LinkedIn Queens Game: Algorithmic Solvers and Puzzle Generation

Introduction

The LinkedIn Queens game represents a variant of the classic n-queens problem, incorporating colored regions as additional constraints. This project explores multiple algorithmic approaches to solving and generating LinkedIn Queens puzzles, focusing on comparing Boolean Satisfiability (SAT) and Constraint Programming (CP) techniques. Inspired by our experience applying SAT solvers to solving and generating Sudoku puzzles, we wanted to extend similar methodologies to this game while doing a comparative analysis of solver performance across different puzzle complexities.

Game Description and Constraints

The LinkedIn Queens game is an n x n grid with n distinctly colored regions. The objective is to place exactly n queens on the board to satisfy each constraint. First, exactly one queen must exist in each row. Second, exactly one queen must exist in each column. Third, exactly one queen must appear in each colored region. Fourth, no queen can be diagonally adjacent to another queen. These constraints create a complex puzzle where algorithmic solvers and heuristics can be useful.

Implementation Approaches

Backtracking Solver

As a baseline for performance comparison, we implemented a naive backtracking solver. This approach systematically places queens on the board while ensuring all constraints remain satisfied, backtracking when violations occur. The algorithm begins with an empty board and attempts to place queens individually, starting from the first row. For each cell in the current row, it checks whether placing a queen would violate any constraints. If a placement is valid, it recursively attempts to place queens in subsequent rows. If no valid placement exists for the current row, it backtracks to the previous row and tries a different configuration.

While conceptually straightforward, this method proved computationally expensive for larger board sizes due to its exhaustive search through the solution space without sophisticated pruning techniques. This solver becomes infeasible very quickly as the puzzle size increases. However, it provides a valuable reference point for validating the correctness of our more sophisticated solvers.

PycoSAT Solver

Our primary implementation encoded the LinkedIn Queens game as a Boolean satisfiability problem using PycoSAT. For an n x n board, we defined n^2 boolean variables, each representing whether a queen is placed at a specific cell. True values indicate queen placement; false values indicate empty cells.

To encode the basic constraints into CNF clauses, we incorporated the "at most one" and "at least one" ideas in our Sudoku solving homework. For example, to encode that there is precisely one queen in a row, we created one clause for "at least one" queen where every positive variable for every cell in a row was in the same clause (meaning at least one must be true). Then we created pairs of clauses for every cell in the row, which had the negation of the corresponding variables (meaning we cannot have two queens in the same row). Similar logic allowed us to encode the rest of our constraints.

These constraints were translated into CNF clauses suitable for the PycoSAT solver. After establishing these constraints, we validated our solver's accuracy against known puzzle solutions, confirming the correctness of our constraint encoding.

CP-SAT Solver

We implemented a Constraint Programming approach using CP-SAT, applying inequality and equality constraints to regulate queen placement. The model utilized a binary grid representation (1 for queen presence, 0 for absence) with AddExactlyOne constraints ensuring proper queen distribution. The invariants that are encoded within CP-SAT exactly follow the constraints we encoded for PycoSAT. Namely, we ensured that exactly one queen was placed in each row, column, and region, but we also ensured that at most one queen was located within each 3x3 region.

The CP-SAT model directly translated these game rules into a cohesive constraint model, leveraging the solver's built-in constraint propagation mechanisms (AddExactlyOne) to explore the solution space efficiently.

Heuristics and Optimizations

Both solvers were enhanced with heuristics created from human solving techniques. These heuristics reflect the logical deductions experienced puzzle solvers employ to narrow down potential queen placements without resorting to trial and error.

Immediate Placement Heuristic

A queen must be placed there when a region contains only one cell (a singleton).

Region Exclusivity Heuristic

If a region's cells all occupy a single row or column, other regions' cells in that row/column cannot contain queens.

Paired Regions Heuristic

This heuristic is an extension of the previous heuristic. If two regions exclusively occupy the same two rows/columns, other regions' cells in those rows/columns cannot contain queens.

