurations early.

Beyond backtracking, several studies have applied constraint satisfaction problem frameworks to the N-Queens problem. These approaches use constraint propagation techniques such as forward checking and arc consistency to reduce the number of viable assignments at each step. Such techniques are particularly effective for problems where domain reduction plays a significant role in pruning the search tree.

The Queens puzzle implemented in this work introduces an added layer of complexity by incorporating irregular jigsaw-like regions, each of which must contain exactly one queen, as well as a proximity restriction that forbids any queen from being placed within a one-cell radius of another. These modifications transform the problem into a hybrid between the N-Queens puzzle and jigsaw Sudoku, where positional constraints are different by area.

IV. Method

To solve the Queens puzzle, this study utilizes the A* algorithm as a method for finding a valid and complete queen placement configuration under a set of constraints. A* is chosen due to its ability to efficiently explore the state space using a combination of actual cost and heuristic estimates to guide the search toward optimal or feasible solutions.

Each state in the search space represents a partially filled board, where some queens have already been placed. The goal is to reach a final state where the board is fully filled with queens, while satisfying all three of the puzzle's constraints: no two queens share the same row or column, no queen is within a one-block radius of another, and each irregular area contains exactly one queen.

A. State Representation

Queens puzzle board is represented using a two-dimensional array of characters, where each cell corresponds to a square on the puzzle grid. The array structure allows for efficient access, modification, and checking of queen placements during the search process.

To represent the irregular area grouping required by the puzzle, a separate two-dimensional integer array is used. Each cell in this array contains a number indicating the area to which that cell belongs. For example, the area configuration:

```
22288330
22228830
24433333
66473711
66473711
66477715
64446555
66666655
```

Image 4. Example of Board Representation , taken from [2]

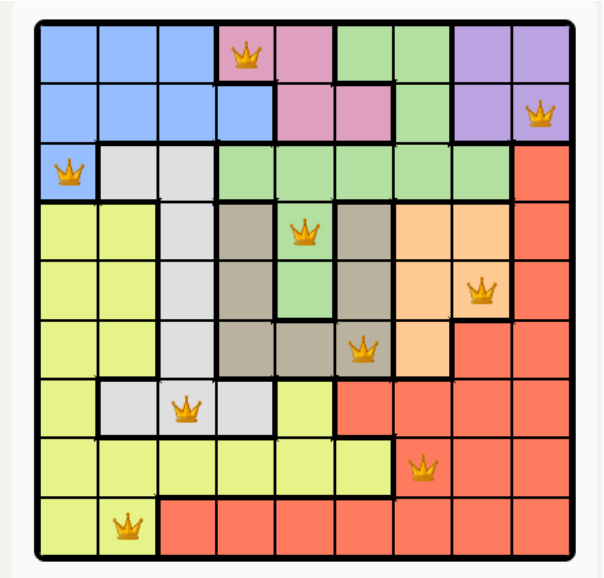

Image 5. Original Board , taken from [1]

The main board used during A* execution is a character-based 2D array with the following symbol conventions:

- 'Q' denotes a cell currently occupied by a queen,
- 'X' represents a forbidden cell (due to proximity or conflict),
- '.' or ' ' (space) denotes an empty, available cell.

By combining these two data structures, one for the queen placements and one for the area labeling the algorithm can efficiently check constraint violations, calculate heuristic values, and generate successor states.

B. Constraints Handling

In the implementation of the Queens puzzle solver using the A* algorithm, constraints are enforced through a combination of board updates and validation checks defined within the Board_state class. Each candidate queen placement undergoes multiple checks to ensure it complies with the puzzle's rules before the resulting state is expanded and added to the priority queue. The main constraints are as follows:

The three core constraints are handled as follows:

1) Row and Column Exclusivity

```python
def is_valid_placement(self, coordinate) -> bool:
    """Checks valid placement of coordinate on board"""
    for i in range(self.row):
        if self.board[coordinate[0]][i] == "Q":
            return False
    for i in range(self.col):
        if self.board[i][coordinate[1]] == "Q":
            return False

    if (
        self.board[coordinate[0]][coordinate[1]] == "Q"
        or self.board[coordinate[0]][coordinate[1]] == "X"
    ):
        return False

    return True
```

Image 6. is_valid_placement function , taken from [2]

To ensure that no two queens occupy the same row or column, the is_valid_placement() method checks the entire row and column of the candidate cell for any existing queens. If a queen is found, the method returns False, disallowing the move.

