TECHNISCHE UNIVERSITÄT DRESDEN

SUBMITTED BY:
URMEE PAL
MATRICULATION NR.: 4865875
MODULE: MCL- MASTER'S THESIS
DEGREE PROGRAM: INTERNATIONAL MSC.
IN COMPUTATIONAL LOGIC
FACULTY OF COMPUTER SCIENCE

SUPERVISED BY:
PROF. DR. RER. POL. MARKUS KRÖTZSCH
DR. HABIL. HANNES STRASS
FACULTY OF COMPUTER SCIENCE
TUTORED BY:
DIPL.-MATH. MAXIMILIAN MARX

Master's Thesis
Construction of Human-Solvable Akari Puzzles

Urmee Pal

Dresden, July 14, 2025

Abstract

This master's thesis explores the construction and human-solvability of unique Akari puzzles using Answer Set Programming (ASP) with Python integration. It presents an efficient puzzle constructor capable of generating unique puzzles across various grid sizes (6×6 to 14×14), with 12×12 identified as the largest feasible size. The research also introduces a human strategy-based solver that implements six Akari-solving strategies used by human solvers. The puzzle constructor, which ensured the uniqueness of solutions, also produces puzzles that are unsolvable by the human strategy-based solver. To address this, I redesign the constructor to embed human solvability constraints directly into the generation process, ensuring that only puzzles uniquely solvable using the six strategies are accepted. Performance analysis demonstrated a significant trade-off: the redesigned constructor guarantees human accessibility but provokes a steep increase in construction time, especially for larger grids. The largest feasible grid size of the redesigned constructor is identified as 10×10. The study demonstrates ASP's effectiveness in handling complex puzzle generation and solving, providing a foundation for further research in automated puzzle systems and constraint satisfaction problems. A case study of a 10×10 puzzle solution illustrates the solver's deduction process, validating the implemented strategies. A comparative analysis with an established C++ implementation is also conducted in this thesis to contextualize the strengths and limitations of the developed ASP-based Akari puzzle construction and solving framework.

Contents

List of Figures

Listings

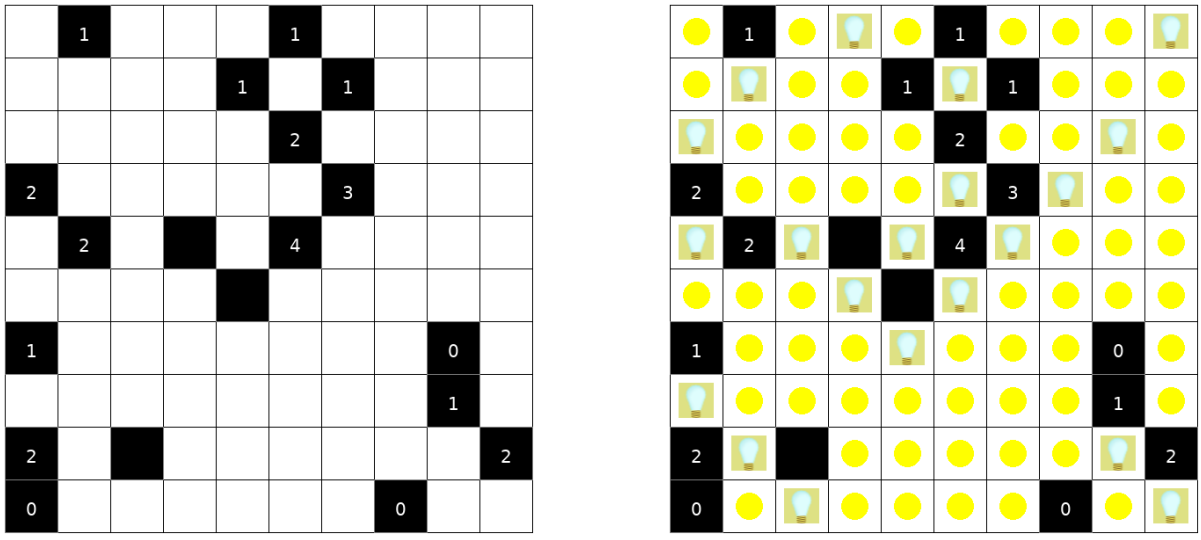
List of Tables

1 Introduction

Puzzles that need deductive/logical reasoning to solve are known as logic puzzles. As logic puzzles gain popularity, scientific studies on their complexity are also being conducted. Despite being challenging to solve, the puzzles are intriguing to study since a limited number of straightforward rules frequently explain them.

Akari is one such popular paper-and-pencil puzzle that originated in Japan. Akari means 'Light' in Japanese. It is also commonly referred to as the "Light Up" puzzle. It was originally named Bijutsukan, which means "museum" or "Art Gallery" in Japanese. Complete lighting is essential in museums for properly presenting the Art. Keeping this concept as motivation, the Akari puzzle is formed. Akari puzzles were published by Nikoli, a Japanese publishing company, in 2001. It was focused on publishing logic puzzles and games that are culturally independent. Akari has gained popularity for its unique blend of logical deduction and spatial reasoning challenges.

As most other paper-and-pencil puzzles, an Akari puzzle consists of a grid of cells with some clues given in advance and a list of rules. The initial grid consists of white and black cells. The black or shaded cells are represented as obstructions in the way of light. The goal is to put light bulbs in the white cells of the grid. The light bulbs are to be placed to illuminate all white cells using a limited number of light sources. If a black cell has a number on it, it is from 0 to 4. The number indicates the number of bulbs that have to be placed adjacent to its four sides. Bulbs on the diagonal sides of a numbered cell do not count. A black cell without a number can have any positive number of light bulbs or no light bulbs around it. Figure 1 illustrates an example of a 10×10 Akari puzzle grid in both unsolved and solved states.



(a) Unsolved Akari Grid

(b) Solved Akari Grid

Figure 1: A 10×10 Akari puzzle grid in Unsolved and Solved State.

In this thesis, I studied the complex world of Akari puzzles. The goal of this thesis is to design an algorithm that generates Akari puzzles with "nice" solutions using the ASP language. Here, "nice" solutions refer to puzzles that have a unique solution that can be found by human solvers applying a sequence of logical deductions without having to resort to guessing whether a cell contains a light. The main focus of this thesis is the use of Answer Set Programming (ASP). It is an efficient declarative programming approach ideal for modeling and resolving combinatorial problems and also building human-like

problem-solving strategies. The Akari solution techniques, puzzle generation, and the formalization and resolution of these puzzles are done using ASP. This study attempts to close the gap between computer and human Akari-solving strategies. It provides insights into the characteristics of a puzzle's difficulty and the potential for automatically generated and solved puzzles. In addition to developing a greater comprehension of the logic of the puzzle, formalizing the puzzle rules establishes the groundwork for ensuring human solvability of the puzzles.

The work begins with a detailed literature review focusing on previously published Akari solution strategies and difficulty assessments. Next, I focus on one of the major goals of this thesis, the construction of an algorithm to generate Akari puzzles with unique solutions. This is a crucial component of well-designed logic puzzles. After that, six strategies that the human players use for solving the Akari puzzle were identified and discussed. These strategies are then formalized in ASP, providing a logical framework to simulate human strategies. The unique solutions of the constructed puzzles are then analyzed to be reached through the application of the discussed formalized strategies using an ASP human strategy-based solver. Upon analysis, the constructor has been redesigned to ensure both uniqueness and human solvability.

The intricate mechanics of captivating Akari logic puzzles are explored, providing insights into their structure, solution strategies, and the algorithm behind them. All of these are made possible through the understanding and application of ASP.

2 Related Work

Shi-Jim Yen and Cheng-Wei Chou have published a paper [1] that presents a novel method for solving Akari. The authors presented a solver that integrated pattern matching, followed by techniques from local search to improve solution efficiency. First, the solver used a pattern-matching strategy to quickly solve parts of the puzzle that can be solved with basic heuristics. In this part, the solver identified a configuration to determine the possible bulb placements from the number clues given in the black cells. They categorized the grid into distinct regions based on the configuration of the black and white cells. The categorization involved identifying specific patterns that can be quickly solved using predefined Akari rules. They utilized four types of pattern matching to address these configurations. The first one was identifying black cells that can be completely resolved based on their surrounding white cells. The algorithm checked if the number of adjacent white cells (potential bulb positions) matched exactly the requirement specified by the clue in the black cell. If it did, the algorithm placed bulbs in those positions. The second pattern examined specific configurations where certain placements of bulbs were mandated by the constraints of adjacent black cells. For instance, if only a limited number of adjacent white cells were not illuminated after placing the required number of bulbs according to the clue in a black cell, those cells must be filled with bulbs to satisfy the requirement. The third pattern involved marking certain positions where bulbs cannot be placed based on existing placements or constraints from neighboring black cells. If placing a bulb in a specific white cell would exceed the required number of bulbs set by an adjacent black cell, that position was marked as unavailable for bulb placement. The last pattern analyzed how existing bulbs influence other areas of the grid, enabling it to deduce additional placements or exclusions indirectly. This pattern helped in resolving areas that were not directly influenced by immediate constraints but were affected by more complex interactions within the grid. This pattern handled straightforward cases rapidly. This reduced the complexity significantly for the local search techniques used later. After this, any remaining unsolved areas were addressed using a local search strategy. This involved treating each unmarked white cell as a small subproblem where they tested potential bulb placements iteratively and refined their approach based on feedback from previous placements. For example, let an Akari puzzle grid have six unmarked white cells after applying the integrated pattern matching. Taking each of these as a small subproblem, it will check two possibilities (placing a bulb or not) for each unmarked white cell. Thus, with two on each step in the local search process, it will evaluate a total of $6 \times 2 = 12$ configurations, making six decisions. On each step, it will choose one configuration according to feedback from previous steps. After applying local search to all unresolved cells, a final verification step checks if all placements comply with the puzzle's rules. If any inconsistencies were found, adjustments were made based on earlier deductions. One of the advantages of using local search, as noted by the authors, was that it did not require exhaustive searching through all possible configurations ($2^6 = 64$ configurations for the example above). Instead, it focused on making educated guesses and testing them, which streamlined the problem-solving process. This hybrid approach allowed for rapid completion of Akari puzzles, as evidenced by their experiments, where they solved 100 25×25 grid puzzles collected from the internet with an average time of 0.1377 seconds per puzzle. Among those, 30.93% of each puzzle was solvable by pattern matching alone. This took a mean of 0.00015 seconds per region [1].

Salcedo-Sanz et al. proposed a novel two-step evolutionary algorithm (EA) to solve Akari, aka the Light-up puzzle [2]. This aligns with other works emphasizing hybrid strategies for complex problem-solving [1]. Salcedo-Sanz et al.'s approach begins with a preprocessing phase. This phase was designed to simplify the puzzle and reduce the search space before applying the main EA. The authors employed some rules from Akari to identify and place bulbs in certain positions that are guaranteed to be correct in the final solution. These rules were based on the constraints of the puzzle and the properties of the black cells. For instance, if a black cell with a number had exactly that many empty cells around it, bulbs were automatically placed in those empty cells. Similarly, if a black cell with zero had no empty cells around it, all adjacent cells were marked as illuminated. The preprocessing also identified cells that cannot contain bulbs due to the puzzle's constraints. By applying these rules iteratively, the algorithm

significantly reduced the number of cells that need to be considered in the main optimization process, thereby improving the efficiency and effectiveness of the overall solution method. The authors then employ a binary encoding scheme to represent the puzzle state. The encoding process transformed each potential solution to the Light-up puzzle into a binary string. Each bit in this string corresponds to a cell in the puzzle grid, with 1 representing a possible bulb placement and 0 representing an empty cell. Black cells were not included in the encoding, as they cannot contain bulbs. This binary solution string allowed the algorithm to efficiently manipulate and evaluate different puzzle configurations. This binary solution string allows the algorithm to efficiently manipulate and evaluate different puzzle configurations. Then, the two-step (EA) works with a population of these binary solution strings. The first step of the algorithm focused on the global structure of the puzzle. It used an EA to determine the optimal placement of bulbs in the puzzle grid. It employed genetic operators such as crossover and mutation to explore different bulb arrangements across the entire puzzle board. The second step, which is nested within the first, is another EA that operates on a more local level. This inner algorithm is activated whenever the outer algorithm generates a new potential solution. Its purpose is to fine-tune the bulb placements by considering the specific constraints and illumination rules of Akari. This inner algorithm works on smaller sections of the puzzle, optimizing the bulb positions to satisfy local constraints and maximize illumination. This step allows for a more efficient exploration of the solution space. This two-step approach enabled the algorithm to balance between global exploration and local optimization. It potentially led to better solutions than a single-level evolutionary approach. Their method effectively reduced the search space, enhancing computational efficiency and accuracy. This study demonstrates the power of EA for tackling constraint satisfaction problems, providing a robust framework applicable to other similar puzzles or optimization challenges [2].

