Application of Search Algorithms in Solving LinkedIn Queens Puzzles

Caleb Petterson
Shivani Sairam
pette184@umn.edu
saira008@umn.edu

Abstract

The LinkedIn Queens puzzle challenges players to place N queens on an N x N grid such that each row, column, and colored region contains exactly one queen. This experiment evaluates the performance of five algorithms, Breadth-First Search, Depth-First Search, Local Search, Backtracking, and Genetic Algorithm Search, in solving these puzzles. The algorithms were assessed based on their completion rates, time efficiency, and space usage across various board sizes. The results showed that Backtracking was the most effective algorithm for solving the LinkedIn Queens puzzle, consistently solving puzzles with low memory usage and reasonable speed.

1 Introduction

This project focuses on solving a puzzle called the LinkedIn Queens game, which is a creative variation of the classic N-Queens problem. The traditional N-Queens puzzle is a well-known problem in which the goal is to place N queens on an $N \times N$ chessboard so that no two queens can attack each other. This means that no two queens can be in the same row, column, or diagonal.

The LinkedIn Queens game adds a new twist to this challenge by introducing additional rules. In this version of the puzzle, the chessboard is divided into several colored regions, and the goal is to place N queens on the board in such a way that:

- Each row contains exactly one queen.
- Each column contains exactly one queen.
- Each colored region contains exactly one queen.
- No queens are adjacent vertically, horizontally, or diagonally to each other.

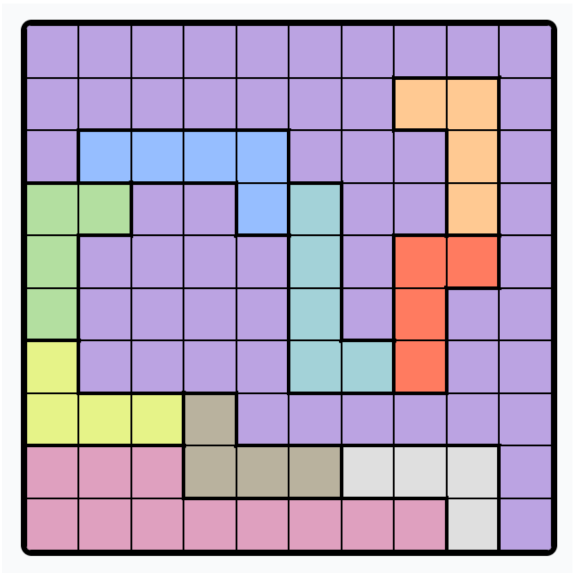

Figure 1. Unsolved 10x10 Linkedin Queens puzzle showing initial board state before any queens are placed

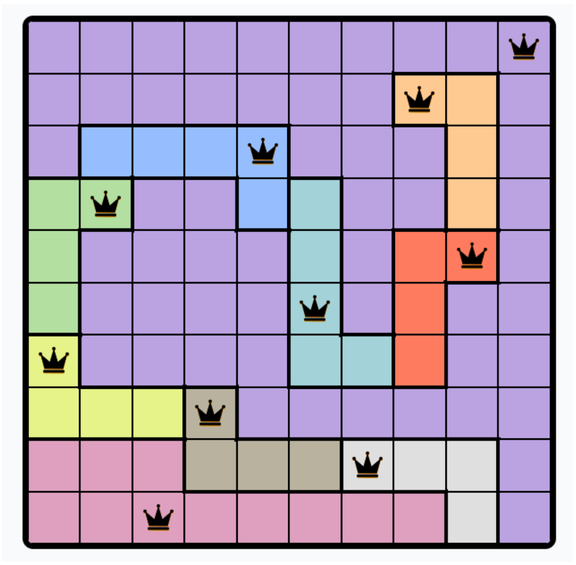

Figure 2. Solved 10x10 LinkedIn Queens puzzle where queens are placed in valid positions following all constraints

Unlike the traditional N-Queens problem, the LinkedIn Queens game does not have diagonal restrictions. However, the new rule requiring exactly one queen per colored region makes the puzzle much more difficult. Players must think about both the usual row and column constraints and the added restriction of the colored regions. These added rules make the puzzle more difficult to solve and more interesting to study.

2 Related Work

The N-queens problem is a well-known puzzle and problem in the field of artificial intelligence (AI), where it has been used extensively as a benchmark for evaluating search algorithms. This literature review will touch on significant evolutions in applying the N-queens problem, highlighting various innovative approaches. These include a novel probabilistic local search algorithm, divide-and-conquer techniques, neural network models, genetic algorithms, quantum algorithms, Monte Carlo search methods, non-backtracking methods, blind search approaches, and others. Each method's efficiency toward the N-queens problem are discussed.

2.1 The N-Queens Problem

The N-queens problem, which involves placing N queens on an N x N chessboard such that no two queens can attack each other, has long served as a benchmark in AI for evaluating search algorithms due to its complex state space. Although now mostly used as an exercise in introductory AI courses, it remains a staple benchmark for backtracking and search algorithms at large [8].

2.2 Blind Search Approaches

Research into blind search approaches proposed a combined Depth-First Search and Breadth-First Search approach. This method directly places queens on the chessboard following a regular pattern, demonstrating improved performance and runtime compared to backtracking and hill climbing methods; additionally, it also lowered the time and space complexity of solving N-queens problems when compared to older methods [11].