Extended Region Patterns

For specifically PycoSAT, we also tried to further generalize the concept of paired regions to handle cases where three regions occur in the same three rows/columns or more, but we found a marginal to no increase in success, leading us to not implement for CP-SAT and remove from our final model testing.

Cell Elimination Based on Region Constraints

We implemented a constraint that eliminates cells whose occupation would leave a region with no valid queen placements. This involves analyzing each cell and simulating its occupation to determine if any region would subsequently have no valid queen placement possibilities due to row, column, or diagonal constraints.

PycoSAT Heuristic Implementation

Whenever a cell is determined to not be able to hold a queen from the heuristics, we simply add a singleton clause with the negation of that cell's variable. On the other hand, if a cell must have a queen, we add a singleton clause of that cell's variable.

CP-SAT Hinting System

The CP-SAT implementation additionally employed a sophisticated hinting system that assigned placement likelihood scores to guide the solver's search process. This system calculated scores based on multiple factors:

1. Region density: Cells in regions with fewer total cells received higher scores, reflecting the increased probability that a queen must be placed in one of these limited options. The score component was proportional to the inverse of the region size.
2. Neighbor diversity: Cells with more diverse neighboring cells (belonging to different regions) received higher scores. This metric captured the intuition that queens are more likely to be placed at region boundaries where they can satisfy multiple constraints simultaneously. The diversity was calculated by counting distinct regions in the cell's vicinity.
3. Distance from center: Edge and corner positions received lower scores than central positions, acknowledging the geometric property that central placements typically offer more flexibility in satisfying diagonal attack constraints. The penalty was proportional to the Manhattan distance from the board's center.

These scores were normalized and combined into a composite hinting value that guided the CP-SAT solver's variable selection during search, effectively incorporating domain-specific knowledge into the general-purpose constraint solver.

Puzzle Generation

Unlike Sudoku, LinkedIn Queens lacks a deterministic puzzle generation approach. We developed a stochastic generation method as follows:

The generation process begins by assigning "starter" cells to each of the n regions, ensuring an initial diverse distribution across the board. From these seed points, we employ a random flood-fill algorithm to expand each region until the entire board is covered. This process assigns each cell to a region by randomly selecting adjacent unassigned cells and extending the region boundaries.

The resulting board configuration, however, might not be a puzzle with a unique solution. To move the board toward a uniquely solvable puzzle, we identify "border" cells—those adjacent to cells of different regions—and randomly modify their region assignments. After each modification, we test whether the board has exactly one solution using our PycoSAT solver's itersolve functionality.

We ran into one issue during this process: naively changing border cells could fragment regions into disconnected components, creating invalid puzzles where a region exists as multiple isolated patches. Consider a region with five cells arranged vertically. Changing the middle cell's region would split the original region into two disconnected components. This fragmentation violates the implicit connectivity constraint of LinkedIn Queens puzzles.

We resolved this issue by implementing a "number of islands" algorithm based on BFS. Before accepting any border cell modification, we verify that the total number of connected components across all regions remains equal to n. If at any point there are more than n islands, we revert whatever change we previously made.

To prevent algorithm stagnation in cases where a valid solution is difficult to find, we incorporated a max_changes parameter. If the algorithm exceeds this threshold of attempted modifications without finding a uniquely solvable puzzle, it restarts with a fresh board configuration. This approach significantly improved the efficiency of puzzle generation, particularly for larger board sizes. One reason for this is that the generation process may have repeatedly tried to switch the same border cell back and forth.

Testing

When we first created the Backtrack Solver, we tested the solution on hard-coded puzzle boards and checked the results by hand to ensure that our logic was correct. We did the same process for testing the initial solution of the PycoSAT solver without heuristics, as well as comparing the results to our verified Backtrack Solver. After we generated puzzles with PycoSAT, we then simply tested every iteration afterwards (PycoSAT with heuristics, CP-SAT, and then CP-SAT with heuristics) by comparing the solvers' solutions on all of the generated puzzles with our verified Backtrack Solver and verified PycoSAT solver (without heuristics). This allowed us to create robust, verified solvers and test that our heuristics were correct.

Results

We conducted extensive performance evaluations using thousands of generated puzzles ranging from 5×5 to 16×16 dimensions.