Igor Rosberg, Elizabeth Goldbarg, and Marco Goldbarg present a two-phase algorithm for solving the Light-Up puzzle [3]. Rosberg et al. also did pre-processing in the first phase, similar to Salcedo-Sanz et al. [2]. Then, they introduced the Ant Colony Optimization (ACO) algorithm in the second phase. In the preprocessing phase, they applied logical rules to determine white cells that are forbidden from receiving light bulbs and those that must receive them. This process reduced the search space for the subsequent ACO algorithm. The preprocessing algorithm consists of six steps that are applied iteratively until no more changes occur. These steps included assigning value 0 to white cells adjacent to black cells with entry 0, placing light bulbs in white cells that must receive them due to constraints, and setting values based on the satisfaction of numbered black cell constraints. The ACO phase then worked on this simplified grid. It used artificial ants to construct solutions based on heuristic information, pheromone trails, and probabilistic decision-making. Each white cell in the grid was assigned a heuristic value and a pheromone level. The heuristic value remained constant throughout the algorithm's execution. It was calculated as the sum of the integers in adjacent numbered black cells plus one. This indicated the cell's preference for receiving a light bulb. Pheromone levels were initially set to 1 for each white cell. These were dynamically updated during the algorithm's execution. The ACO algorithm operates in iterations. It involved multiple artificial ants constructing solutions independently in each iteration. To build a solution, an ant considers each row group of white cells, deciding whether to place a light bulb in the group based on a probability formula. If a light bulb is to be placed, the specific cell within the group is chosen based on both pheromone and heuristic information. Both factors were balanced using a probability calculation. The ants also maintain a memory to track cells that become prohibited from receiving light bulbs during solution construction. At the end of each iteration, pheromone levels were updated. First, pheromone evaporates from all cells according to a specified rate. Then, the ants that constructed the best solution(s) in that iteration deposit new pheromones on the cells where they placed light bulbs. The amount of pheromone deposited is inversely proportional to the solution's cost, encouraging future ants to favor components of good solutions. The number of white cells that are not illuminated and the number of constraints that are not satisfied concerning the numbered black cells are the two components that are added together to determine the quality of a solution. According to this criterion, the quality of the solution increases with a smaller sum. For example, a solution cost would be 24 if it contained 19

non-illuminated white cells and five faults with the numbered black cells. The ant colony optimization algorithm's pheromone updating procedure depends extensively on the solution cost. The amount of pheromone deposited by ants that found the best solution(s) in an iteration is calculated as:

$$\Delta\tau_{ij}^{k} = \frac{Q}{E}$$

Here $\Delta\tau_{ij}^{k}$ is the amount of pheromone deposited by the k-th ant in position (i, j), E is the value (cost) of the optimal solution, and Q is a constant. Because of this inverse relationship, more pheromone is deposited in lower-cost (better) solutions, directing future ants toward solution components that show promise. The algorithm continues for a maximum number of iterations until a perfect solution is found with zero unilluminated cells and satisfied constraints. The ACO approach allows for a balance between the exploration of the solution space and the exploitation of good solution components. The algorithm was tested on 32 Light Up instances ranging from 7x7 to 40x30 grids. Without preprocessing, the algorithm successfully solved all instances up to 14x14 with an average convergence rate of 93% for the 20 smaller instances. The authors note that this performance is promising compared to previous works. The addition of the preprocessing phase improved the algorithm's performance. According to the result analysis, this preprocessing phase reduced the search space significantly. The reductions ranged from 0% to 100% depending on the puzzle difficulty. For some easy puzzles, preprocessing alone was sufficient to solve them completely. The preprocessing phase took less than 0.05 seconds for all tested instances [3].

Bram Pulles gives a thorough and formal analysis of the popular logic puzzle Akari in his bachelor's thesis "Analysis of Akari" [4]. This work goes deep into the theoretical and practical implications of the computational complexity of Akari. The puzzle complexity and algorithm design are studied here in a comprehensive approach to both theoretical analysis (complexity proofs) and practical implementation (solving algorithms and puzzle generation). The core of Pulles' research is six carefully selected variants of Akari. These variants were selected to examine how slight variations in the rules of the game can significantly alter its computational complexity. This method not only produces a thorough comprehension of Akari itself. It also provides valuable insights into the nature of computational complexity in logic puzzles. One of the major results of this work is the proof that three of the variants of Akari are NP-complete. Pulles proves this via reduction from the well-known NP-complete problem Circuit-SAT. Through a process of iteration, Pulles creates complex gadgets that correctly translate the Boolean logic of Circuit-SAT to the spatial and illumination constraints of Akari. This is nontrivial and requires a deep understanding of both the source and target problems. The application of SAT solver verification provides additional support for the correctness of the gadgets, enhancing confidence in the validity of the NP-completeness proofs for the Akari variants studied. In addition to the NP-complete cases, Pulles also shows two Akari variants that lie in the P complexity class. For these variants, the author develops polynomial-time algorithms. Pulles' work demonstrates that certain restricted variants of Akari can be solved in polynomial time, making them computationally 'easier' than the classic NP-complete version. While the original Akari puzzle is NP-complete, this research identifies specific rule modifications that result in polynomial-time solvable variants. Minor rule modifications can result in notable differences in computational complexity. A case study is presented by the contrast between NP-complete and P variants within the same puzzle framework. Pulles has implemented three different algorithms to solve Akari puzzles. The algorithms are:

- A partial solution trivial solver. It works as a baseline for solving the puzzle and better understanding its mechanics.
- A SAT solver that applies the strength of SAT-solving methods by reducing Akari to a satisfiability problem.
- A systematic approach to exploration into the solution space via backtracking.

These implementations are not just academic exercises. Pulles uses them to do actual empirical tests of

the speeds with which various Akari instances are solved. The results show that the SAT solver significantly outperforms the other methods. For example, it can solve a 40 × 40 puzzle in under one second, a 30 × 30 puzzle in under half a second, and solve a 14 × 14 puzzle in about a tenth of a second. The backtrack solver takes a few seconds to up to a minute(s) when solving a simple 14 × 14 puzzle. This hands-on method offers useful information on how various problem-solving techniques perform in the real world, which may be overlooked by purely theoretical analysis. The creation of an algorithm for producing fresh Akari puzzle cases with original solutions is another noteworthy contribution of the thesis. This makes it possible to create a wide variety of puzzles with known properties. Pulles did not mention any direct comparison of the puzzle generation algorithm with other methods or human-designed puzzles. Creating puzzles with regulated properties is essential for comprehensive algorithm testing and benchmarking. That way, this constructor is useful for testing various solving algorithms. The wider consequences of Akari's NP-completeness are also examined in this work. This classification has practical significance for both puzzle designers and solvers. The author explains how it impacts the scalability of solving arbitrary Akari instances efficiently. This provides insights into what makes Akari difficult to solve by analyzing the connection between the computational complexity and the rule constraints of the puzzle. Beyond Akari, this approach offers a foundation for comprehending and categorizing additional logic puzzles. Pulles ensures that the work is grounded in a strong theoretical framework by offering comprehensive formal definitions of Akari and its variations. This formalism makes it possible to communicate the properties and restrictions of the puzzle precisely and is essential to the validity of the complexity proofs. Pulles' method of examining Akari variations systematically isolates specific rule modifications, focusing on number constraints on walls. This approach involves constructing gadgets for NP-completeness proofs of more complex variants and developing polynomial-time algorithms for simpler variants. This shows how different rule changes affect the computational complexity of the puzzle. More systematic knowledge of the relationship between game rules and computing complexity may result from this rigorous approach to variant analysis. This might also be used as a model for comparing other logic puzzles or games. It provides information on the characteristics of NP-completeness in spatial reasoning problems and illustrates methods for establishing complexity outcomes in related fields. With overall consideration, this work represents an important advancement in the study of computational complexity in puzzles. Akari's computational features are thoroughly studied here by the combination of rigorous theoretical study, proof strategies, and real-world algorithm implementation [4].

3 ASP in Akari

3.1 ASP

ASP emerged as a substitute for conventional procedural programming techniques, enabling a more natural and simple illustration of complicated problems. This method of representing knowledge is declarative. Declarative programming lets users define "what" the problem is instead of "how" to solve it, letting the solver handle the problem-solving. The foundation of ASP is logic programming, which represents knowledge using facts and logical rules. The ASP solver automatically finds solutions that meet these criteria. Atoms, Literals, and Rules are the core elements of ASP. The fundamental propositions (factual statements) that can be either true or false are called atoms. Atoms consist of a predicate symbol followed by a list of terms (constants or variables) enclosed in parentheses. Atoms and their negations make up literals. Rules are statements that specify the connections and constraints between atoms. Rules are similar to implications in propositional logic [5, 6].

Non-monotonic logic, specifically stable model semantics, is the foundation of ASP. Non-monotonic logic allows conclusions to be withdrawn in light of adding new information. For this flexibility, ASP can efficiently manage nested conditions and exceptions. The answer sets are stable models. Answer sets are groups of literals that satisfy the program's specified rules. Under the specified guidelines, every set of answers relates to a consistent interpretation of the problem domain. Grounding and solving are the two major phases in answer set computation. By replacing variables with all possible ground terms, grounding turns a program with variables into a propositional program. This process simplifies the problem for solvers to compute answer sets. After grounding, the solver computes the answer sets from the resulting propositional program. This involves determining which combinations of literals can coexist without violating any rules [5, 6].

ASP extends traditional logic programming by introducing constructs such as default negation (denoted as "not"), choice rules, and optimization statements. Choice rules allow flexibility in model generation. Optimization statements can minimize and maximize certain criteria in solutions. These provide powerful ways to represent incomplete or uncertain information. ASP excels in constraint satisfaction problems (CSP), where solutions must satisfy a set of conditions, e.g., graph problems (coloring, hamiltonian paths), scheduling, and planning etc. Constraints can eliminate infeasible solutions early, making the search efficient [5, 6].

3.2 ASP in Constructing and Solving Akari

Given ASP's foundation in declarative problem-solving, logical reasoning, and constraint satisfaction, ASP is particularly suited for logic puzzles like Akari. Akari's rules, like proper bulb placement, satisfaction of numbered cell constraints, illuminations of all white cells without conflicts, etc., can easily be expressed as logical statements. ASP allows these rules to be encoded in a clear, compact form, making puzzle construction straightforward and reducing the chance of errors.

Solving Akari involves exploring many possible configurations as the grid size increases. Akari puzzles often require trial-and-error placements. ASP's non-monotonic reasoning supports hypothetical bulb placements while applying all the game rules. This is crucial for handling puzzles where the solution isn't immediately found. Once an answer set (solution) is found, it can be automatically verified against the puzzle's rules. This ensures correctness and completeness, a vital aspect for both puzzle creation and solving. The optimization feature of ASP can give the most efficient solution to a puzzle. These features make ASP a powerful tool for both solving and constructing puzzles like Akari.

3.3 Akari Game Rules

Intending to illuminate every white cell with light, the following are the rules [7] that must be followed while playing the game.

- If a black cell isn't in the way, bulbs will light as far as they can in all four orthogonal (horizontal and vertical) directions.
- No cells are illuminated diagonally by light bulbs.
- Light does not go through black cells.
- No light bulb may illuminate another light bulb.
- The number of lights that can be positioned orthogonally adjacent to numbered black cells must match the number (clue) mentioned in the black cell. For instance, there will be a lightbulb above, below, and to the left and right of a black cell with the number 4.
- Bulbs next to a numbered black cell positioned diagonally do not count against the limit.
- Not every wall has a number on it. Any number of bulbs can be positioned next to a wall without a number, as long as all other guidelines are maintained.
- Light bulbs may be positioned in white cells not adjacent to black cells.

Next, the implementation of validators will be done using ASP. These validators are important for ensuring that Akari puzzles are valid and solvable. These work as the foundation for constructing and verifying Akari puzzles.

4 Implementation of Validators in ASP

Before designing a constructor to generate Akari puzzles, it must be ensured that the puzzles that are intended to work with are valid and well-structured. The idea of validators becomes necessary at this point. Validators work as tools to verify the correctness of Akari puzzles. They have two basic functions: validating the problem and validating the solution.

4.1 Problem Validator

The initial goal is to verify that a particular Akari puzzle is correctly constructed and complies with all relevant rules. This is referred to as the Problem Validator. The puzzle's components are reviewed by the Problem Validator to make sure it satisfies the requirements for a valid Akari game. It examines the locations of the black cells or walls, clue numbers, grid size, and cell content. For the puzzle to be solved using logic, each of these elements is crucial. For example, the puzzle might not be solvable if the grid size is insufficient or if the cells include invalid characters. Similarly, errors in black cell placement or clue numbers outside of the permissible range may result in contradictions when solving the puzzle. Using a problem validator, it can be ensured that no such problems arise before trying to solve or analyze the puzzle. By the problem validator, the following crucial components of the puzzle are verified:

- Grid Size: The validator first determines whether the grid size is appropriate. Since Akari puzzles tend to be square, it must be ensured that there are as many rows as columns. This guarantees that the puzzle is shaped correctly for solving.
- Cell Content: Then it confirms the contents in each cell. In an Akari puzzle, a cell may be empty or black (wall).
- Clue Numbers: The validator determines whether the clues in black cells containing numbers fall within the acceptable range. Clue numbers in Akari can only be 0, 1, 2, 3, or 4. These clues indicate the number of lights that have to be positioned next to the clue cell. Puzzles containing clue numbers that are in the range of 0 to 4 will be generated here.

The problem validator is designed using ASP. The important ASP rules mentioned in Listing 1 are implemented here.

Listing 1: Akari Problem Validator

```
% Grid size validation
#const size = 10.
row(1..size).
col(1..size).
cell(X,Y) :- col(X), row(Y).

% Each cell can be empty or a black cell
{ black_cell_unnumbered(C,R) : col(C), row(R) }.

% For black cells, assign a number from 0 to 4 or leave it blank
0{ black_cell(C,R,N) : N = 0..4 } 1 :- black_cell_unnumbered(C,R).
```

In the code snippet shown in Listing 1, the grid cells are generated in accordance with the grid size, which is defined by a constant named "size". A 10×10 grid is created here for the Akari puzzle in the initial step. All of the valid cell coordinates in this grid are generated by the cell(X,Y) predicate. This assures that the size of the puzzle will be constant and precise. The next rule says that every cell has the potential to be an unnumbered black cell (wall). Implicitly, cells that are not marked as black

are regarded as empty. Because of this adaptive representation, puzzles can be validated with different black-and-white cell combinations. The next rule works on giving clue numbers to the unnumbered black cells. According to this rule, an unnumbered black cell can either be left unnumbered or assigned a number between 0 and 4. Each black cell is given a maximum of one integer through the 0 {...} 1 syntax. This is identical to the Akari rules, which state that black cells may have a clue number between 0 and 4 or none at all.