2.3 Backtracking and Basic Methods

One of the simplest ways to solve the N-Queens problem is by using backtracking. This method is works by trying possibilities out one at a time and pivoting backward if one fails. It works by placing a queen in the first row and then moving on to the next row, attempting to place the next queen in a safe spot. If the algorithm finds a conflict (i.e. two queens are attacking each other), it will go back to the previous row and move that queen to a new position. This process continues until all queens are placed safely.

While this method is simple, it can be slow for larger problems. This is because the number of possible placements increases rapidly as the number of queens increases. To improve this, some versions of backtracking use forward checking. Forward checking helps by eliminating certain options that are already known to be invalid before trying them, making the search process faster. But even with these improvements, backtracking can still be too slow for very large problems, like when the number of queens is over 100 [2].

2.4 Local Search Techniques

Another way to solve the N-Queens problem is by using local search methods. In this approach, the algorithm starts with an initial arrangement of queens and then make small changes to improve the solution. For example, it might try moving a queen to a different column in the same row if that helps reduce conflicts. If the new arrangement produces fewer conflicts, the algorithms selects the new state.

2.4.1 Simulated Annealing. One popular local search method is simulated annealing. This technique is inspired by the way metals cool and harden. When metals cool slowly, they have a better chance of forming a stable structure. Similarly, simulated annealing starts by allowing big random changes to the solution and then slowly reduces the size of those changes as it gets closer to an optimal solution. This helps the algorithm avoid getting stuck in a local optimum, where the solution is good but not the best possible solution. By allowing suboptimal moves at the beginning, the algorithm can explore more possibilities and find a better solution later [7].

2.4.2 Probabilistic Local Search. A recent advancement in solving the N-queens problem is the probabilistic local search algorithm that leverages a gradient-based heuristic. This approach excels at finding solutions for extremely large N-queens problems, handling up to 500,000 queens. The algorithm's efficiency is demonstrated through execution statistics, highlighting its substantial improvement over traditional methods, which often rely on backtracking and suffer from exponential time complexity [12].

2.5 Genetic Algorithm Search

A meta-heuristic approach introduces the use of adaptive genetic algorithm alongside heuristics for solving the N-queens problem. This approach employs a novel fitness function alongside genetic operations like mutation and crossover. The genetic algorithm demonstrates promising results for large values of N, showcasing improved performance and efficiency over previous methods. Its high convergence rate and reduced computation time make it a valuable tool for optimization in complex systems [10].

2.6 Quantum Algorithms

Two innovative quantum algorithms have been applied to solving the N-queens problem: the Direct Column Algorithm and the Quantum Backtracking Algorithm. These algorithms utilize controlled W-states and dynamic circuits to efficiently address the N-queens problem. Despite the exponential circuit complexity, the Direct Column Algorithm strategically reduces the search space, while the Quantum Backtracking Algorithm emulates classical backtracking techniques within a quantum framework. These quantum approaches showcase significant potential for solving complex combinatorial problems with applications in satellite communication, routing, and even VLSI testing [9].

2.7 Divide-and-Conquer Algorithm

An alternative approach to the traditional N-queens problem uses a divide-and-conquer algorithm. This method addresses both the N-queens problem and the related problem of arranging N queens on a toroidal chessboard. The divide-and-conquer technique significantly speeds up the solution process compared to conventional backtracking algorithms, offering faster solutions in both sequential and parallel computing environments [1].

2.8 Monte Carlo Simulations for Counting Solutions

Another method to solve the N-Queens problem, especially for large boards, is Monte Carlo simulations. This method uses randomness to estimate solutions instead of trying to check every possible arrangement. Instead of placing the queens one by one and checking each possibility, Monte Carlo methods try many different arrangements and use probability to guess how many valid solutions there are. This is much faster when the problem is very large, because every possible solution does not need to check .

Monte Carlo simulations were used to estimate the number of solutions for large N-queens puzzles, even when the number of queens was as large as 10,000. The method treats the problem like a "thermodynamic system," where more stable configurations (fewer conflicts) are considered better. By sampling different configurations at various "temperatures," the method can estimate the number of solutions more efficiently, even for very large puzzles [13].

2.9 Non-Backtracking Algorithm

A paper by Marian Kovac, presented at the 10th International Conference on Digital Technologies, introduced an efficient non-backtracking algorithm using local search, heuristics, and Tabu search elements. This algorithm can find solutions for massive instances of the N-queens problem (hundreds of millions of

N) in a short time, even on ordinary personal computers. Its performance surpassed previously known methods, establishing it as the fastest algorithm in the N-queens bibliography [4].

2.10 Neural Network Approaches

Other research explores the application of neural networks to solving combinatorial optimization problems, with a specific focus on the N-queens problem. The paper discusses various improvements to the Hopfield models, including deterministic, stochastic, chaotic, hybrid, and digitally feasible modifications. These models, benchmarked against the N-queens problem, show high convergence rates and low experimental time requirements, making them effective solutions for the problem [5].

2.11 Non-Sequential Conflict Resolution (NSCR)

A new algorithm called Non-Sequential Conflict Resolution (NSCR) has been created to solve the N-queens problem more efficiently. Traditional methods like backtracking often struggle with large problems because they use a lot of time and memory. NSCR works by adjusting the position of queens when conflicts happen, instead of immediately going back and trying a different path. This saves time and resources.