Our first observation was the huge performance advantage of PycoSAT over CP-SAT across all puzzle dimensions.

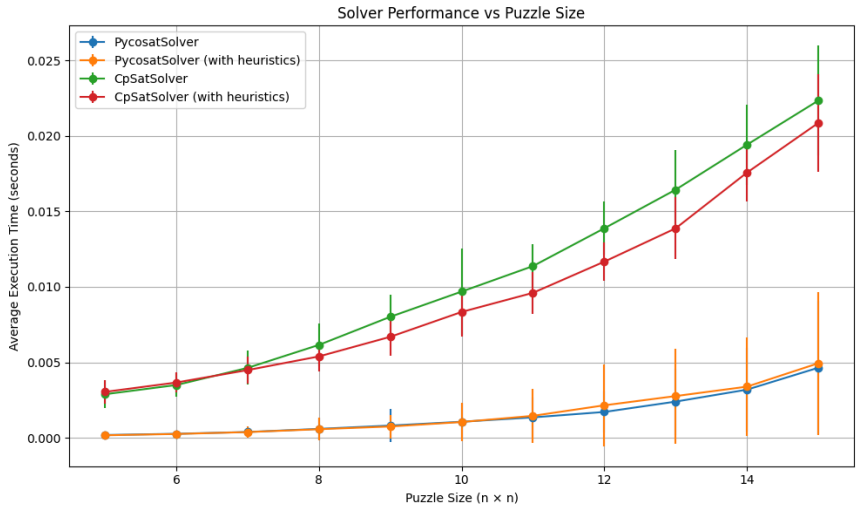

Figure plotting the different runtimes of solvers across different puzzle sizes

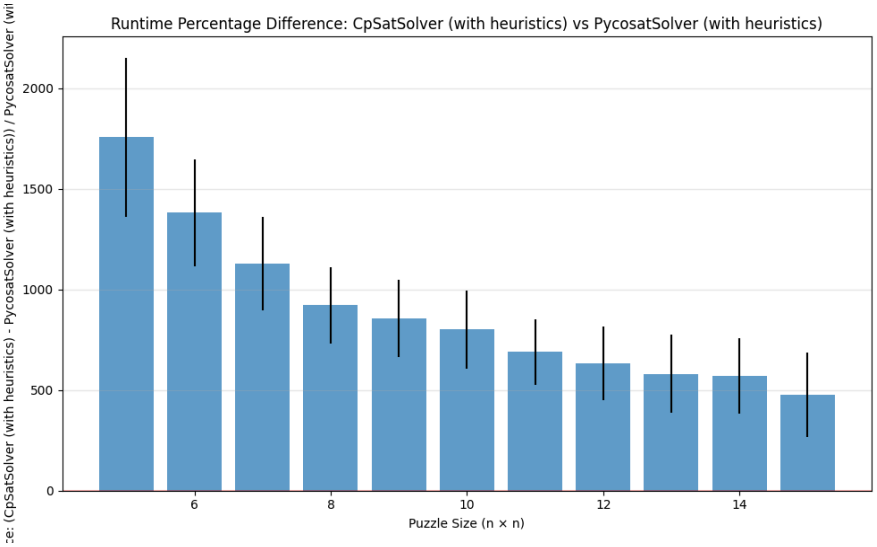

Figure plotting the percentage difference between the CP-SAT Solver and the PycoSAT Solver (positive means the CP-SAT Solver is slower).

As illustrated in the first figure, PycoSAT consistently executed much faster than CP-SAT, with the performance difference widening as puzzle dimensions increased. The second figure provides a clearer visualization of this performance gap, showing that CP-SAT with heuristics required between 500% and 1700% more execution time than PycoSAT with heuristics, depending on puzzle size. This substantial differential decreased as puzzle size increased, suggesting that the relative inefficiency of CP-SAT becomes less pronounced for more complex problems, although still very large.

What was more interesting was the different impact of heuristics on solver performance.