4.2 Solution Validator

The second goal of validators is to verify that a suggested solution accurately resolves the puzzle and that the puzzle is solvable. This is referred to as the Solution Validator. I focus on the solution validator after using the problem validator to confirm the input puzzle's validity. While implementing the approaches to develop a solution, it must be ensured that it satisfies all Akari rules. The Solution Validator makes sure that no two bulbs illuminate one another, all clues in the black cells are satisfied with bulb placements, and verifies that all of the white cells are lit. This important part checks if a suggested solution follows all of the Akari rules and solves the puzzle appropriately, hence the puzzle is solvable. By the Solution Validator, the following crucial components of the puzzle are verified:

- Valid Placement of light bulbs: Proper light bulb placement guarantees that lights are only positioned in non-black cells and follow the game rule.
- Ensure Illumination rules of light bulbs: cells are properly illuminated when light from each light source travels in all four directions and stops at black cells.
- Satisfy Number clue constraints: this rule verifies that each numbered black cell corresponds to the number of orthogonally adjacent light bulbs, specifically from clue values 1 to 4.
- Ensuring no lights next to clue 0: this verifies that the rule referring to black cells having a clue value of 0 is being observed.
- Confirming illumination of all white cells: this rule guarantees that at least one light illuminates each non-black cell in the grid.
- Prevention of light bulbs illuminating each other: this rule is to confirm that there must be a black cell between any two lights in the same row or column for them not to illuminate one another.

Similar to the Problem Validator, ASP is used by the Solution Validator to effectively define and verify these rules. Here in the code snippet shown in Listing 2, the first rule implements the placement of lights. It allows lights to be placed in any non-black cell of the grid. In the Problem validator (Listing 1), clues were assigned to the unnumbered black cells later. So all the black cells with clues, included in predicate black_cell/3, are also included in predicate black_cell_unnumbered/2. The second set of rules implements the illumination logic. They define how light spreads in all four directions (up, down, left, right), stopping at black cells. No diagonal cells are included in the illumination relation of light and the cells. It confirms that no cells are illuminated diagonally by light bulbs. The third set of rules satisfies the clue constraints. It only counts the lights orthogonally adjacent to numbered black cells from 1 to 4 and ensures that this count matches the cell's value. So, bulbs next to a numbered black cell positioned diagonally do not count against the clue limit. The fourth set of rules ensures that no lights are placed orthogonally next to black cells with a clue value of 0. This adheres to the Akari rule that cells orthogonally adjacent to a 0-valued black cell must be empty. The fifth rule ensures complete illumination of all non-black cells. It creates a constraint that fails if any non-black cell is not illuminated. The last set of rules prevents light interference. They ensure that no two lights can illuminate each other in the same row or column without a black cell between them. Maintaining all these solution validation rules, light bulbs can be positioned in any other white cells. By implementing these rules, the solution validator ensures that any proposed solution follows all the rules of Akari mentioned in Section 3.3.

Listing 2: Akari Solution Validator

```
% Place lights in non-black cells
{ light(C,R) : col(C), row(R),  not black_cell_unnumbered(C,R) }.

% Define illumination logic
illuminated(X,Y) :- cell(X,Y), not black_cell_unnumbered(X,Y),
                    light(X,Y).
illuminated(X,Y) :- cell(X,Y), not black_cell_unnumbered(X,Y),
                    light(X1,Y), X1 < X,
                    not black_cell_unnumbered(X2,Y) : X2 = X1+1..X-1.
illuminated(X,Y) :- cell(X,Y), not black_cell_unnumbered(X,Y),
                    light(X1,Y), X1 > X,
                    not black_cell_unnumbered(X2,Y) : X2 = X+1..X1-1.
illuminated(X,Y) :- cell(X,Y), not black_cell_unnumbered(X,Y),
                    light(X,Y1), Y1 < Y,
                    not black_cell_unnumbered(X,Y2) : Y2 = Y1+1..Y-1.
illuminated(X,Y) :- cell(X,Y), not black_cell_unnumbered(X,Y),
                    light(X,Y1), Y1 > Y,
                    not black_cell_unnumbered(X,Y2) : Y2 = Y+1..Y1-1.


% Satisfy Clue constraints
count_lights(C,R,N) :-  black_cell(C,R,N), N > 0,
                        N = #count {
                            1,C-1,R : light(C-1,R);
                            1,C+1,R : light(C+1,R);
                            1,C,R-1 : light(C,R-1);
                            1,C,R+1 : light(C,R+1)}.
:- black_cell(C,R,N), N > 0, not count_lights(C,R,N).

% Ensure no lights are placed next to black cells with a value of 0
:- light(C,R), black_cell(C-1,R,0).
:- light(C,R), black_cell(C+1,R,0).
:- light(C,R), black_cell(C,R-1,0).
:- light(C,R), black_cell(C,R+1,0).

% Ensure all non-black cells are illuminated
:-  col(C), row(R), not black_cell_unnumbered(C,R),
    not illuminated(C,R).

% Prevent lights from illuminating each other
:- light(C,R), light(CC,R), C < CC,
   not black_cell_unnumbered(X,R) : col(X), X > C, X < CC.
:- light(C,R), light(C,RR), R < RR,
   not black_cell_unnumbered(C,X) : row(X), X > R, X < RR.
```

These validators work well together to ensure the production of valid Akari puzzles in a computational setting. The unwanted difficulties and mistakes can be prevented by verifying the validity of the puzzles before implementing any nice puzzle construction techniques. With both the Problem Validator and Solution Validator in place, there is now a robust system for managing Akari puzzles. These validators form the foundation upon which the puzzle construction algorithm can be built.

5 Puzzle Construction Algorithm

In this section, I introduce the Akari puzzle constructor using ASP and explain how it works. A Python-Clingo integration has been added for puzzle visualization and ASP-based encoding of the constructor. The constructor creates well-formed Akari puzzles, each with a unique solution. The uniqueness is verified in two stages of the puzzle generation process. Later, the solvability of each generated puzzle is also analyzed using a human strategies-based ASP solver.

5.1 Designing Puzzle Constructor

The process of generating Akari puzzles with unique solutions involves two main steps: puzzle generation and uniqueness verification. This two-step approach consists of several integrated components. All these components work together to ensure the creation of valid puzzles while maintaining the uniqueness of the generated puzzles. The core generation follows a constraint-based approach, where various conditions must be satisfied to produce valid and interesting puzzles. Rather than building puzzles incrementally, the ASP solver explores the solution space to find configurations that satisfy all constraints simultaneously.

The foundation of the constructor is the problem validator mentioned in Listing 1 and the solution validator from Listing 2. These verify that the generated puzzles are solvable, maintaining all Akari game rules, including proper light placement and illumination. Then, a set of carefully formulated constraints was integrated in ASP. These constraints guide the puzzle generation process towards generating nicer configurations of them. The guiding constraints are as follows:

- Distribution constraints:
  - The total number of black cells is bounded between 20% to 30% of the total number of cells in the grid.
  - At least 60% of the black cells must be concentrated in the central area to create a balanced puzzle.
- Numbered cell distribution: between 60% and 80% black cells must be numbered, ensuring enough clues for solvability without over-constraining the puzzle.
- Number type distribution: For creating a mix of clue types,
  - At least 1 cell must have a clue value of 3, and at least 1 cell must have a clue value of 4.
  - At least 60% of the black cells must have clue values of 1's or 2's.
  - At least 20% of the black cells must have 0's.

  This creates varied but solvable puzzles.
- Structural constraints: every row and column must contain at least two black cells, preventing trivial solution patterns.
- Pattern avoidance:
  - No more than 3 black cells can be adjacent, preventing overly dense black cell clusters.
  - No 2×2 areas can be completely empty, eliminating trivial solution spaces.
  - Diagonal patterns of 4 consecutive black cells are prohibited, increasing puzzle variety and challenge.

Along with the previously discussed validators, these constraints ensure a balance of difficulty and validity of the puzzles.

Next, the uniqueness of generated Akari puzzles is ensured through a Model Counting verification approach. In this, the generated puzzle is subjected to an independent Akari solver that solely contains the solution rules of Akari, which is the solution validator itself from Listing 2. It enumerates all possible models of the generated puzzle. Each model represents a distinct solution to the puzzle. Theoretically, this approach explores all possible solutions. It considers every way to place light bulbs that follows the puzzle rules. The puzzle is unique if only one model is found. This means there's only one correct way to place the light bulbs, following all the rules of the Akari puzzle. With the iterative nature of this process, puzzles failing the second verification stage are discarded, and new ones are generated. This perfectly ensures that only Akari puzzles with provably unique solutions are accepted. The next aim is to implement the whole process by integrating ASP and Python.

5.2 Implementation

In this study, the implementation of the Akari puzzle constructor design combines ASP programming with Python to generate, validate, and visualize puzzles. This section details the technical aspects of the implementation, including code structure, integration with Clingo [8], and data handling procedures.

5.2.1 ASP Constructor

The first part of the constructor is implemented as an ASP program consisting of the problem validator, some crucial logical constraints, and the solution validator. Following is the step-by-step implementation of the ASP puzzle constructor part.

- At first, the constructor uses the previously mentioned problem validator from Listing 1 to nondeterministically generate valid candidate puzzle grids.
- Then the grid is filtered by different constraints mentioned in Section 5.1. In Listing 3, all the constraints described in Section 5.1 are formulated in ASP. With the help of the grid size initialized in the problem validator (in Listing 1) and the total black cell count mentioned in the first rule of Listing 3, all the constraints are formalized here. The first set of constraints describes the black cell distribution constraints bounding them to be between 20% to 30% of the grid and concentrating 60% of the black cells in the central area of the grid. The second set of constraints limits the numbered black cells to be 60% to 80% of the total black cells. The third set of constraints distributes different types of clue numbers. At least 2 cells must have clue values, one of each numbered type 3 and 4, at least 60% of the total black cells having clue value 1 or 2, and at least 20% having clue 0. The fourth set of constraints confirms placing at least two black cells (numbered or blank) in each row and column. Finally, the last set of constraints ensures avoiding patterns like 3 adjacent black cells, an empty 2×2 area, and 4 diagonally consecutive black cells. The constraints were experimentally tuned, aiming to generate nice, challenging, and uniquely solvable Akari puzzles. The final set of constraints represents a balance between puzzle quality, difficulty, and solvability.
- Then the solution validator mentioned in Listing 2 is integrated here. This integration ensures that the generated puzzles maintain all Akari game rules to have a valid puzzle grid.

Next, this ASP constructor will be integrated into Python to carry out the whole process of puzzle generation and visualization as well.

5.2.2 Python Integration with Clingo

The ASP puzzle constructor is integrated with Python through the clingo [8] module, which provides a programmatic interface to the clingo solver. This combines the declarative power of ASP with Python's flexibility for handling input/output, visualization, and data management. The process of generating unique Akari puzzle grids involves the following steps:

Listing 3: ASP Formalization of the Puzzle Generation Constraints

```
% Count total black cells
total_black_cells(T) :- T = #count{R,C : black_cell_unnumbered(R,C)}.
% Distribution Constraints
:- #count { R,C : black_cell_unnumbered(R,C) } < (size * size)/5.
:- #count { R,C : black_cell_unnumbered(R,C) } > (size * size)*3/10.
:- #count { R,C : black_cell_unnumbered(R,C),
    R = 2..(size-1), C = 2..(size-1) } < T * 6 / 10, total_black_cells(
       T).

% Numbered cell Distribution
:- #count { R,C : black_cell(R,C,N), N > 0 } < T * 6 / 10,
    total_black_cells(T).
:- #count { R,C : black_cell(R,C,N), N > 0 } > T * 8 / 10,
    total_black_cells(T).

% Number type Distribution
%:- not 1 { black_cell(C, R, N) } , N = 3..4.
:- #count { C, R : black_cell(C, R,3) } = 0.
:- #count { C, R : black_cell(C, R,4) } = 0.

:- #count { R,C : black_cell(R,C,N), N = 1..2 } < T * 6 / 10,
    total_black_cells(T).
:- #count { R,C : black_cell(R,C,N), N = 0 } < T * 2 / 10,
    total_black_cells(T).

% Structural Constraints
:- row(R), not 2 { black_cell_unnumbered(R,C) : col(C) }.
:- col(C), not 2 { black_cell_unnumbered(R,C) : row(R) }.

% Pattern avoidance
:- #count {
    1,R-1,C : black_cell_unnumbered(R-1,C);
    1,R+1,C : black_cell_unnumbered(R+1,C);
    1,R,C-1 : black_cell_unnumbered(R,C-1);
    1,R,C+1 : black_cell_unnumbered(R,C+1);
    1,R,C   : black_cell_unnumbered(R,C)
} > 3, black_cell_unnumbered(R,C).
:- not black_cell_unnumbered(R,C),
    not black_cell_unnumbered(R+1,C),
    not black_cell_unnumbered(R,C+1),
    not black_cell_unnumbered(R+1,C+1),
    not light(R,C), not light(R+1,C),
    not light(R,C+1), not light(R+1,C+1),
    row(R), row(R+1), col(C), col(C+1).
:- black_cell_unnumbered(R,C), black_cell_unnumbered(R+1,C+1),
   black_cell_unnumbered(R+2,C+2), black_cell_unnumbered(R+3,C+3).
```

- Generating a valid candidate puzzle with nicer configuration through the ASP constructor
- Verifying Uniqueness – model counting
- Accepting only candidates with Unique solutions
- Creating visual representations of both the solved and unsolved puzzles
- Formatting the puzzle as ASP facts for further usage
- Collecting and analyzing construction performance metrics

It begins with the initialization of an AkariPuzzleGenerator class, which sets the size of the puzzle grid. Then the Clingo logic programming system creates a random puzzle configuration. This involves generating a random seed and using it to construct a valid Akari grid from the ASP constructor of Section 5.2.1. This randomization is crucial for generating diverse puzzles rather than consistently producing the same configuration. In this phase, a candidate Akari puzzle grid is generated. Next, to verify the uniqueness of the candidate puzzle, it (in the unsolved state) is taken as input into an independent ASP solver, which contains only the Akari solution validator of Listing 2. Then, using clingo, the Python script searches for solutions for the candidate and counts the number of its solution models. If the puzzle has multiple solutions, the puzzle is discarded, and the process returns to the ASP constructor to construct another candidate. This repeats until a puzzle with a unique solution is found. If only one solution is found in this model counting step for a candidate puzzle generated in the first step, the puzzle is accepted. Then, a Python imaging library - PIL [9] is used to visualize the puzzles. This involves creating an image with a grid layout, where black cells are drawn as walls, and if desired, the solution can be shown by highlighting illuminated cells and placing bulb images at the positions of the lights. In this process, a variety of unique Akari puzzle grids are constructed and added to the collection. This integration follows a structured file organization pattern for the generated solved, unsolved puzzles, and the abstract formatted ASP outputs. The abstract-formatted ASP outputs are used later as facts for analysis with the human strategy-based ASP solver. Additionally, the constructor was designed with performance evaluation in mind. The timing data of each puzzle generation was collected to analyze the average generation time required for puzzles of different sizes.