NSCR checks multiple positions for each queen and picks the one with the least conflict, making it faster and better with memory, especially for bigger problems. This approach can also help solve more complicated problems, like the LinkedIn Queens problem, where queens must be placed on different colored squares with extra rules. NSCR's ability to adjust in real-time makes it a good choice for handling large problems [6].

2.12 Hybrid Approaches

Recently, researchers have combined different methods to solve the N-Queens problem more effectively. For example, a hybrid approach might use a genetic algorithm to quickly search for possible solutions, then apply local search techniques to fine-tune those solutions. This combination takes advantage of both broad exploration (finding many solutions) and detailed refinement (improving the best ones). Hybrid methods are especially useful for large N-Queens puzzles because they can explore many possibilities and then focus on improving the best ones [3].

3 Problem Statement

The LinkedIn Queens puzzle builds on the traditional N-Queens problem, where the objective is to place N queens on an N x N chessboard such that no two queens threaten each other. The additional constraints of the LinkedIn Queens game introduce complexity:

- Each row contains exactly one queen.
- Each column contains exactly one queen.
- Each colored region contains exactly one queen.
- No queens are adjacent vertically, horizontally, or diagonally to each other.

The challenge lies in adapting classical algorithms used for solving N-Queens puzzles to this extended problem. Unlike the standard N-Queens problem, which focuses on row, column, and diagonal conflicts, the LinkedIn Queens puzzle demands consideration of region-based constraints. This transforms the puzzle into a novel constraint satisfaction problem (CSP) that tests the capabilities of traditional and advanced search algorithms.

The implications of solving this puzzle extend beyond theoretical exploration. Its similarities to real-world problems like resource allocation, scheduling, and planning makes it an ideal vehicle for testing algorithms' performance in CSPs across various metrics. By analyzing the performance of algorithms like Breadth-First Search, Depth-First Search, Local Search, Backtracking Search, and Genetic Algorithm Search, this study seeks to identify search algorithms that balance computational efficiency and accuracy.

More specifically, each algorithms' time complexity, space complexity, and completion ability will be measured to determine which apply best toward solving LinkedIn Queens puzzles.

3.1 Significance of Problem

It Models Real-Life Problems. Many real-world problems involve dealing with multiple rules or restrictions at once. For example, scheduling tasks, managing resources, or creating efficient plans all require finding solutions that follow specific rules. The LinkedIn Queens puzzle is similar because it needs a solution that works within several limits. This makes it a good example of a CSP, where the goal is to meet all the rules while solving the challenge. Understanding this puzzle provides insights into handling real-world challenges that involve balancing multiple conditions.

It is Harder Than the Original Puzzle. The addition of colored regions creates a new layer of difficulty that does not exist in the original N-Queens problem. Traditional solutions for the N-Queens problem, such as placing queens to avoid rows, columns, and diagonals, cannot simply be reused here. Solving this new puzzle requires more advanced techniques and adaptations of algorithms.

It Explores Algorithms. The Linked-In Queens puzzle can be solved using a variety of search algorithms, including simple methods like Breadth-First Search and Depth-First Search, complex approaches like Backtracking Search, and even more advanced techniques like Genetic Algorithm Search and Local Search. Testing these different algorithms reveals which ones work best for this unique problem. Comparing these methods also helps understand their strengths and weaknesses.

Our project will be a platform for exploring the unique use cases of each algorithm in terms of its time complexity, space complexity, and solving efficiency. Due to the N-Queens Problem's nature as a varying state space, our paper will lay the foundation for future assessment of diverse state space conditions.

It Teaches Important Skills. This project is not just about solving a puzzle, it is also a learning opportunity. It shows well-known search algorithms, like those used for the classic N-Queens problem, can be adapted to work for a new and more complicated challenge. This is a valuable skill for anyone working in Computer Science, as it is often necessary to modify existing solutions to fit new problems.

In this project, different algorithms will be tested and compared to solve the LinkedIn Queens puzzle. How fast and efficiently each one finds a solution while following all the rules will be shown. This will help understand which algorithms work best and why. The project combines problem-solving, designing algorithms, and analyzing their performance, showing how computational thinking helps solve complex problems in a clear and practical way.

3.2 Hypothesis

We hypothesize that the Backtracking algorithm will effectively solve the LinkedIn Queens puzzle and perform better than the other proposed search algorithms, just as it performs well in the traditional N-Queens problem. We expect it to consistently find solutions while satisfying all puzzle constraints, including region-based restrictions.

4 Methodology

To address this challenge, many components had to be conceptualized and created, from code to figures. Various search algorithms and metrics were considered. In addition, a state space representation was constructed to retain the semantic information of the puzzle necessary for deriving a solution, which is used by the search algorithms. The methodology behind this experiment is detailed below.

4.1 Search Algorithms

The implementations for all algorithms will be sourced from previous work on the N-Queens problem. An in-depth explanation of each algorithm as it applies in an N-Queens context is discussed in this section. Additionally, potential modifications necessary for implementation in solving LinkedIn Queens are detailed.

4.1.1 Breadth-First Search & Depth-First Search.

Breadth-First Search and Depth-First Search are simple algorithms that look at all the ways to place the queens on the board. Breadth-First Search checks all possible queen placements one layer at a time, making sure to explore every option at the current level before moving on to the next. This can help find the shortest solution, but it uses a lot of memory to keep track of all the placements. Depth-First Search, on the other hand, goes down one path of placements all the way to the end before backtracking and trying another path. It does not require as much memory as Breadth-First Search, but it does not always find the shortest solution. Both Breadth-First Search and Depth-First Search work well for smaller or simpler problems as they are guaranteed to find a solution if there is one.