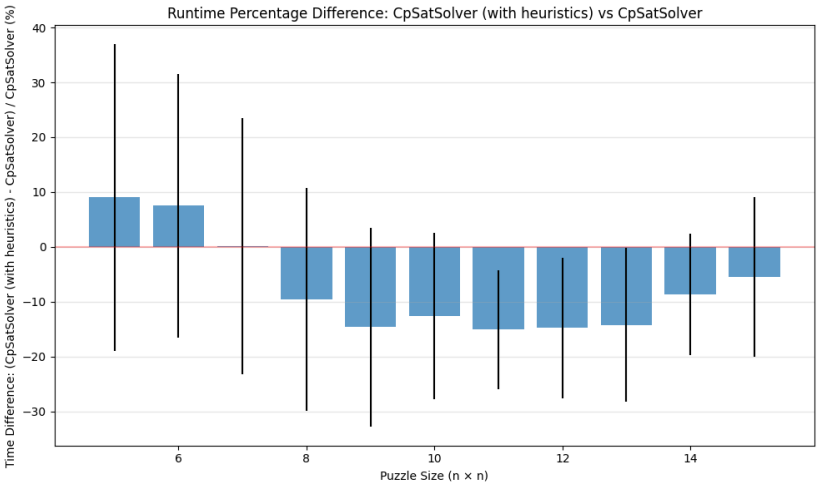

Figure plotting the percentage difference between the CP-SAT Solver with heuristics and without (positive means that with heuristics is slower).

As shown in this figure, CP-SAT showed improvement with heuristics for puzzles larger than 7×7, with runtime reductions averaging 10-15%. This benefit increased with puzzle complexity, confirming our hypothesis that heuristic preprocessing becomes more valuable as the problem space expands. Conversely, PycoSAT demonstrated a performance decrease when enhanced with the same heuristics.

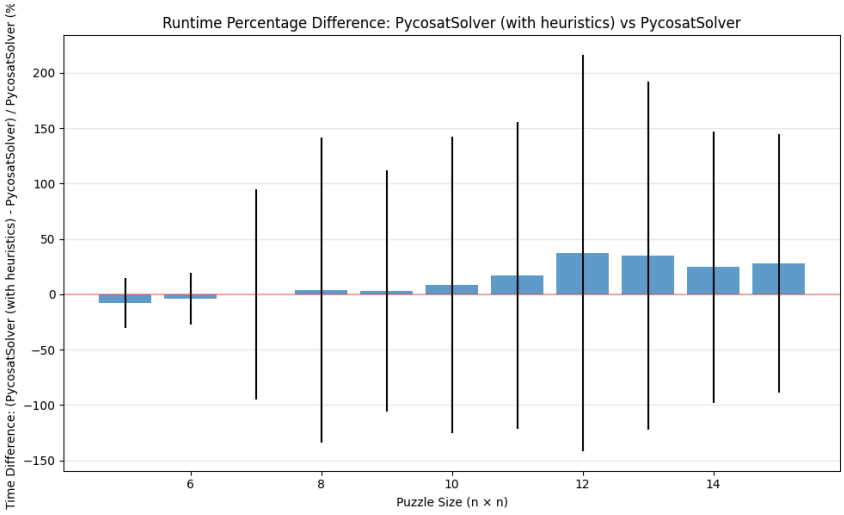

Figure plotting the percentage difference between the PycoSAT Solver with heuristics and without (positive means that with heuristics is slower).

This penalty increased with puzzle size, with heuristic-enabled PycoSAT executing up to around 40% slower than the non-heuristic version for larger puzzles. This suggests that for the PycoSAT solver, the computational overhead of calculating and applying domain-specific heuristics may exceed the benefits of search space reduction. We also experimented with adding even more heuristics, but the difference in solve time between the heuristic and non-heuristic versions only further increased.

We chose not to include the backtracking solver in the benchmarking graphs since it took far too long to execute on puzzles compared to the other solvers.

Results Analysis

We decided to form hypotheses on why the impact of heuristics differed across our solvers.