By implementing the model counting verification process, the methodology ensures the generation of Akari puzzles that are not only valid but also guaranteed to have unique solutions. This is crucial for creating engaging and fair puzzles for human solvers. More information on the structure of this Python integration is available in Section A.1 in Appendix A. The next aim is to identify and formalize humans' logical method of thinking to solve an Akari puzzle into a computer framework and construct a human strategy-based solver using ASP. Then the solver will be employed to solve the constructed puzzles and check if the constructed puzzles can be solved with this.

6 Akari Solving Strategies for Humans

Akari challenges players to light up a grid while following the game rules and satisfying the clues present in the grid. Solving them usually involves a combination of logical deductions and pattern recognition. Human solvers usually use various tactics to solve these types of puzzles quickly and effectively. Some of these strategies are simple tricks that manipulate the game mechanics. Some are more complex and involve understanding how various aspects of the grid interact with one another in different configurations. In this study, six strategies used by human solvers [10] will be discussed. These strategies can be grouped into two categories: cell exclusion strategies from bulb placement and bulb placement strategies.

Cell Exclusion Strategies from Bulb Placement

- Forced Empty Cell
- Diagonal Exclusion
- Clue Saturation-based Exclusion

Bulb Placement Strategies

- Restricted Clues
- Clue Patterns
- Limited light options

Each of these strategies focuses on a different aspect of the puzzle. They effectively help to deduce where to place light bulbs and which cells can never have a bulb. This knowledge and these techniques will allow the solvers to methodically approach any Akari puzzles that have different levels of complexity. Following the above-mentioned strategies, the solver gradually illuminates the board until it is solved. Next, in the forthcoming subsections, each strategy and its application will be elaborated in detail.

6.1 Forced Empty Cell

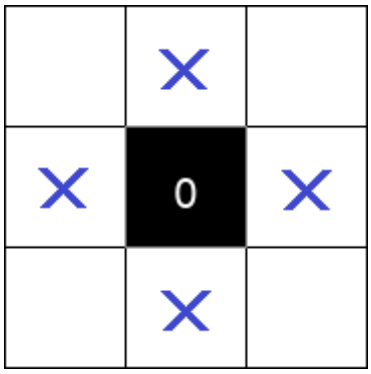


Figure 2: Representation of Forced Empty Strategy on cells adjacent to clue 0

The Forced Empty cell strategy is a primary method for solving Akari puzzles. This is a good starting point to limit the number of potential places for light bulbs when solving a new puzzle. This is very effective when used in the early stage of puzzle-solving, which can lead to subsequent deductions. This strategy is based on one fundamental: cells orthogonally adjacent to 0 clues. This strategy has one of the simplest applications on those orthogonally adjacent cells. In Akari, a 0 clue means that it cannot have any light bulbs in any of the orthogonally adjacent cells. This rule follows from the basic limit that the digit that lies on a clue cell is exactly the number of light bulbs that need to be placed in its orthogonally

adjacent cells.

As an example; in the case of a black cell with a 0 clue (Figure 2), each of the four existing orthogonally adjacent cells needs to be marked as no bulb or empty. These cells are marked with a cross sign (X). If a light bulb is placed in any of these cells, it would violate the condition of the 0 clue. This deduction significantly reduces the complexity of the problem by removing potential light bulb placements. This leads to further logical deductions about the surrounding cells.

6.2 Diagonal Exclusion

A more advanced case - Diagonal Exclusion is implemented while having clues with available empty cells diagonally adjacent to them. In some specific configurations, each clue excludes its diagonal empty cells from having light bulbs. This strategy depends on the interactions among - the value of the clues, the number of available adjacent cells, the position of the clue in the grid, and the illumination rules of the puzzle.

If the clue cell has 4, then all four adjacent orthogonal cells should contain light bulbs according to Akari rules. In this case, if there are available empty cells diagonally adjacent to this clue, they cannot have light bulbs (Figure 3a). If a light bulb is placed in any of these diagonal cells, two light bulbs will end up illuminating each other, which is prohibited by puzzle rules. So, the logical reasoning for this is as follows:

- According to the rules, all four cells orthogonally adjacent to clue 4 must have light bulbs.
- These light bulbs will light up all cells in their row and column.
- Therefore, placing a light bulb in one of the diagonal cells leads to a rule violation. Any light bulb placed in at least one of the diagonals of clue 4 will always lie in the same row or the same column as one of the needed light bulbs.

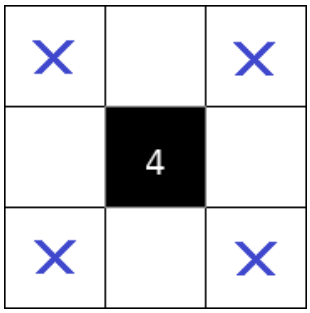
The same principle can also be applied to clue 3 cells in some specific positions. The following are three possible position specifications for this.

- Clue 3 positioned somewhere in the middle of the grid surrounded by empty cells (Figure 3b).
- Clue 3 positioned in an edge cell of the puzzle grid (Figure 3c).
- Clue 3 positioned in a non-edge cell which has an orthogonal adjacent black cell (Figure 3d).

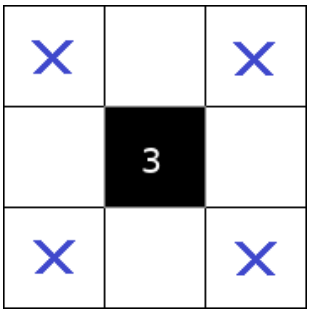
In the first case (Figure 3b), any three orthogonal adjacent cells of the clue must have light bulbs. These light bulbs will light up all cells in their row and column. With this in mind, any three positions chosen for bulb placement will illuminate all the diagonal cells. So, all the diagonal empty cells must be excluded from positioning new light bulbs (Figure 3b). Otherwise, the game rule of light bulbs not illuminating each other will be violated. In the other two cases (Figure 3c, Figure 3d), all available orthogonally adjacent cells of the clue must contain light bulbs to satisfy the clue. If a light is placed in either of the diagonal cells, that would have to be illuminated by one of the light bulbs that the clue requires. This would make it impossible to follow the puzzle rules. Following this constraint, the available empty diagonal neighboring cells must not have any light bulbs to avoid violating the game rule. This is the reasoning for the diagonal exclusions of clue 3 cells.

Some of the other possible configurations of diagonal exclusion are as follows.

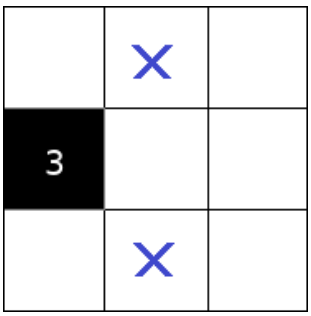
- Clue 2 positioned in an edge cell or corner cell (Figure 3e).
- Clue 1 positioned in a corner cell (Figure 3f).



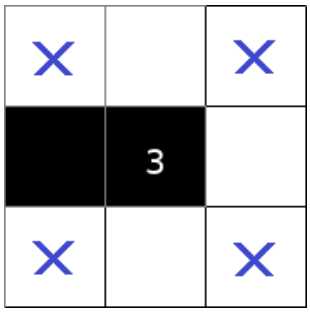
(a) Diagonal Exclusion on diagonal empty cells to clue 4



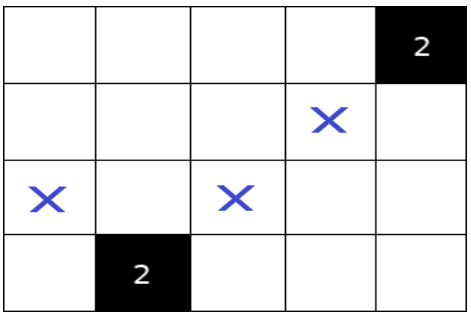
(b) Diagonal Exclusion on diagonal empty cells to clue 3 located in the middle



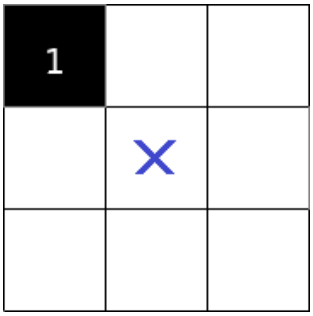
(c) Diagonal Exclusion on diagonal empty cells to clue 3 located in edge of the grid



(d) Diagonal Exclusion on diagonal empty cells to clue 3 located in non-edge cell of the grid and one side blocked



(e) Diagonal Exclusion on diagonal empty cells to clue 2 located in edges and corners of the grid



(f) Diagonal Exclusion on diagonal empty cell to clue 1 located in corner cell of the grid

Figure 3: Representation of Diagonal Exclusion Strategy

In the above-mentioned configuration of clue 2 (Figure 3e) positioned in edge cell(2,4), placing light bulbs in any two orthogonally adjacent cells will illuminate all cells in their row and column. So, the diagonal available empty cells cannot have any bulb placement. Similarly, for 2 in the corner cell(5,1) in the same figure, placing light bulbs in both orthogonally adjacent cells will also illuminate the diagonal available empty cell, as it is in the same row or column as the two lights. For similar reasoning, in the corner configuration of clue 1 represented in Figure 3f, the empty diagonal cell must be excluded from placing any light bulb.

Using Diagonal Exclusion, solvers can rapidly identify impossible light bulb positions. It also serves as crucial information about the puzzle. This strategy is often used as a gateway to more difficult deductions. It is particularly useful to solve progressively tougher Akari puzzles when combined with other solving strategies.

6.3 Clue Saturation-based Exclusion

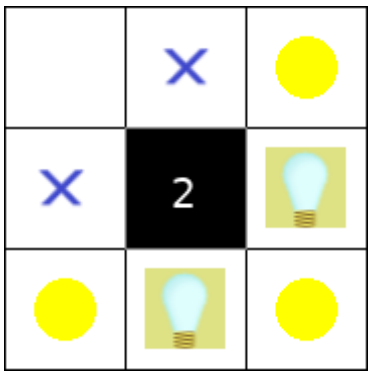


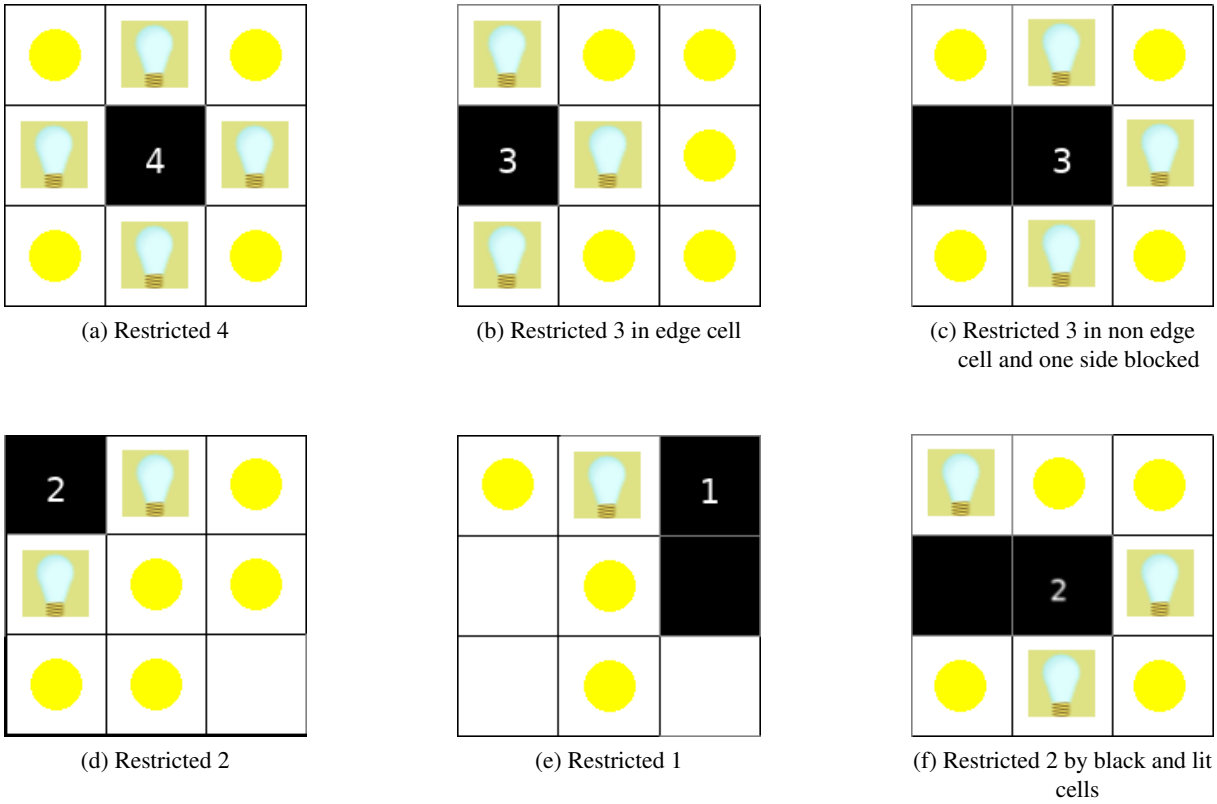
Figure 4: Representation of Clue Saturation based Exclusion Strategy

In situations when a black cell's clue number equals the count of its neighboring light bulbs, it means the clue is already satisfied. So all remaining adjacent cells must stay bulb-free. Using this technique, the Clue Saturation-based Exclusion strategy is applied. This strategy ensures numbered black cells receive exactly the required number of adjacent bulbs while preventing over-illumination. This avoids breaking puzzle rules of clue satisfaction and helps players reduce the solution space by eliminating impossible bulb placements.