However, for the LinkedIn Queens puzzle, which has many possible placements due to the board size and color rules, these methods can be slow and inefficient since they need to check so many paths. To make them more efficient, heuristics could be implemented, which guide the algorithms to focus on the moves that are more likely to lead to a solution. This can help save time by reducing unnecessary checks and concentrating on the best arrangements.

4.1.2 Local Search.

The Local Search Algorithm starts with a rough placement of queens on the board and then gradually improves it instead of checking every possible arrangement. It begins with a random setup and makes small changes to reduce conflicts, like moving a queen if it is in the same row as another, aiming to reduce rule violations with each adjustment. However, a big limitation is the local optimum, which is when it gets stuck in a position that partly solves the puzzle but does not meet all the rules. In order to tackle this, simulated annealing will be used, which allows the algorithm to employ "bad" moves in order to escape tricky spots, or have random restarts where it resets with a new starting arrangement when it reaches an impasse.

4.1.3 Backtracking.

A backtracking algorithm takes a step-by-step approach where queens are placed on the board one at a time. If a placement breaks any rules (like having two queens in the same row or putting a queen on the wrong color), the algorithm "backtracks," going back to the last successful placement and trying a new option. This approach checks all possible placements in a logical order, ensuring that every valid arrangement is considered. Backtracking is dependable and will eventually find the correct solution if one exists. However, it is slow for larger boards or puzzles with many rules, like the LinkedIn Queens Puzzle. Each time a placement fails to meet all the rules, the algorithm has to backtrack and try a new path, which can lead to a lot of repeated checking. One way to make it better is by eliminating choices that are clearly wrong early on.

4.1.4 Genetic Algorithm.

A genetic algorithm is based on natural evolution and starts with a population of random potential solutions with random arrangements of the queens. Each arrangement is given a score of fitness or how well it meets the puzzle's rules. The best scoring arrangements are selected to create the next generation of solutions, which includes crossover of the elements from the best scoring solution and also mutations, which are random changes to keep things diverse. The genetic algorithm is good for handling big, complex problems as it does not need to follow rigid rules and can explore a wide range of possibilities quickly. A genetic algorithm is not guaranteed to find the best solution but can often find a workable one faster than other methods.

4.2 Metrics

4.2.1 Time Complexity.

The time complexity, measured in seconds(s), measures how long it takes for each algorithm to find a solution to the puzzle. Time is an important factor because shows how efficiently an algorithm can solve the puzzle, especially as the puzzle size increases. Some algorithms may solve the puzzle faster than others. The time it takes can vary depending on the algorithm used. For example, algorithms like Breadth-First Search and Depth-First Search may take longer as the size of the board grows, because they explore all possible options before reaching a solution. On the other hand, algorithms like Local Search may be faster because they make gradual improvements to a starting solution without checking every possibility. This metric will help in comparing how quickly each algorithm can solve the puzzle.

4.2.2 Space Complexity.

The space complexity, measured in megabytes (MB), measures how much memory the algorithm uses while trying to solve the puzzle. This is important because some algorithms need more memory to store the data about different board configurations and their possible moves. For example, Breadth-First Search needs a lot of memory because it stores all possible configurations at each level of the search tree, which can grow large very quickly, especially for bigger puzzles. On the other hand, Depth-First Search uses less memory because it only needs to store the current path it is exploring, but it may need to explore many paths.

In this project, it is important to understand how well each algorithm can handle large grids without using too much memory. If an algorithm uses too much memory, it could crash or take up too many resources, making it unsuitable for larger problems. This metric will help in comparing the memory usage of different algorithms, and it will be measured in megabytes.

4.2.3 Completion.

The completion metric, measured by accuracy in decimal form, tracks how successfully an algorithm can solve the puzzle. A solution is considered complete when all queens are placed on the board following the puzzle's rules. These rules include things like ensuring no two queens are in the same row, column, or color region, and that no two queens are adjacent, including diagonally.

Algorithms like Local Search and Genetic Algorithm Search may struggle with completion. Local Search often gets stuck at local optima, where it cannot make progress toward the full solution, requiring random restarts to try again. Similarly, Genetic Algorithm Search, which explores multiple potential solution, may not always find the valid solution within the number of iterations and fail to complete the puzzle.

As the board size increases, Breadth-First Search and Depth-First Search might also struggle with completion due to high memory usage and inefficient path exploration. Breadth-First Search consumes excessive memory, while Depth-First Search becomes slower and less reliable. In contrast, Backtracking is more efficient for larger puzzles, as it prunes invalid paths early and avoids redundant searches, making it the most reliable algorithm for larger grid sizes.

4.3 State Space Representation

The state space for a LinkedIn Queens grid must contain multiple elements. The data structure must keep track of each tile's location in terms of row and column, while also defining the color region it is associated with. Additionally, the placement of queens needed to be tracked for algorithms such as local search.

As such, a 2D array of tuples was devised, in which each tuple contained two numbers:

- **is_queen:** Indicates whether a queen is placed in the cell (1 for yes, 0 for no).
- **region_id:** Identifies the region (or color) associated with the cell.