For CP-SAT, heuristics proved beneficial because they significantly reduced the search space, particularly for complex puzzles where the cost of computing heuristics was amortized by the substantial pruning benefits. The CP framework, built to handle a wide variety of constraint types, benefits considerably from domain-specific knowledge that can guide its search process. The hinting system further enhanced performance by influencing variable selection during solution construction, effectively prioritizing the most promising paths in the search tree.

The observed performance improvement for CP-SAT with heuristics aligns with theoretical expectations. Constraint Programming excels when specialized knowledge can be incorporated to prune invalid solution paths early. The computational cost of preprocessing the board to apply heuristics is offset by eliminating large portions of the search space that would otherwise be explored fruitlessly. As puzzle complexity increases, this tradeoff becomes increasingly favorable, explaining the growing advantage of heuristic application for larger board sizes.

Conversely, PycoSAT's performance degraded with heuristics. We hypothesize that the PycoSAT solver is already highly efficient at solving problems like these, to the point that the actual cost of computing the heuristics is comparable to the cost of just solving the problem. Computing heuristics is not necessarily "cheap", and can take up to around $O(n^2)$ or $O(n^3)$ time to compute, depending on what heuristics we choose. Additionally, one flaw of our heuristic usage in the PycoSAT solver is that there is no notion of "hinting" in PycoSAT like there is in CP-SAT. We can only apply to heuristics at the very beginning of solving the puzzle, not in the middle of the process of finding a solution, which is how humans usually play the puzzle (they don't guess random spots, they apply heuristics over and over again to solve the Queens game). Essentially, the cost of computing the heuristics is rather large compared to the overhead of just solving the puzzle directly with the constraints of PycoSAT, leading to the differences that we see in the graph.

Our main hypothesis behind why CP-SAT was overall slower is that the CP-SAT library is much more versatile and generalizable than PycoSAT. CP-SAT is designed to solve a much wider range of problems, since it takes constraint programming problems and then converts them into SAT problems to then solve. It supports things like booleans, integers, etc. and more complex constraints that PycoSAT does not. This results in extra overhead to solve relatively simple problems like the Queens game, whereas PycoSAT does

not need to. We also do not fully take advantage of some of the generalizability that CP-SAT offers over PycoSAT. The Queens game is very naturally programmable into a SAT problem instance to be solved by PycoSAT, unlike some other constraint programming problems that we have solved throughout the semester. Thus, PycoSAT is almost perfect for solving this problem, whereas CP-SAT takes in a relatively more complex constraint programming encoding of the problem that it then has to convert into a SAT problem and solve, resulting in more runtime.

Another key insight involves the timing of heuristic application. In human solving, heuristics are applied iteratively throughout the solution process, with each deduction potentially enabling further deductions in a cascading manner. Our PycoSAT implementation could only apply heuristics as preprocessing steps before invoking the solver, limiting their impact to initial problem simplification. In contrast, CP-SAT's hinting mechanism provided guidance during the solution process itself, better mirroring human solving approaches and allowing the heuristics to influence decision-making throughout the search.

Conclusion

This project demonstrates possible approaches to solving the very popular Queens game on LinkedIn. Our findings reveal that while PycoSAT delivers superior raw performance for the puzzle, the architectural differences between SAT and CP solvers highlight important tradeoffs in algorithm selection. This problem is naturally optimized for a PycoSAT approach due to its intuitive encoding and easy constraints translation, with heuristics doing much more for CP solvers than pure SAT. This insight suggests that optimal algorithm selection depends not only on performance metrics but also on considerations such as problem variability, adaptability requirements, and the value of incorporating human-like solving approaches.

Our work provides practical implementations for LinkedIn Queens puzzle solving and generation, as well as theoretical insights into constraint satisfaction algorithm behavior. With its region connectivity preservation approach, the stochastic puzzle generation methodology successfully produces uniquely solvable puzzles while maintaining the original invariants and rules behind the original game. Future research directions could explore hybrid approaches that combine the raw efficiency of specialized SAT solvers with the hinting capabilities of constraint programming frameworks, potentially achieving performance improvements across diverse problem instances while maintaining the flexibility to incorporate domain-specific knowledge. This hybrid could offer the most optimal approach, though PycoSAT is already a very good tool for this problem.