For example, if a black cell with a '2' clue already has two light bulbs placed directly below and to its right, the empty orthogonal cells to the left and above must remain bulb-free. Figure 4 illustrates this case. In the larger scenario, this deduction creates a chain reaction, helping solve connected areas of the grid. The strategy mimics the way humans systematically eliminate options after satisfying numerical clues.

6.4 Restricted Clues

Another fundamental Akari solving strategy of Restricted Clues uses the numbered clues on black cells to place light bulbs unambiguously. This is one of the foundations of efficient Akari-solving approaches. This works especially when the exact number of orthogonal adjacent cells (excluding black or already lit cells) to a clue number is available. In this situation, no guesswork is needed in the bulb's placement. This mirrors the way humans recognize forced bulb placements when options become limited to satisfy a clue, preventing dead ends in puzzle solving. The strategy works alongside other rules, creating a feedback loop where one deduction enables another. I will now discuss a few cases where a clue can get restricted.



(a) Restricted 4

(b) Restricted 3 in edge cell

(c) Restricted 3 in non edge cell and one side blocked

(d) Restricted 2

(e) Restricted 1

(f) Restricted 2 by black and lit cells

Figure 5: Representation of Restricted Clues Solution Strategy

The rules already define that each number in a clue must refer to the number of bulb placements in orthogonal adjacent cells. According to that, clue 4 must have lights in all four orthogonally neighboring cells of it. This is also the highest number of light bulbs that can be placed around a single cell in Akari. So, in this case, the solver would have to put light bulbs in all of the adjacent cells - up, down, left, and right, as shown in Figure 5a. As a result of this, a large section of the grid surrounding the clue is illuminated.

A '3' clue at the edge of the puzzle grid defines a constrained condition where light bulbs need to be placed in every accessible neighboring cell. In situations like this (Figure 5b), the solver will need to put light bulbs in all three possible adjacent cells. Another such case is where one of the orthogonally adjacent cells of clue 3 is a black cell (Figure 5c). So, it has only three available adjacent cells, which must contain light bulbs. The strategy is useful in such cases as it maintains clue constraint and also eliminates candidates for other empty cells along the edge of the puzzle or the respective row and column.

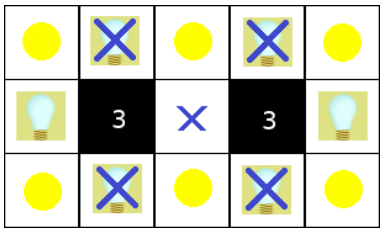
If a '2' clue is in the corner of the puzzle, both of the two orthogonally adjacent cells must have light bulbs (Figure 5d). So, the solver must place light bulbs in both adjacent cells. Another case is if a '1' clue in a corner cell has one black orthogonally adjacent cell, as shown in Figure 5e, it will end up having only one candidate to place a light bulb. Lastly, Figure 5f illustrates a situation where a '2' clue has an orthogonal black cell at its left, and the upper orthogonal cell is lit by the light bulb positioned in cell(1,1). With these, the clue got restricted by exactly 2 light bulb placement candidate cells to its right and below to satisfy the clue. In different scenarios, similar situations can occur with clue '3' and clue '1' as well, making them restricted.

The Restricted Clues strategy is very important. This technique gives absolute information about a normally fuzzy space in the grid. It is a great foundation for further deductions. It guarantees exact

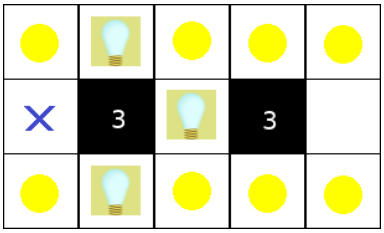
bulb locations, which saves from any guessing. The bulbs light their row and column, which leads to additional deductions. This means that the puzzle's complexity can be reduced by being certain about lighting up an area of the grid. As the puzzle solution begins, handling the constraints of the relatively few restricted clues directly removes the need to consider those portions of the grid. The added bulbs and illuminated cells with this strategy influence the rest of the puzzle. The resolution of one restricted clue can trigger a chain of deductions as well. Applying this strategy correctly, a number of the basic clues are satisfied. This confirms to avoid simple errors caused by missing these easy placements. A detailed Restricted Clues strategy serves as a foundation for solving the Akari puzzle faster. When this is employed in conjunction with other methods, this strategy enables the solver to solve complex and difficult Akari puzzles with logical precision.

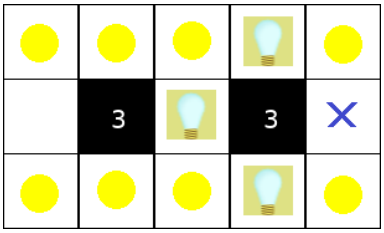
6.5 Clue Patterns

With increasingly complex and advanced Akari grids, more sophisticated approaches are needed to progress toward solving it. One such significant configuration is the pattern of two '3' clues positioned in the same row or column, separated by a single cell. In Figure 6, the application of the Clue Patterns strategy is represented with two '3' clues in the same row with one empty cell between them. This specific arrangement compels a deterministic set of light bulb placements. This considerably reduces the puzzle's complexity in its vicinity. When this pattern is encountered, the cell situated between the two '3' clues can be deduced with certainty to require a light bulb. The underlying logic is grounded in Akari's fundamental rules: each numbered cell must be adjacent to precisely that number of light bulbs, and direct illumination between light bulbs is forbidden. This deduction arises from the impossibility of satisfying both '3' clues without violating the rule against direct illumination between light bulbs if the central cell remains empty.

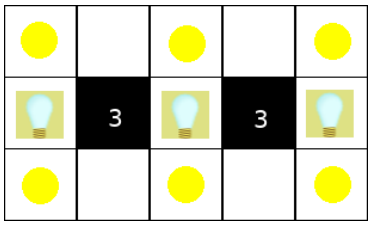


(a) Excluding the middle cell forces light bulbs to illuminate each other

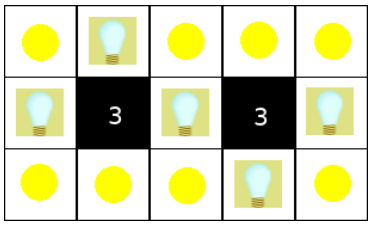




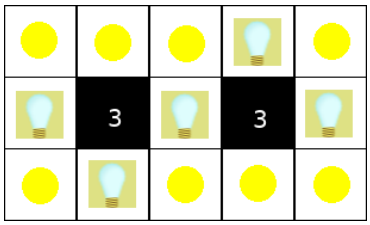
(b) Excluding either of the edge will cause such bulb placements, which will lead to illuminating at least one other bulb



(c) Therefore, these positions must be lit in such configurations



(d) One possible bulb placement



(e) Another possible bulb placement

Figure 6: Representation of Clue Patterns Strategy

As a counter to this logic, Figure 6a illustrates a scenario where it is considered not to place a light bulb in the empty intervening cell between clue 3's. In this situation, it is a must to place light bulbs in all the remaining orthogonal adjacent cells of clue 3 to satisfy the number constraints. But that violates the rule of light bulbs not illuminating each other for at least four light bulb placements. As excluding light bulb placement in the middle cell results in Akari rule violation, it can be concluded that a light

bulb must be placed in the cell that lies between the clues.

Now, each clue 3 requires two further adjacent light bulbs. Figure 6b shows that if cell(1,2) is excluded from light bulb placement possibility, then bulbs cannot be placed in either cell(4,1) or cell(4,3). But to satisfy clue 3 in cell(4,2), either one of the above-mentioned cells must have a light bulb. The same situation happens for clue 3 in cell(2,2) if cell(5,2) is excluded from the possibility of bulb placement. From these scenarios, it can be concluded that excluding either of the furthest cells (the edge cells of the middle row in the example) from bulb placement will result in at least two bulbs in the grid illuminating each other. To avoid these conflicts here, placing light bulbs at the second row's furthest extents is a must.

After these confirmed placements of light bulbs shown in Figure 6c, it is evident that four possibilities are available for the remaining two light bulb placements in the grid. In addition to simplifying the issue, this step establishes a solid framework for addressing the remaining areas that remain unresolved. In Figure 6d and Figure 6e, two possible bulb placement configurations are presented to light up the remaining unlit part of the grid.

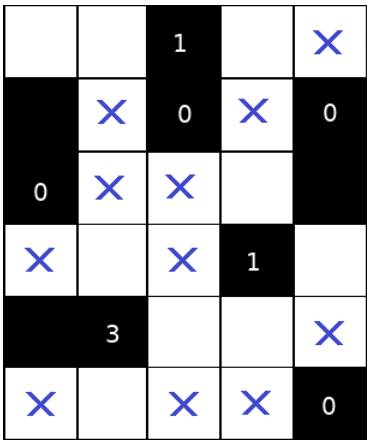
One thing that has to be confirmed while implementing this strategy is that the adjacent cells of each of these cells having light bulbs must not contain any other clue (except the '3' clue cells in the pattern). Otherwise, situations might occur in which these confirmed bulb placements can conflict with some other clue constraint nearby, and thus, the puzzle solution might become unsatisfiable. This way, the puzzle might become unsolvable by a human strategy-based solver. So, while implementing this strategy in ASP, the other adjacent cells of the light placements must be checked to confirm the application of this clue pattern strategy.

This pattern recognition can make the solving process of the Akari solver algorithm more efficient and effective. Programming the solver to recognize this configuration and correct light bulb placement leads to faster and more logical solution routes by reducing the search space. Understanding this pattern also encourages more strategic analysis regarding the connections between clues and how they affect the location of light bulbs. This strategy shifts from a cell-by-cell approach to a more integrated one. Following this strategy, the interaction of various components of the grid becomes crucial in solving the task.

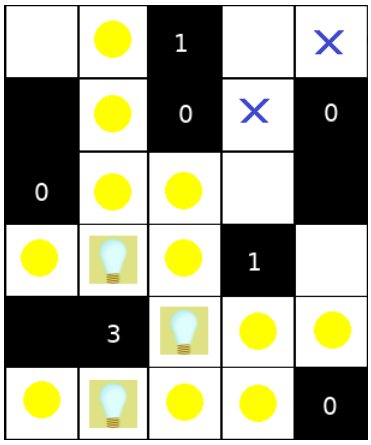
6.6 Limited Light Options

After applying the Forced Empty Cell, Diagonal Exclusion, Restricted Clues, and Clue Patterns strategies along with the basic game rules of Akari, there can be some special cells in the grid that are still not lit. The goal of the Limited Light Options strategy is to find the cells that must have a light bulb because of their special placement in the grid. The basic idea is that an unlit cell must have a light bulb in it if no other possible light bulb placement can enlighten it. This condition occurs when an unlit cell is orthogonally surrounded by the combinations of black cells, cells that are already illuminated by some other light bulbs, or cells affected by Forced Empty or Diagonal Exclusion Strategy. Under such circumstances, the only method to guarantee that this cell is lit, as required by the puzzle's regulations, is to place a light bulb in that particular cell.

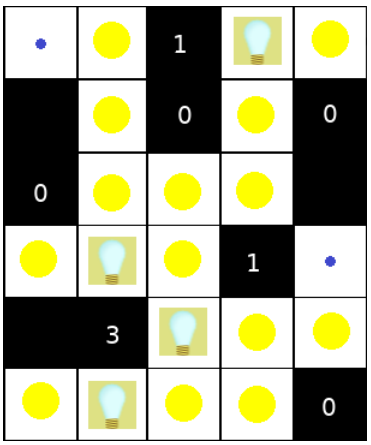
Figure 7 illustrates the occurrence and implementation of the Limited Light Options Strategy in a portion of an Akari puzzle grid. In Figure 7a, the Forced Empty and the Diagonal Exclusion Strategies are applied for clue 0 and clue 3. In this step, cells that cannot have any light placements are marked with a cross sign (X). Next, in Figure 7b, the Restricted Clue strategy is implemented for clue 3. This implementation results in lighting up a big area of the grid. After this, two clues of 1 are remaining to satisfy. Among them, clue 1 in cell(3,1) has exactly one available empty adjacent cell on the right. This makes the mentioned 1 clue restricted. So, a light bulb must be placed there to satisfy the corresponding 1 clue (shown in Figure 7c). Next, there are two empty cells available in the grid marked with a blue dot



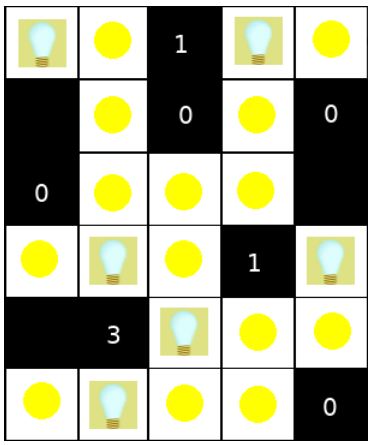
(a) After applying Forced Empty Strategy for clue 0's and Diagonal Exclusion Strategy for clue 3



(b) After applying Restricted Clue Strategy for clue 3



(c) After applying Restricted Clue Strategy for the clue 1 in cell(3,1)



(d) After applying limited light options strategy for cell(1,1) and cell(5,4)

Figure 7: Representation of Limited Light Option Strategy while combined with previously discussed strategies (step by step)

in Figure 7c. In this situation, these cells cannot be lit by any other possible light bulb placement. So, these unlit cells must have light bulbs in them to complete the solution, as shown in Figure 7d.