4.4 Data Collection

In order to conduct experiments, data needed to be collected pertaining to LinkedIn puzzle grids. The form of the data had to align easily with the state space representation; as such, raw images of LinkedIn puzzles were collected and fed through a processing pipeline to convert them into usable state spaces.

Puzzles of various grid sizes were collected and grouped. Since performance of each search algorithm would likely differ depending on the grid size of a puzzle, it was determined that 5 separate grid sizes should be accounted for in the experiment. Luckily, LinkedIn Queens puzzles exist in 5 different sizes: 7x7, 8x8, 9x9, 10x10, and 11x11. However, only 2 11x11 have been released to date, which was accounted for by reducing the number of trials to 4 and including 11x11 as an extra trial for observation only. Since the data for 11x11 was limited, it opened the possibility for less accurate results, so it was marked differently in experiments and distinguished in plots and results.

4.5 Experiment Design

The experiment design focuses on evaluating the performance of Breadth-First Search, Depth-First Search, Local Search, Backtracking Search, and Genetic Algorithm Search in solving the LinkedIn Queens puzzle. The experiments were conducted on grids of varying sizes, from 7x7 to 11x11, to analyze the scalability and efficiency of each algorithm under increasing complexity. Metrics such as execution time, memory usage, and solution completeness were measured to assess the algorithms' strengths and weaknesses. These metrics provide insights into which algorithms are best suited for specific grid sizes and constraint configurations.

4.5.1 Trial Methodology. For each trial, an initial puzzle state was generated based on the grid size and region constraints extracted from 5 input images. Trials ran 5 puzzles of the same grid size to account for variations in runtime and memory usage. Each algorithm attempted to solve all 5 puzzles while their execution time and memory usage were tracked. Results were averaged across the runs to produce consistent metrics for execution time, memory usage, and solution completeness.

4.5.2 Plot Methodology. The results of the trials were visualized through comparative plots to illustrate the performance trends across different grid sizes. Line graphs were used to represent execution time and memory usage, with each algorithm assigned a unique color for clarity. Bar charts depicted solution completion as a decimal, with 1.0 representing 100% accuracy and 0.0 representing 0% accuracy. Data points for each grid size were derived from the averaged results of five puzzles. These visualizations were designed to emphasize differences in algorithm efficiency, scalability, and performance.

4.6 Code Overview

The code consists of the following main components:

- **plotting.py:** Plots the performance of five search algorithms (Breadth-First Search, Depth-First Search, Local Search, Backtracking Search, and Genetic Algorithm Search) on puzzles of varying sizes (7x7 to 11x11).
- **screen_capture.py:** Facilitates capturing and processing puzzle images from external sources (e.g., screenshots).
- **search_algorithms.py:** Defines various puzzle-solving algorithms and the logic behind searching for solutions.
- **solver_experiment.py:** Serves as the main driver script for conducting experiments. It allows users to configure and test different algorithms, record performance metrics, and analyze results.
- **requirements.txt:** Lists the necessary Python packages and their versions required for the project to function correctly.
- **README.md:** Provides an overview of the project, instructions for setup, and guidelines for using the code.
- **.gitignore:** Ensures temporary or unnecessary files (e.g., cache or intermediate outputs) are excluded from version control.
- **Processed Puzzle Images:** A collection of preprocessed puzzle data for various grid sizes (7x7, 8x8, 9x9, 10x10, 11x11) to support experimentation and testing.

5 Results

Multiple experiment trials were conducted to evaluate the performance of each search algorithm on puzzles of different grid sizes. The following sections showcase figures and summarize the key findings across five puzzle sizes: 7x7, 8x8, 9x9, 10x10, and 11x11.

5.1 Performance Figures

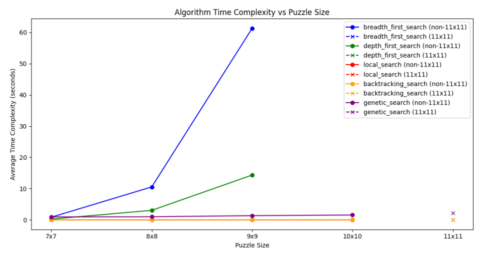

Figure 3. Graph of Time Complexity vs. Puzzle Size. For this figure, local_search is hidden behind genetic_search, as their timings aligned closely. Puzzle size 11x11 is isolated due to limited samples potentially making its result unreliable.

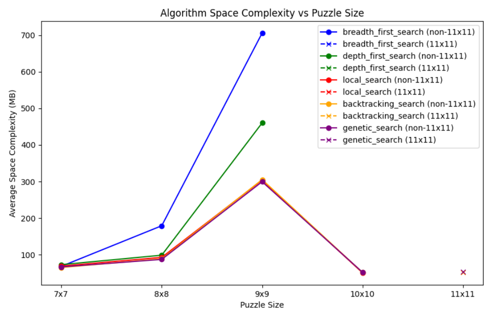

Figure 4. Graph of Space Complexity vs. Puzzle Size. local_search, backtracking_search, and genetic_search overlap, as their memory usage aligned closely. Puzzle size 11x11 is isolated due to limited samples potentially making its result unreliable.