2) One-Block Radius Rule

```python
def place_queen(self, coordinate):
    """Adds queen to board as well as blocks the next invalid placements"""

    if not self.is_valid_placement(coordinate):
        return

    area_id = self.get_area_id(coordinate)
    if not self.is_valid_area(area_id):
        heapq.heappop(self.list_of_areas)

    # Fills x and y of the coordinate with X
    for i in range(self.col):
        self.board[coordinate[0]][i] = "X"
    for j in range(self.row):
        self.board[j][coordinate[1]] = "X"

    # Fills around the coordinate with X
    if coordinate[0] > 0 and coordinate[1] > 0:
        self.board[coordinate[0] - 1][coordinate[1] - 1] = "X"

    if coordinate[0] < self.row - 1 and coordinate[1] > 0:
        self.board[coordinate[0] + 1][coordinate[1] - 1] = "X"

    if coordinate[0] > 0 and coordinate[1] < self.col - 1:
        self.board[coordinate[0] - 1][coordinate[1] + 1] = "X"

    if coordinate[0] < self.row - 1 and coordinate[1] < self.col - 1:
        self.board[coordinate[0] + 1][coordinate[1] + 1] = "X"

    area = self.get_area(area_id)
    for coor in area.list_of_coordinates:
        self.board[coor[0]][coor[1]] = "X"

    self.board[coordinate[0]][coordinate[1]] = "Q"
```

Image 7. place_queen function, taken from [2]

After placing a queen using the place_queen() method, the surrounding 8 adjacent cells (both diagonally and orthogonally) are marked as "X" to indicate that they are no longer valid for future queen placement. This prevents violations of the one-cell-radius rule.

3) One Queen per Area

```
def is_valid_area(self, area_id) -> bool:
    """Checks if areas is full by X"""
    area = self.get_area(area_id)
    coor_x_count = 0

    for coor in area.list_of_coordinates:
        if self.board[coor[0]][coor[1]] == "X":
            coor_x_count += 1
    return coor_x_count <= area.size
```

Image 8. is_valid_area function , taken from [2]

Each cell belongs to a specific irregularly shaped area, represented by the Area class. The get_area_id() function maps a coordinate to its area. After a queen is placed, all other cells in the same area are marked "X" to ensure that only one queen is ever placed in that area. The is_valid_area() method checks that an area has not been fully blocked before attempting to place a queen in it, thus enforcing this constraint during search expansion.

4) Validity Checks for State Expansion

Before a state is accepted into the A* queue:

a) is_valid_placement() ensures the move doesn't violate any rules at the point of placement.

b) is_valid_area() ensures that placing a queen in the target area won't make it impossible to solve (i.e., fully block the area).

c) is_valid_board() can be used as an extra check to ensure global consistency when needed.

Additionally, the number of queens placed is tracked with get_queen_count(), and filled_count is used as part of the priority queue comparison logic via the __lt__() method to guide the A* search.

Together, these constraint mechanisms ensure that only legal and promising states are expanded, significantly reducing the search space and allowing A* to converge more efficiently toward a valid solution.

C. *Heuristic Function*

In the A* search algorithm, the heuristic function $h(n)$ plays a crucial role in guiding the search toward promising board configurations. It provides an estimate of the cost from the current state n to a goal state, allowing the algorithm to prioritize more favourable paths.

For the Queens puzzle, the goal is to place a queen in every area while satisfying all placement constraints. To reflect this, the heuristic function is designed to estimate the number of placements remaining and potential constraint violations that may occur as the board fills.

In this implementation, the heuristic is implicitly encoded using the filled_count and step variables in the Board_state class. The filled_count represents the number of valid placements on the current board, while step represents the number of moves made so far.

The priority queue (min-heap) used by the A* algorithm orders board states based on the following evaluation function:

$$f(n) = g(n) + h(n)$$

Where:

$g(n)$ = step is the actual cost from the start state to the current state,

$h(n)$ = filled_count is used as a heuristic estimate of how close the board is to being complete.

This means that the algorithm prioritizes states that have more queens successfully placed, while also considering how many moves have been made to reach that state.

D. *Successor State Generation*

In the A* search algorithm, generating successor states is a key step in exploring the search space. For the Queens puzzle, each successor state represents a new board configuration that results from placing an additional queen in a valid position. The quality and efficiency of this process directly influence the overall performance of the algorithm.

In this implementation, successor states are generated by iterating through all available cells on the board and attempting to place a queen in each one. The place_queen() method is responsible for updating the board when a valid placement is found, while preserving the problem's constraints.

```
found = False
while len(pq) > 0:
    current_board_state = heapq.heappop(pq)
    isFinished = current_board_state.is_finish()
    if isFinished:
        found = True
        break
    queen_count = current_board_state.get_queen_count()
    area = current_board_state.list_of_areas[queen_count]
    for coor in area.list_of_coordinates:
        if current_board_state.is_valid_placement(coor):
            new_board = copy.deepcopy(current_board_state.board)
            new_areas = copy.deepcopy(current_board_state.list_of_areas)
            new_state = Board_state(
                new_board, new_areas, current_board_state.step + 1
            )
            new_state.place_queen(coor)
            if new_state.is_valid_board:
                new_state.set_filled_count()
                heapq.heappush(pq, new_state)

if not found:
    print("No Solution")
    return []
else:
    return current_board_state.get_queen_coordinates()
```

Image 9. Main Algorithm , taken from [2]

Steps to Generate Successors:

1) Iterate Through the Board
The algorithm examines each cell in the 2D array to determine whether it is empty (i.e., not marked with 'X' or 'Q').