Many Akari puzzles can be solved by using a combination of these six techniques. Each one provides a different method for logical deduction of where lights must and must not be placed. The Forced Empty Cells strategy shows where lights cannot be placed because of violating cell clue constraints. The Diagonal Exclusion relies on the light propagation in geometry to prune possible light placements. The Restricted Clue strategy depends on fulfilling the numbers of a clue with only one possible variation. The Clue Patterns identify common arrangements that result in predictable answers. Lastly, the Limited Light Options concentrates on possible suspects in areas where there aren't many other options. The next aim is to formalize this logical method of thinking into a computer framework using ASP.

7 Formalization of Human Solving Strategies in ASP

In this section, I formalize the six previously discussed human-solving strategies for Akari puzzles(in Section 6) in ASP. These formalizations translate intuitive human strategies into logical rules. These strategies will be processed in an ASP solver combined with a solution validator, bridging the gap between human intuition and computational problem-solving. The ASP solver will later be used to solve the constructed Akari puzzles.

Before the formalization, all black cells "black_cell_unnumbered/2" and "black_cell/3" are combined in one predicate "black/2" despite them having clue values or being unnumbered to improve clarity and ease of later implementation. Listing 4 represents this.

Listing 4: Predicate aliases used to combine all black cells

```
black(C, R) :- black_cell_unnumbered(C, R).
black(C, R) :- black_cell(C, R, _).
```

7.1 Forced Empty Cell

Next in Listing 5, the "Forced Empty Cells" strategy (discussed in Section 6.1) has been formalized. This formalization captures the strategy of identifying cells that cannot contain light bulbs due to their adjacency to black cells with a clue value of 0. The first ASP rule states that for any cell(C, R) that is horizontally adjacent to a black cell with a value of 0, no light bulb can be placed in cell(C,R). The second rule does the same for vertically adjacent cells. The $|C - C1| = 1$ and $|R - R1| = 1$ conditions ensure that we're only considering immediately adjacent cells. The cell(C, R) condition ensures that we're only applying this rule to valid cells on the grid. This formalization effectively implements the strategy by marking all cells adjacent to a 0-valued black cell as unsuitable for light bulb placement, which is exactly what a human solver would do when encountering such a situation (Section 6.1).

Listing 5: ASP Formalization of the "Forced Empty Cells" Strategy

```
% Human-solving strategy 1: Forced Empty Cells adjacent to 0
no_lightbulb(C, R, 0) :- black_cell(C1, R, 0),
                         |C - C1| = 1, cell(C, R).
no_lightbulb(C, R, 0) :- black_cell(C, R1, 0),
                         |R - R1| = 1, cell(C, R).
```

7.2 Diagonal Exclusion

The formalization in Listing 6 implements the Diagonal Exclusion strategy, discussed in Section 6.2. This strategy excludes diagonal cells from light bulb placement based on the clue values and positions of black cells. The first set of rules describes that for black cells with a clue value of 4, all diagonal cells are excluded from light bulb placement. The second set of rules applies in the case of black cells with a clue value of 3, all diagonal cells are also excluded. This rule set covers all special configurations of clue 3, discussed in Figure 3b, Figure 3c, and Figure 3d. The next set of rules formalizes the cases of black cells with a value of 2 that are on the edges or at the corners of the grid. Here, also, the diagonal available cells are excluded from placing light bulbs. For black cells with a value of 1 that are in the corners of the grid, the diagonal cell is excluded from placing light bulbs. The formalizations use three helper predicates: edge_cell/2, corner/2, and diagonal_cell/4. These predicates define special positions like edge cells, corner cells on the grid, and diagonal cells for each corner. These allow the rules to accurately represent the strategy for different scenarios. This whole implementation collectively

Listing 6: ASP Formalization of the "Diagonal Exclusion" Strategy

```
% Human-solving strategy 2: Diagonal Exclusion
% For black cells with a 4
no_lightbulb(C+1, R+1, 4) :- black_cell(C, R, 4),
                             cell(C+1, R+1), not black(C+1, R+1).
no_lightbulb(C-1, R+1, 4) :- black_cell(C, R, 4),
                             cell(C-1, R+1), not black(C-1, R+1).
no_lightbulb(C+1, R-1, 4) :- black_cell(C, R, 4), cell(C+1, R-1),
                             not black(C+1, R-1).
no_lightbulb(C-1, R-1, 4) :- black_cell(C, R, 4), cell(C-1, R-1),
                             not black(C-1, R-1).

% For black cells with a 3
no_lightbulb(C+1, R+1, 3) :- black_cell(C, R, 3), cell(C+1, R+1),
                             not black(C+1, R+1).
no_lightbulb(C-1, R+1, 3) :- black_cell(C, R, 3), cell(C-1, R+1),
                             not black(C-1, R+1).
no_lightbulb(C+1, R-1, 3) :- black_cell(C, R, 3), cell(C+1, R-1),
                             not black(C+1, R-1).
no_lightbulb(C-1, R-1, 3) :- black_cell(C, R, 3), cell(C-1, R-1),
                             not black(C-1, R-1).

% For black cells on the edge with a 2
no_lightbulb(C+1, R+1, 2) :- black_cell(C, R, 2), edge_cell(C,R),
                             cell(C+1, R+1), not black(C+1, R+1).
no_lightbulb(C-1, R+1, 2) :- black_cell(C, R, 2), edge_cell(C,R),
                             cell(C-1, R+1), not black(C-1, R+1).
no_lightbulb(C+1, R-1, 2) :- black_cell(C, R, 2), edge_cell(C,R),
                             cell(C+1, R-1), not black(C+1, R-1).
no_lightbulb(C-1, R-1, 2) :- black_cell(C, R, 2), edge_cell(C,R),
                             cell(C-1, R-1), not black(C-1, R-1).
% Define edge cells
edge_cell(C, R) :- cell(C, R), C = 1.
edge_cell(C, R) :- cell(C, R), C = n.
edge_cell(C, R) :- cell(C, R), R = 1.
edge_cell(C, R) :- cell(C, R), R = n.

% For black cells in the corners with a 1
no_lightbulb(C+DC, R+DR, 1) :- black_cell(C, R, 1), corner(C,R),
                               diagonal_cell(C, R, DC, DR),
                               cell(C+DC, R+DR),
                               not black(C+DC, R+DR).
% Define corner cells
corner(1,1). corner(1,n). corner(n,1). corner(n,n).
% Define diagonal cells for each corner
diagonal_cell(1, 1, 1, 1).
diagonal_cell(1, n, 1, -1).
diagonal_cell(n, 1, -1, 1).
diagonal_cell(n, n, -1, -1).
```

Listing 7: ASP Formalization of the "Clue Saturation based Exclusion" Strategy

```
% Human-solving strategy 3: Clue Saturation-based Exclusion
no_lightbulb(C1, R1) :-black_cell(C, R, N),
                       N > 0,
                       count_adjacent_lights(C, R, N),
                       adjacent(DC, DR),
                       C1 = C + DC,
                       R1 = R + DR,
                       cell(C1, R1),
                       not lightbulb(C1, R1),
                       not black(C1, R1).

count_adjacent_lights(C, R, N) :- black_cell(C,R,N), N > 0,
                                  #count {
                                      DC,DR : adjacent(DC,DR),
                                      lightbulb(C+DC, R+DR),
                                      cell(C+DC, R+DR),
                                      not black(C+DC, R+DR)
                                  } = N.
```

captures the human reasoning process of eliminating diagonal cells based on the values of black cells and their positions on the grid effectively.

7.3 Clue Saturation-based Exclusion

The formalization in Listing 7 implements the Clue Saturation-based Exclusion strategy, discussed in Section 6.3. This strategy translates to ASP through two core rules. The count_adjacent_lights/3 predicate verifies whether a numbered black cell already has enough adjacent bulbs. It uses ASP's count function to tally existing bulbs in orthogonal cells (up, down, left, right) of black cells with clues. If the count matches the number of the clue (e.g., finds exactly 2 bulbs for a '2' clue), the rule of no_lightbulb/2 predicate activates. This rule identifies all remaining empty cells orthogonally adjacent to the clue and blocks bulb placement in those cells. The implementation uses arithmetic (C+DC, R+DR) to locate neighboring cells and 'not' constraints to enforce exclusions. This directly replicates the human process of elimination logic through automated constraints.

7.4 Restricted Clues

Next in Listing 8, the Restricted Clues strategy (discussed in Section 6.4) has been formalized. The implementation precisely targets orthogonal neighbors. For each black cell with a positive number N, the rule forced_by_strategy3 checks two conditions:

- whether the clue number of a black cell matches its remaining valid adjacent cell count
- whether those remaining valid cells are not in no_lightbulb_or_black/2

If the conditions hold, the rule forces bulb placement. The #count aggregate function counts the number of valid adjacent cells. The helper predicate no_lightbulb_or_black/2 identifies invalid light bulb positions by combining multiple constraints: cells excluded from bulb placements (by previous strategies), black cells, illuminated cells only by some other light bulbs, and grid boundaries. The adjacent/2 predicate defines the four orthogonally adjacent positions. This formalization effectively captures the human reasoning of placing light bulbs when there is only one way to satisfy the constraint of a numbered black cell.

Listing 8: ASP Formalization of the "Restricted Clues" Strategy

```
% Human-solving strategy 4: Restricted Clues
lightbulb(C, R) :- forced_by_strategy4(C, R).

forced_by_strategy4(C+DC, R+DR) :- black_cell(C, R, N), N > 0,
                        #count{ DC1,DR1 : cell(C+DC1, R+DR1),
                        not black(C+DC1, R+DR1),
                        adjacent(DC1,DR1) } = N,
                        cell(C+DC, R+DR), not black(C+DC, R+DR),
                        adjacent(DC,DR).
forced_by_strategy4(C+DC, R+DR) :- black_cell(C, R, N), N > 0,
                        #count{ DC1,DR1 : cell(C+DC1, R+DR1),
                        not no_lightbulb_or_black(C+DC1, R+DR1),
                        adjacent(DC1,DR1) } = N, cell(C+DC, R+DR),
                        not no_lightbulb_or_black(C+DC, R+DR),
                        adjacent(DC,DR).

% Define cells which are already illuminated or black
no_lightbulb_or_black(C, R) :- cell(C, R), no_lightbulb(C, R, _).
no_lightbulb_or_black(C, R) :- cell(C, R), no_lightbulb(C, R).
no_lightbulb_or_black(C, R) :- cell(C, R), black(C, R).
no_lightbulb_or_black(C, R) :- cell(C, R), illuminated(C, R),
                               not lightbulb(C,R).
no_lightbulb_or_black(C, R) :- cell(C1, R1),
                               C = C1 + DX,
                               R = R1 + DY,
                               DX = -1..1, DY = -1..1,
                               |DX| + |DY| = 1,
                               not cell(C, R).

% Define adjacent cells
adjacent(0,1). adjacent(0,-1). adjacent(1,0). adjacent(-1,0).
```

7.5 Clue Patterns

The formalization in Listing 9 implements the Clue Patterns strategy for clue '3', discussed in Section 6.5. This specifically works for the pattern of two black cells with clue value '3's' separated by one non-black cell. This strategy enforces bulb placements in specific positions when detecting vertical or horizontal '3-3' clue patterns. For vertical patterns, it places bulbs at adjacent (R-1/R+1) or three cells away (R-3/R+3) from the pattern's center column. Similarly, horizontal patterns trigger bulb placements at adjacent (C1) or distant (C3) positions along their row axis. The implementation uses two separate predicates: vertical_3_3_pattern/2 and horizontal_3_3_pattern/2 to identify these configurations.

The next four helper rules identify the cell between two black cells with clue value '3's', either vertically or horizontally. The next two rules identify the farthest opposite cells of a vertical '3-3' pattern. The last two rules here perform similar identification in the case of horizontal '3-3' patterns as shown in Figure 6c. While identifying the eligible cells to place light bulbs in these patterns in ASP, some additional nearby cells around the pattern are checked to ensure that no other clue is positioned in those nearby cells that might contradict the light placements of this strategy.