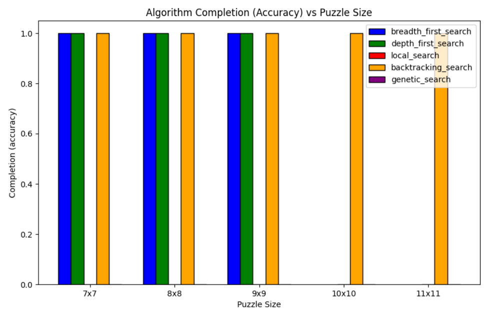

Figure 5. Histogram of Completion vs. Puzzle Size. Puzzle size 11x11 is not considered as a legitimate trial due to limited samples potentially making its result unreliable.

5.2 Trial 1: 7x7

For the 7x7 puzzles, all algorithms except for local_search and genetic_search successfully found solutions.

- **breadth_first_search** completed in an average time of 0.845476 seconds per image, with an average space complexity of 68.409375 MB per image, and achieved 100% accuracy.
- **depth_first_search** had a lower average time complexity of 0.245405 seconds per image, with a space complexity of 72.978125 MB per image, also achieving 100% accuracy.
- **local_search** failed to find a solution, resulting in a accuracy of 0%, though it showed a very low average time complexity of 0.001051 seconds per image and a space complexity of 69.081250 MB per image.
- **backtracking_search** found solutions for all puzzles, with a high average time of 0.001204 seconds per image and space complexity of 65.230469 MB per image, achieving 100% accuracy.
- **genetic_search** failed to find a solution for all five puzzles, with a time complexity of 0.938612 seconds per image, a space complexity of 66.853906 MB per image, and an accuracy of 0%.

5.3 Trial 2: 8x8

For the 8x8 puzzles, all algorithms except for local_search and genetic_search successfully found solutions.

- **breadth_first_search** took an average of 10.534963 seconds per image, with a space complexity of 179.224219 MB per image and 100% accuracy.
- **depth_first_search** showed a time complexity of 3.042529 seconds per image, with a space complexity of 98.814844 MB per image, achieving 100% accuracy.
- **local_search** once again failed to find a solution and had a time complexity of 0.000981 seconds per image, with a space complexity of 92.857812 MB per image and a accuracy of 0%.

- **backtracking_search** successfully solved all puzzles with a time complexity of 0.000789 seconds per image, a space complexity of 88.960156 MB per image, and a accuracy of 100%.
- **genetic_search** also failed to find a solution, with an average time of 0.984549 seconds per image and a space complexity of 87.332031 MB per image, resulting in a accuracy of 0%.

5.4 Trial 3: 9x9

For the 9x9 puzzles, all algorithms except for local_search and genetic_search successfully found solutions.

- **breadth_first_search** showed a significant increase in time complexity, taking 61.283823 seconds per image, with a space complexity of 705.515625 MB per image, and achieved 100% accuracy.
- **depth_first_search** had a time complexity of 14.325647 seconds per image and a space complexity of 460.087500 MB per image, with an accuracy of 100%.
- **local_search** failed to find a solution and recorded a time complexity of 0.001681 seconds per image and a space complexity of 303.792969 MB per image, with an accuracy of 0%.
- **backtracking_search** found solutions for all puzzles, with a time complexity of 0.001802 seconds per image, a space complexity of 305.300781 MB per image, and 100% accuracy.
- **genetic_search** failed to find a solution for all puzzles, with an average time complexity of 1.341594 seconds per image and a space complexity of 299.550000 MB per image, resulting in an accuracy of 0%.

5.5 Trial 4: 10x10

For the 10x10 puzzles, both breadth_first_search and depth_first_search failed due to memory limitations. Only backtracking_search was able to find solutions.

- **breadth_first_search** ran out of memory and could not be recorded.
- **depth_first_search** ran out of memory and could not be recorded.
- **local_search** failed to find a solution, with a time complexity of 0.001948 seconds per image, a space complexity of 50.601562 MB per image, and an accuracy of 0%.
- **backtracking_search** found solutions, with a time complexity of 0.001903 seconds per image, a space complexity of 51.511719 MB per image, and 100% accuracy.
- **genetic_search** failed to find solutions, with a time complexity of 1.559411 seconds per image and a space complexity of 52.136719 MB per image, leading to 0% accuracy.

5.6 Trial 5 (bonus): 11x11

Trial 5 is not treated as an official trial and exists for observation only. An interesting feature of this trial is that breadth_first_search and depth_first_search continue the trend of memory issues from the previous trial, highlighting the impact that larger search spaces have on blind search algorithms.

- **breadth_first_search** ran out of memory and could not be recorded.
- **depth_first_search** ran out of memory and could not be recorded.
- **local_search** had an average time complexity of 0.001109 seconds per image, an average space complexity of 52.177734 MB per image, and an accuracy of 0%.
- **backtracking_search** had an average time complexity of 0.001963 seconds per image, an average space complexity of 52.552734 MB per image, and an accuracy of 100%.
- **genetic_search** had an average time complexity of 2.146893 seconds per image, an average space complexity of 53.144531 MB per image, and an accuracy of 100%.

6 Analysis

The performance of each algorithm was evaluated based on its ability to solve the LinkedIn Queens puzzle, taking into account the time complexity, space complexity and completeness. This section provides a detailed breakdown of how each algorithm performed, highlighting the algorithms' strengths and weaknesses.

6.1 Breadth-First Search & Depth-First Search

Breadth-First Search and Depth-First Search are reliable for smaller puzzles since they will always find a solution if one exists. However, as the puzzle size increases, they struggled to complete in a reasonable time.