2) Constraint Checking
For each candidate cell, the is_valid_placement() function is called to verify that placing a queen would not violate the following rules:

   a) No other queen exists in the same row or column.

   b) The cell is not already marked as 'X'.

3) Area Validation
If the basic constraints pass, the get_area_id() function retrieves the area ID of the candidate cell. The is_valid_area() function is then used to check whether a queen has already been placed in that area. If the area already contains a queen, the cell is skipped.

4) State Creation
If all checks pass, a deep copy of the current Board_state object is made. The queen is placed on the copied board using place_queen(), which also marks forbidden cells. The new state is then added to the A* open set for further evaluation.

5) Priority Evaluation
The new board state's priority is calculated based on the combined value of filled_count and step. The priority queue automatically orders these states using the custom __lt__() method.

E. End State

The end state in the Queens puzzle represents a fully completed board configuration where all constraints are satisfied and no further queen placements are required. Determining whether a board has reached a valid end state is a critical step in the A* algorithm, as it serves as the stopping condition for the search process.

A board is in an end state if the following conditions are met:

1) All areas contain exactly one queen
The total number of queens placed on the board must equal the number of unique area regions. Each area is checked to ensure that it contains exactly one queen and that no conflicting placements exist within that region.

2) No two queens threaten each other
This includes validation that:

   a) No queens share the same row or column,

   b) No queen is placed within a one-cell radius of another queen,

3) No empty or unprocessed spaces remain
All non-queen cells on the board must either be marked as forbidden ('X') or belong to a region that already contains a queen. If any empty cell (' ') remains in a region without a queen, the state is considered incomplete.

```python
def is_finish(self) -> bool:
    """Checks if board is finish state"""
    for row in self.board:
        if row.count("X") == self.col or row.count(" ") > 0:
            return False

    for area in self.list_of_areas:
        if not self.is_valid_area(area.id):
            return False
    return True
```

Image 10. is_finish function, taken from [2]

This validation is handled programmatically by the is_finish() method in the Board_state class, which performs a final consistency check over rows, columns, regions, and cell contents. Only when all these checks pass is the current board configuration accepted as a valid solution

V. Conclusion

This paper presented an approach to solving the Queens puzzle using the A* search algorithm, adapted to accommodate unique constraints such as one-queen-per-area, no queens in the same row or column, and a one-cell radius restriction. By modelling the puzzle as a state-space search problem, A* was used to efficiently explore valid configurations through a combination of actual cost (number of moves made) and heuristic estimation (progress toward completion).

The algorithm successfully finds valid queen placements by expanding only those states that satisfy all given constraints. Object-oriented design, such as the use of Area and Board_state classes, ensured modular, readable, and extensible code that can easily support further enhancements.

However, although the implementation using A* works as intended, it is not necessarily the most optimal algorithm for this specific problem. Unlike traditional pathfinding tasks where the sequence of steps matters, the Queens puzzle is fundamentally a placement and constraint satisfaction problem. Since the path taken is irrelevant as long as the final configuration is valid a backtracking approach may be more natural, efficient, and easier to reason about. Backtracking directly explores valid configurations recursively, pruning invalid paths early without the need for heuristic estimation or priority queues.

In conclusion, while this implementation demonstrates how classical search algorithms like A* can be applied to constraint-heavy logic puzzles, future work could explore and compare other strategies such as backtracking or constraint propagation to improve performance and simplicity.

Video Link at Youtube

https://youtu.be/15a62zyToXQ

ACKNOWLEDGMENT

I would like to express my sincere gratitude to God for His guidance and strength throughout the completion of this paper. I am deeply thankful to my family for their constant support, encouragement, and belief in my academic journey. I also extend my heartfelt appreciation to my lecturers Dr. Ir. Rinaldi Munir, M.T and Dr. Nur Ulfa Maulidevi, S.T, M.Sc., whose expertise and dedication in the fields of algorithms and artificial intelligence have been a constant source of inspiration. Special thanks to my friends and peers, whose discussions, feedback, and shared moments of problem-solving greatly enriched the development of this work. Lastly, I hope this paper serves as a useful contribution to the study of algorithms and puzzle-solving and encourages further exploration into the creative applications of classical techniques like A* in modern problem spaces.

REFERENCES

[1] LinkedIn, "Queens," LinkedIn Games, 2024. [Online]. Available: https://www.linkedin.com/games/queens/. [Accessed: Jun. 20, 2025].

[2] B. Kunanda, "Queens Solver Using A*," GitHub, 2025. [Online]. Available: https://github.com/BobSwagg13/Queens_Solver_Using_ASTAR. [Accessed: Jun. 20, 2025].

[3] GeeksforGeeks, "N-Queen Problem | Backtracking-3," [Online]. Available: https://www.geeksforgeeks.org/dsa/n-queen-problem-backtracking-3/. [Accessed: Jun. 20, 2025].

[4] R. Munir, "Route Planning and A* Algorithm," *Strategi Algoritma*, Institut Teknologi Bandung, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf. [Accessed: Jun. 20, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Juni 2025

Bob Kunanda 13523086