Listing 9: ASP Formalization of the "Clue Patterns" Strategy

```
% Human-solving strategy 5: Clue Patterns

lightbulb(C, R) :- forced_by_strategy5(C, R).

% Force bulb placements in middle cells for '3-3' patterns
forced_by_strategy5(C, R-1) :- vertical_3_3_pattern(C, R-1).
forced_by_strategy5(C, R+1) :- vertical_3_3_pattern(C, R+1).
forced_by_strategy5(C-1, R) :- horizontal_3_3_pattern(C-1, R).
forced_by_strategy5(C+1, R) :- horizontal_3_3_pattern(C+1, R).

% Force bulb placements in farthest cells for '3-3' patterns
forced_by_strategy5(C, R-3) :- vertical_3_3_pattern(C, R-3).
forced_by_strategy5(C, R+3) :- vertical_3_3_pattern(C, R+3).
forced_by_strategy5(C-3, R) :- horizontal_3_3_pattern(C-3, R).
forced_by_strategy5(C+3, R) :- horizontal_3_3_pattern(C+3, R).

vertical_3_3_pattern(C, R-1) :- black_cell(C,R,3),
    black_cell(C,R-2,3), cell(C,R-1), not black(C,R-1), cell(C-1,R-1),
    not black(C-1,R-1), cell(C+1,R-1), not black(C+1,R-1).
vertical_3_3_pattern(C, R+1) :- black_cell(C,R,3),
    black_cell(C,R+2,3), cell(C,R+1), not black(C,R+1), cell(C-1,R+1),
    not black(C-1, R+1), cell(C+1, R+1), not black(C+1, R+1).
horizontal_3_3_pattern(C-1, R) :- black_cell(C,R,3),
    black_cell(C-2,R,3), cell(C-1,R), not black(C-1,R), cell(C-1,R-1),
    not black(C-1, R-1), cell(C-1, R+1), not black(C-1, R+1).
horizontal_3_3_pattern(C+1, R) :- black_cell(C,R,3),
    black_cell(C+2,R,3), cell(C+1,R), not black(C+1,R), cell(C+1,R-1),
    not black(C+1, R-1), cell(C+1, R+1), not black(C+1, R+1).

vertical_3_3_pattern(C, R-3) :- black_cell(C, R, 3),
    black_cell(C, R-2, 3), cell(C, R-3), not black(C, R-3),
    cell(C-1,R-3), not black(C-1,R-3), cell(C+1,R-3),
    not black(C+1,R-3), cell(C,R-4), not black(C,R-4), cell(C-1,R-1),
    not black(C-1,R-1), cell(C+1,R-1), not black(C+1,R-1).
vertical_3_3_pattern(C, R+3) :- black_cell(C, R, 3),
    black_cell(C, R+2, 3), cell(C, R+3), not black(C, R+3),
    cell(C-1,R+3), not black(C-1,R+3), cell(C+1,R+3),
    not black(C+1,R+3), cell(C,R+4), not black(C,R+4), cell(C-1,R+1),
    not black(C-1, R+1), cell(C+1, R+1), not black(C+1, R+1).
horizontal_3_3_pattern(C-3, R) :- black_cell(C, R, 3),
    black_cell(C-2, R, 3), cell(C-3, R), not black(C-3, R),
    cell(C-3,R+1), not black(C-3,R+1), cell(C-3,R-1),
    not black(C-3,R-1), cell(C-4,R), not black(C-4,R), cell(C-1,R-1),
    not black(C-1, R-1), cell(C-1, R+1), not black(C-1, R+1).
horizontal_3_3_pattern(C+3, R) :- black_cell(C, R, 3),
    black_cell(C+2, R, 3), cell(C+3, R), not black(C+3, R),
    cell(C+3,R+1), not black(C+3,R+1), cell(C+3,R-1),
    not black(C+3,R-1),cell(C+4,R), not black(C+4,R), cell(C+1,R-1),
    not black(C+1, R-1), cell(C+1, R+1), not black(C+1, R+1).
```

Listing 10: ASP Formalization of the "Limited Light Options" Strategy

```
% Human-solvable strategy 6: Limited light options
lightbulb(C, R) :- forced_by_strategy6(C, R).

forced_by_strategy6(C, R) :- cell(C, R), not black_cell(C, R),
                        not illuminated_by_other(C, R),
                        surrounded_by_unusable(C, R).
illuminated_by_other(C, R) :- illuminated_by(C, R, C1, R1),
                            (C1, R1) != (C, R).
illuminated_by(X, Y, X, Y) :- cell(X, Y), not black_cell(X, Y),
                            lightbulb(X, Y). % Self-illumination
illuminated_by(X, Y, X1, Y) :-  cell(X, Y), not black_cell(X, Y),
                            lightbulb(X1, Y), X1 < X,
                            not black_cell(X2, Y) : X2 = X1+1..X-1.
illuminated_by(X, Y, X1, Y) :- cell(X, Y), not black_cell(X, Y),
                            lightbulb(X1, Y), X1 > X,
                            not black_cell(X2, Y) : X2 = X+1..X1-1.
illuminated_by(X, Y, X, Y1) :- cell(X, Y), not black_cell(X, Y),
                            lightbulb(X, Y1), Y1 < Y,
                            not black_cell(X, Y2) : Y2 = Y1+1..Y-1.
illuminated_by(X, Y, X, Y1) :- cell(X, Y), not black_cell(X, Y),
                            lightbulb(X, Y1), Y1 > Y,
                            not black_cell(X, Y2) : Y2 = Y+1..Y1-1.
adjacent_cells(C, R, C-1, R) :- cell(C, R), cell(C-1, R).
adjacent_cells(C, R, C+1, R) :- cell(C, R), cell(C+1, R).
adjacent_cells(C, R, C, R-1) :- cell(C, R), cell(C, R-1).
adjacent_cells(C, R, C, R+1) :- cell(C, R), cell(C, R+1).

surrounded_by_unusable(C, R) :- cell(C, R),
            unusable_for(C-1, R, C, R) : adjacent_cells(C, R, C-1, R);
            unusable_for(C+1, R, C, R) : adjacent_cells(C, R, C+1, R);
            unusable_for(C, R-1, C, R) : adjacent_cells(C, R, C, R-1);
            unusable_for(C, R+1, C, R) : adjacent_cells(C, R, C, R+1).

unusable_for(C1, R1, C, R) :- cell(C,R), C1 = C + DX,
                        R1 = R + DY, DX = -1..1, DY = -1..1,
                        |DX| + |DY| = 1, not cell(C1, R1).
unusable_for(C1, R1, C, R) :- adjacent_cells(C, R, C1, R1),
                        cell(C1, R1), black(C1, R1).
unusable_for(C1, R1, C, R) :- adjacent_cells(C, R, C1, R1),
                        cell(C1, R1), no_lightbulb(C1, R1).
unusable_for(C1, R1, C, R) :- adjacent_cells(C, R, C1, R1),
                        cell(C1, R1), no_lightbulb(C1, R1, _).
unusable_for(C1, R1, C, R) :- adjacent_cells(C, R, C1, R1),
                        cell(C1, R1), illuminated_by(C1, R1, X, Y),
                        (X, Y)!= (C, R), (X, Y)!= (C1, R1).
```

7.6 Limited Light Options

Next, in Listing 10, the Limited Light Options strategy (discussed in Section 6.6) is formalized. This strategy forces light bulb placement if a non-black cell is not already illuminated by light bulbs placed in any other cells and is surrounded by unusable cells – excluded from bulb placements (by previous strategies), black cells, already illuminated cells by some other light bulbs, or grid boundaries. To confirm such a surrounding, the helper predicate surrounded_by_unusable/2 is used, which relies on the unusable_for/4 predicate. Here, unusable_for(C1, R1, C, R) indicates that the cell at coordinates (C1, R1) is unusable for placing a bulb with respect to the target cell(C, R). Cell(C1, R1) is adjacent to cell(C, R) and, for one of several specified reasons, cannot be used to illuminate (C, R). This directionality ensures clarity regarding which cell is being evaluated as unusable and for which target cell. The illuminated_by/4 predicate, used in the body of illuminated_by_other/2, keeps track of the light sources for illuminated cells. The predicate illuminated_by(X, Y, X1, Y1) expresses that the cell(X, Y) is illuminated by a light bulb located at cell(X1, Y1). This explicit pairing distinguishes the illuminated cell from its illuminating source, which is essential for correctly determining whether cell(X, Y) is already illuminated by a bulb placed elsewhere. This complete formalization effectively captures the human reasoning of placing a light bulb in a cell when it's the only option left to illuminate that cell.

These formalizations show the expressiveness of ASP to encode the previously discussed solving logics used by humans for solving the Akari puzzle. Next, how these formalized strategies are used to verify the human solvability of the previously constructed puzzles will be discussed.

8 Human strategy based-Solver

In this section, I discuss the construction of the human strategy-based solver in ASP. With this solver, I also discuss the process (combined with Python) of solving the puzzles produced by the puzzle constructor.

8.1 Human strategy-based Solver with Validator in ASP

To construct the human strategy-based solver from the previously formalized strategies, a validator is required. The validator acts as a quality filter, maintaining alignment between the solver's deductive processes and human cognitive patterns. It ensures puzzle solvability by – deduction of bulb placements using strictly human-applicable strategies (eliminating guesswork), preventing conflicts between valid and invalid bulb placements, and ensuring solution completeness with the basic Akari illumination rules. The Forced Empty strategy and the Restricted Clue strategy already covered the clue satisfaction rule.

At first, grid structure definitions (size, row, col, cell) establish the puzzle's dimensions and valid coordinates. The light bulb placement rule restricts placements to non-black cells while respecting strategic requirements through required_by_strategy/2 predicates. The illumination logic uses directional checks for each band and ensures uninterrupted horizontal/vertical paths between light sources and cells illuminated by light bulbs. The next set of rules prevents bulbs from illuminating each other. From the previous section, it is evident that amongst the six implemented human-solving strategies, the first three strategies work on eliminating cells with invalid light placement options and reducing the search space. But they don't enforce any light bulb placement rule. On the other hand, the last three strategies have light bulb enforcement rules. So the validator integrates forced_by_strategy$X$/3 predicates with $X = 4, 5, 6$ as the rule bodies for the required_by_strategy/2 predicate in the next set of rules of the validator. The last set of rules prevents the solver from any kind of conflicts in the solution, such as: valid-invalid bulb placement on the same cell, no bulb placement beyond the strategies, and finally, ensuring the full grid illumination. This ensures that solutions adhere to both fundamental Akari rules and human deduction patterns.

The solver integrates the validator described in Listing 11 and the predicate alias from Listing 4 with the ASP formalizations of the human solving strategies discussed in Section 7. So the solver is constructed by combining Listing 4, 5, 6, 7, 8, 9, 10 and 11. In Section 5.2.2, it was discussed that the Python Integrator of the puzzle constructor generates Akari puzzles and saves the unsolved state ASP show outputs in separate files for each generated puzzle. These output abstracts serve as input facts for the ASP human strategy-based solver. The unsolved puzzle representations are then processed through the solver and checked to see if the solver can produce solutions to the unique puzzles.

8.2 Python Integration with Clingo

To implement the ASP solver, another Python script is used, integrating Clingo [8]. It involves the following steps to ensure that the constructed puzzles by the constructor are also human-solvable:

- Load puzzle configurations from ASP fact files from the generator
- Apply an ASP human strategy-based solver to find solutions
- Check solution correctness and uniqueness
- Generate visual representations of the solved puzzles
- Collect and analyze performance metrics

At first, the solver loads the saved puzzle configuration as facts from files saved in the puzzle construction process. These configurations specify the locations of black cells and also the clue constraints. For each

Listing 11: Human Solver Validator

```
% Grid size validation
#const size = 10.
row(1..size).
col(1..size).
cell(X,Y) :- col(X), row(Y).

% Light placement required by strategies
{ lightbulb(C, R) : cell(C, R), not black(C, R),
        required_by_strategy(C, R)
} :- not black(C, R).

% Illumination logic
illuminated(X,Y) :- cell(X,Y), not black(X,Y), lightbulb(X1,Y),
                    X1 < X, not black(X2,Y) : X2 = X1+1..X-1.
illuminated(X,Y) :- cell(X,Y), not black(X,Y), lightbulb(X1,Y),
                    X1 > X, not black(X2,Y) : X2 = X+1..X1-1.
illuminated(X,Y) :- cell(X,Y), not black(X,Y), lightbulb(X,Y1),
                    Y1 < Y, not black(X,Y2) : Y2 = Y1+1..Y-1.
illuminated(X,Y) :- cell(X,Y), not black(X,Y), lightbulb(X,Y1),
                    Y1 > Y, not black(X,Y2) : Y2 = Y+1..Y1-1.

% Preventing lights from illuminating each other
:- lightbulb(C, R), lightbulb(CC, R), C < CC,
   not black(X,R) : col(X), X > C, X < CC.
:- lightbulb(C, R), lightbulb(C, RR), R < RR,
   not black(C,X) : row(X), X > R, X < RR.

% Strategy requirement definitions
required_by_strategy(C, R) :- forced_by_strategy4(C, R).
required_by_strategy(C, R) :- forced_by_strategy5(C, R).
required_by_strategy(C, R) :- forced_by_strategy6(C, R).

% Conflict prevention
:-lightbulb(C,R), no_lightbulb(C,R,_).
:-lightbulb(C,R), no_lightbulb(C,R).
:-lightbulb(C,R), not required_by_strategy(C,R).
:- row(R), col(C), not black(C, R), not lightbulb(C,R),
   not illuminated(C, R).
```

puzzle, the solver attempts to find a solution that satisfies all Akari rules including the human strategy-based rules. When a solution is found, the program creates an image showing the solved puzzle. This image displays the grid, black cells, light bulbs, and illuminated cells. The solver also validates solutions by comparing them with the original generator's solutions. It tracks in case the human strategy-based solver finds a different solution. This is technically not possible as the puzzles are unique by construction. This checking ensures that the human strategy-based Akari solver works properly. The program measures solving time for each puzzle and calculates average performance metrics. The solver organizes results into separate folders for the solvable and the unsolvable puzzles by humans. Please see Section A.2 in Appendix A for more details.

9 Results and Discussions

This chapter presents the outcomes of the experiments conducted to evaluate the efficiency of the Akari puzzle generator and the effectiveness of the human-strategy-based solver on the generated puzzles. The discussions on the results are also included here. The experiments were conducted in a virtualized environment on a laptop host machine. The host machine runs Windows 11 Home Single Language (Version 23 H2) and is equipped with an Intel Core i5-8265U CPU @ 1.60GHz (1.80 GHz) and 8GB of RAM (7.82 GB usable). The virtual machine was created using Oracle VM VirtualBox manager and runs Ubuntu 22.04 (64-bit) as the guest operating system. The virtual machine was allocated 4 virtual processors and 5,132 MB of base memory to execute the experiments. The integration was implemented in Python 3.10 within the virtual machine environment.