6.1.1 Breadth-First Search. Breadth-First Search is a reliable algorithm for solving puzzles, but as puzzle sizes increase, it struggles with both time and space complexity.

- **Space Complexity**: Breadth-First Search has the highest space usage among all the algorithms. It grows exponentially as the puzzle size increases due to the need to store all the nodes at the current level of the search tree. In the plots, Breadth-First Search's space usage shoots up to over 700 MB for a 9x9 puzzle and becomes unmanageable for larger sizes.
- **Time Complexity**: Breadth-First Search's time complexity also grows rapidly with puzzle size. It becomes slower as it explores all possibilities at each level before moving deeper. By 9x9, its time exceeds 60 seconds and becomes impractical for larger sizes.
- **Completeness**: Breadth-First Search will always find a solution if one exists, but its high space and time complexity make it impractical for larger puzzles, limiting its effectiveness for the LinkedIn Queens problem.

6.1.2 Depth-First Search. Depth-First Search is a more space-efficient option compared to Breadth-First Search, but it still faces challenges as puzzle size increases.

- **Space Complexity**: Depth-First Search is more space-efficient than Breadth-First Search because it only needs to store nodes along the current path. In the plot, its space usage is significantly lower than Breadth-First Search but still increases for larger puzzle sizes, reaching around 400 MB for a 9x9 puzzle.
- **Time Complexity**: Depth-First Search's time complexity is better than Breadth-First Search for smaller sizes but still increases as it explores deeper paths. For a 9x9 puzzle, it performs similarly to Breadth-First Search, but it could not handle anything larger in our tests.
- **Completeness**: Depth-First Search will always find a solution if one exists, but like Breadth-First Search, its performance suffers with larger puzzles. It is slower and less efficient for larger puzzle sizes, making it unsuitable for solving larger LinkedIn Queens puzzles.

6.2 Local Search

Local Search shows consistently low memory usage since it does not store a full search tree. Instead, it works on a single solution at a time, trying to improve it step by step.

- **Space Complexity**: Local Search shows consistently low memory usage in the plots, staying near 100 MB across all puzzle sizes. This is because it does not store a full search tree but works on a single solution at a time.
- **Time Complexity**: Despite being memory-efficient, Local Search fails to complete any puzzle size reliably in the time complexity plot. This is likely due to the algorithm getting stuck in local optima, where it cannot find a better solution without further exploration.
- **Completeness**: Local Search is not guaranteed to find a complete solution. The algorithm often gets stuck in local optima, where it cannot improve the current solution despite a better one existing. As a result, it frequently fails to complete the puzzle, especially for larger sizes. This limitation can be mitigated by using techniques such as random restarts, but the algorithm remains unreliable for large puzzle sizes.
- **Partial Solutions**: Although Local Search does not always find a full solution, it

often returns partial solutions. For instance, it may correctly place a few queens on the board, but fail to solve the puzzle completely due to being stuck at a local optimum. While these partial solutions can provide useful insights or serve as a starting point for other methods, Local Search does not guarantee a complete configuration of queens.

6.3 Backtracking

Backtracking is the only algorithm able to complete puzzles for all sizes tested. It works by placing queens on the board one at a time and backtracking when an invalid placement is encountered, trying a new path.

- **Space Complexity**: Backtracking is highly memory-efficient, keeping its space usage around 100 MB for all puzzle sizes. It is the most space-efficient algorithm tested, as it stores only the current state of the solution.
- **Time Complexity**: While the time complexity increases as the puzzle grows, backtracking handles smaller puzzles quickly, and the time required for larger puzzles grows at a reasonable rate compared to Breadth-First Search and Depth-First Search. Despite the increase in time complexity, Backtracking remains the most dependable method for solving the puzzle, particularly for larger sizes.
- **Completeness**: Backtracking guarantees a solution if one exists. It is the most reliable algorithm tested, successfully completing puzzles of all sizes with consistent performance. However, it can be slow for larger puzzle sizes due to its exhaustive search strategy.

6.4 Genetic Algorithm Search

The Genetic Algorithm Search is a more advanced approach that creates a group of potential solutions and improves them over time by selecting the best ones, combining them, and introducing random changes.

- **Space Complexity**: The Genetic Algorithm Search maintains low space usage, similar to Local Search, staying around 100 MB in the plot. It operates on a fixed population of solutions rather than exploring the entire search tree, which helps keep its memory usage low.
- **Time Complexity**: The Genetic Algorithm Search is not shown completing puzzles in the time complexity plot, likely due to insufficient tuning or the stochastic nature of the method, which causes variability in its results.
- **Completeness**: The Genetic Algorithm Search is not shown completing puzzles. It is also not guaranteed to find a solution at all, like Local Search, it can get stuck in local optima or fail to converge to an optimal solution.
- **Partial Solutions**: While Local Search does not always find a complete solution, it often returns a partial solution. For example, it may place a few queens correctly, but not all of them, as it gets stuck in a local optimum where a complete arrangement is unreachable without further exploration.

Among all the algorithms tested, **Backtracking** emerged as the most successful method for solving the LinkedIn Queens puzzle. It demonstrated consistent reliability across all tested puzzle sizes, guaranteeing a solution when one exists.

7 Conclusion

In this project, several algorithms were tested to solve the LinkedIn Queens puzzle, where the challenge is to place queens on a board while following certain rules. The algorithms used include Breadth-First Search, Depth-First Search, Local Search, Backtracking, and Genetic Algorithm Search. Each of these algorithms was evaluated based on three key metrics: time to solve the puzzle, memory usage, and the success rate in finding a valid solution.

This experiment is important because it helps understand how different algorithms perform when solving complex problems. The LinkedIn Queens puzzle is harder than the classic N-Queens problem because it includes extra constraints, like the need to place one queen in each colored region. This makes the puzzle more difficult, so it is a good test for how well these algorithms can handle complicated problems. Understanding how different algorithms work is useful, as many real-world problems are similar to this in terms of complexity.

The key takeaway from this experiment is that no single algorithm is optimal for all scenarios. Each approach has its strengths and weaknesses, and their effectiveness can vary greatly depending on the size and complexity of the problem at hand. This study highlights the importance of choosing the right algorithm to solve complex problems. It shows how understanding the strengths and weaknesses of each algorithm can help in making better decisions when solving challenges that involve optimization and constraints.

Our hypothesis was that Backtracking would perform similarly well in solving the LinkedIn Queens puzzle as it did in the traditional N-Queens problem. This was confirmed, as Backtracking proved to be reliable and efficient in solving the puzzle.

8 Future Work

In the future, there are several ways to improve the algorithms for solving the LinkedIn Queens puzzle in the future. One of the main focuses will be making the current algorithms more efficient, especially for larger puzzles. For example, exploring advanced heuristics and optimization techniques can help speed up Breadth-First Search, Depth-First Search, and Local Search by guiding them toward promising solutions while avoiding unnecessary calculations.

Another way to improve is to look into hybrid algorithms that combine the strengths of different approaches could lead to better results. For example, combining Local Search with Backtracking could help avoid getting stuck in local optima while still ensuring the solution is correct.

Parallel computing could also help speed up the algorithms. By dividing the puzzle into smaller parts and solving them at the same time, the time it takes to find a solution could be reduced, especially for larger puzzles.

Lastly, machine-learning techniques could be explored to improve the puzzle-solving process. By training models on previous puzzle solutions, machine learning could help predict good queen placements and guide the search more efficiently. Although this is a long-term goal, it could open up new ways for solving optimization problems in general.

9 Acknowledgments

This report was created by Caleb Petterson and Shivani Sairam. The workload was distributed evenly between them, including the conducting of experiments and coding components.

9.1 Report

Both Caleb and Shivani contributed equally to the Introduction, Related Work, Methodology, Future Works, and Conclusion sections.

Caleb individually contributed the Problem Statement and Results sections. Shivani individually contributed the Abstract and Analysis. Both parties thoroughly reviewed and provided edits for all sections of the report.

9.2 Code

Both Caleb and Shivani contributed equally to the code repository on GitHub.

Caleb individually contributed the screen_capture.py and solver_experiment.py files. Shivani individually contributed the plotting.py file. Both parties contributed equally to the sourcing of data from images of LinkedIn Queens puzzles with varying grid sizes. Additionally, both contributed equally to the search_algorithms.py file.

References

[1] Bruce Abramson and Moti Yung. Divide and conquer under global constraints: A solution to the n-queens problem. *Journal of Parallel and Distributed Computing*, 6(3):649–662, 1989. Received 10 February 1987, Available online 19 February 2004.

[2] Ricky Fernando. Backtracking algorithm for solving star battle puzzle, 2019/2020.

[3] Vishal Khanna, Abhishek Bhardwaj, and Sarvesh Chopra. Optimization of n-queens problem using hybrid memetic algorithms. *International Journal of Advanced Research and Innovative Ideas in Education (IJARIIE)*, 3(3):840, 2017.

[4] Marian Kovac. N-queens problem by efficient non-backtracking algorithm using local search, heuristics and tabu search elements. In *The 10th International Conference on Digital Technologies 2014*, pages 159–163, 2014.

[5] J. Mańdziuk. Neural networks for the n-queens problem: a review. *Control and Cybernetics*, 31(2):217–248, 2002.

[6] Omid Moghimi and Amin Amini. A novel approach for solving the n-queen problem using a non-sequential conflict resolution algorithm. *Electronics*, 13(20):4065, 2024.

[7] NMOF. Local search in n-queens problems, 2016.

[8] Igor Rivin, Ilan Vardi, and Paul Zimmerman. The n-queens problem. *The American Mathematical Monthly*, 101(7):629–639, 1994.

[9] Santhosh G S, Piyush Joshi, Ayan Barui, and Prasanta K. Panigrahi. A quantum approach to solve n-queens problem, 2023.

[10] Uddalok Sarkar and Sayan Nag. An adaptive genetic algorithm for solving n-queens problem, 2017.

[11] Farhad SoleimanianGharehchopogh, Bahareh Seyyedi, and Golriz Feyzipour. A new solution for n-queens problem using blind approaches: Dfs and bfs algorithms. *International Journal of Computer Applications*, 53:45–48, 09 2012.

[12] R. Sosic and J. Gu. Fast search algorithms for the n-queens problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(6):1572–1576, 1991.

[13] Cheng Zhang and Jianpeng Ma. Counting solutions for the n-queens and latin-square problems by monte carlo simulations. *Phys. Rev. E Stat. Nonlin. Soft Matter Phys.*, 79(1 Pt 2):016703, 2009.