9.1 Puzzle Construction

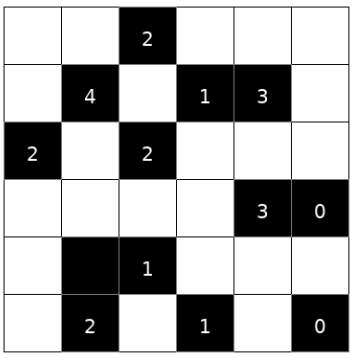
In this thesis, I have constructed an Akari generator capable of generating puzzles with unique solutions. I also constructed another human strategy-based Akari solver to ensure that the constructed puzzles are solvable by humans. Both constructions were written in ASP language, integrating it with the Python Clingo solver.

The two most important objectives of this thesis were to construct puzzles as human human-solvable and each having a unique solution. The constructor, therefore, fulfills the uniqueness of the constructed puzzles with the ASP rules, constraints, validators, and the model-counting based uniqueness verification. The construction of Akari puzzles with unique solutions was employed to 9 different grid sizes: 6×6, 7×7, 8×8, 9×9, 10×10, 11×11, 12×12, 13×13, and 14×14. For each grid size, 100 distinct puzzles, each with a unique solution, were constructed. Figure 8 shows one example of the constructed unique puzzle grid for each size. With the verification approach used in the puzzle construction process, the constructor ensures the uniqueness of each of the 900 constructed puzzles. This fulfills the first part of the goal of this thesis.

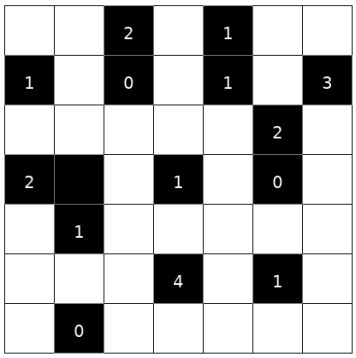
9.2 Constructor Performance

The generation process was timed for each puzzle to assess the efficiency of the algorithm across 9 different puzzle sizes mentioned in Section 9.1. Table 1 presents the average puzzle generation time for each different grid size. From the constructor performance, it is evident that the relationship between grid size and generation time appears to be non-linear, with a more pronounced increase in time for larger grids. As the grid size increases, the generation time grows at a faster rate than the grid size itself. For example, doubling the grid size from 6×6 to 12×12 results in more than double the generation time. While comparing the construction time of grid size 12×12 with 14×14, the average construction time takes a big upward shift from around 3 minutes to around 39 minutes. This non-linear behavior is common in computational problems where larger grid sizes lead to larger search spaces or more complex calculations. For the Akari puzzle constructor, this reflects the increased complexity of finding puzzle configurations with unique solutions as the grid size grows.

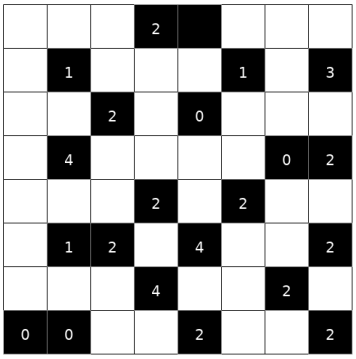
The relationship between puzzle grid size and construction time is visualized in Figure 9. Due to the wide range of construction times, spanning from fractions of a second to over 39 minutes, a logarithmic scale is used for the y-axis. This logarithmic representation allows us to visualize both small and large values on the same graph effectively. Each vertical step on the y-axis represents a 10-fold increase in construction time. The upward nature of the graph of this relationship demonstrates an exponential increase in construction time as the grid size grows. This trend highlights the computational complexity of generating Akari puzzles. Larger grids require significantly more time to construct due to the increased number of possible configurations and constraints that must be satisfied during generation. Each increment in grid size leads to a disproportionate increase in the time required for puzzle construction.



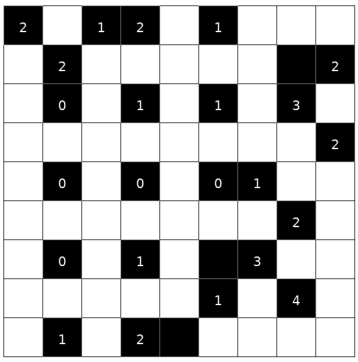
(a) 6×6 example grid



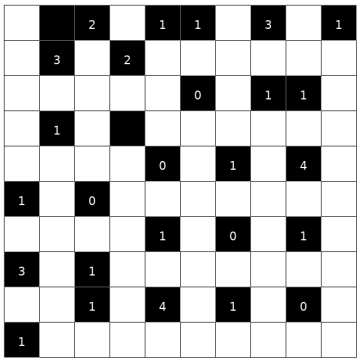
(b) 7×7 example grid



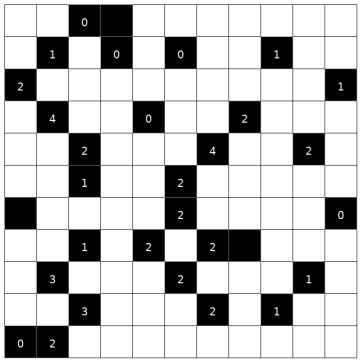
(c) 8×8 example grid



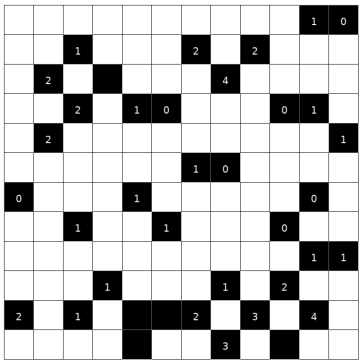
(d) 9×9 example grid



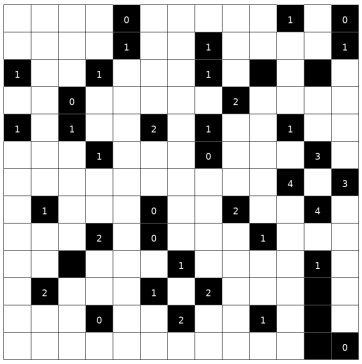
(e) 10×10 example grid



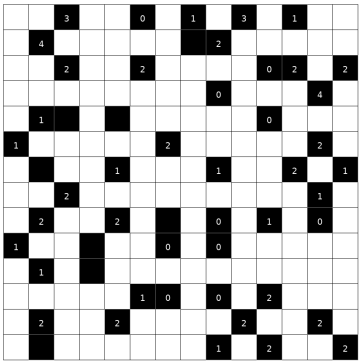
(f) 11×11 example grid



(g) 12×12 example grid



(h) 13×13 example grid



(i) 14×14 example grid

Figure 8: Example grids of unsolved puzzles for each grid size 6×6, 7×7, 8×8, 9×9, 10×10, 11×11, 12×12, 13×13, and 14×14

Puzzle Grid Size	Average Construction Time
6×6	0 min 0.16 sec
7×7	0 min 0.66 sec
8×8	0 min 0.80 sec
9×9	0 min 3.67 sec
10×10	0 min 5.81 sec
11×11	0 min 30.11 sec
12×12	3 min 17.73 sec
13×13	16 min 59.53 sec
14×14	39 min 23.69 sec

Table 1: A table with average puzzle construction times for nine different grid sizes

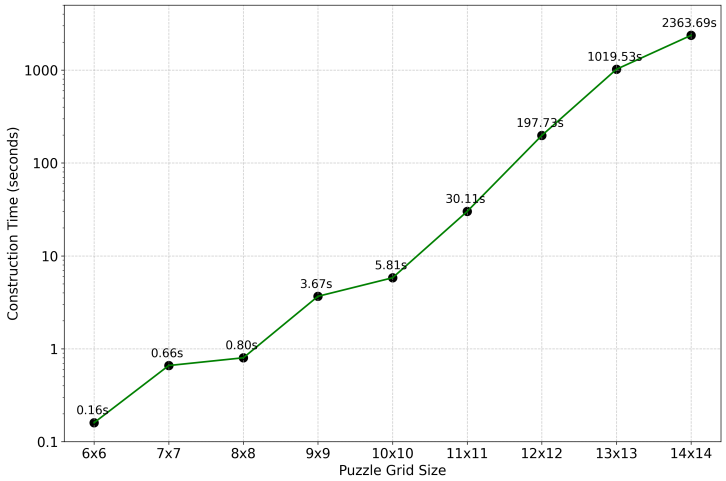


Figure 9: Relationship between Akari puzzle grid size and construction time, illustrating the exponential increase in computational complexity as grid size increases

While working with $n \times n$ grids, the largest n for which the construction algorithm has been tried is 14. But as visible in Table 1, the average construction time for this size was 39 minutes 23.69 seconds, which is not feasible while generating a set of different puzzle configurations. From this table, it can be concluded that for faster puzzle generation, $n = 12$ is still feasible with the constructor, taking an average of 3 minutes 17.73 seconds to construct a puzzle. In the case of larger grid generation, significantly larger generation time needs to be considered.

9.3 Puzzle Solution and Solver Performance Limitations

The second important objective of this thesis is to ensure that the constructed puzzles are solvable using the human strategies applied. Using the developed ASP human strategy-based Akari solver with six human strategies, an analysis across the nine different grid sizes has been done. As the constructed puzzles have unique solutions, the sequence of the logical deduction does not matter much. Having the Akari rules along with it, the solver is employed to get a solution for the constructed puzzles by deducing light placements logically, no matter what the sequence is. It is to be noted that while deducing light placements, the strategies are used following necessity and requirement. If not needed in a puzzle configuration, a strategy might not be implemented there.

Similar to the constructor performance, the solver performance is also recorded for the possible solutions out of the nine different grid sizes. Table 2 presents the number of solvable puzzles for each grid

Puzzle Grid Size	Number of Solvable Puzzle	Average Solving Time
6×6	99	0.03 sec
7×7	100	0.04 sec
8×8	86	0.05 sec
9×9	87	0.06 sec
10×10	89	0.08 sec
11×11	72	0.11 sec
12×12	64	0.13 sec
13×13	47	0.17 sec
14×14	64	0.23 sec

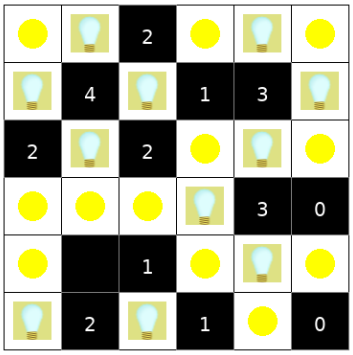
Table 2: A table with the number of solvable puzzles and average solving times for nine different grid sizes by the ASP human strategy-based solver

size, along with the corresponding average solving times using the ASP human strategy-based solver. The results show that all puzzles were solvable for the 7×7 grid, with a total of 100 solvable instances. Similarly, the 6×6 grid had a high number of solvable puzzles, with 99 out of 100 solved. For larger grids, the number of solvable puzzles generally decreases. For example, 8×8 and 9×9 grids had 86 and 87 solvable puzzles, respectively, while the 10×10 grid showed a slight increase to 89. As the grid size increases further, the number of solvable puzzles declines more noticeably. The 11×11 grid had 72 solvable puzzles, and the 12×12 grid had 64. The lowest number was observed for the 13×13 grid, with only 47 solvable puzzles. Interestingly, the 14×14 grid saw an increase to 64 solvable puzzles. The data indicate that the solver performs best on smaller grids, with the number of solvable puzzles tending to decrease as the grid size increases, though some fluctuations are observed for larger grids. Overall, larger grids present greater challenges for the current solving strategy. The required solution time also shows an increasing trend correlating with larger grid dimensions. The solver takes average solving times of 0.03 seconds for 6×6 puzzles and 0.04 seconds for 7×7 puzzles. As the grid size increases, the 8×8 and 9×9 grids require 0.05 and 0.06 seconds on average, respectively, while the 10×10 grid takes 0.08 seconds. This trend continues with larger grids: 11×11 puzzles are solved in 0.11 seconds on average, 12×12 in 0.13 seconds, and 13×13 in 0.17 seconds. The longest average solving time is observed for the 14×14 grid, at 0.23 seconds. These results demonstrate that, although the solver remains relatively fast even for larger grids, the computational time required increases steadily with grid size. This increase in solving time is expected, as larger grids present greater complexity and require more computational resources to solve in the expanded search space.

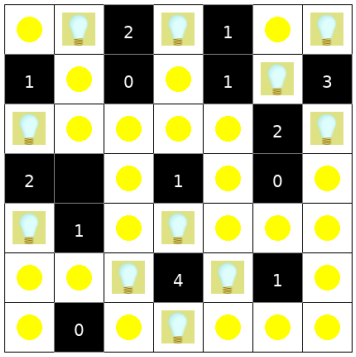
Figure 10 illustrates the solution states of the example grids of Figure 8 for grid size 6×6, 7×7, 8×8, 9×9 and 10×10 using the ASP human strategy-based solver. Figures 8f, 8g, 8h, and 8i are examples of some constructed puzzles which could not be solved by the human strategy-based solver.

The solver's performance limitations are closely linked to its reliance on six specific human solving strategies. While the solver is effective on puzzles where these strategies are sufficient, but may fail on more complex instances that require additional or alternative strategies. This limitation is expected, given that the independent puzzle constructor did not design puzzles with these strategies in mind, and no instructions were provided to align with these six strategies only. Therefore, the solver's inability to solve certain puzzles highlights the solver's inability to adapt when faced with puzzles outside the scope of its implemented strategies. This shows the importance of alignment between puzzle design and the capabilities of strategy-based solvers.

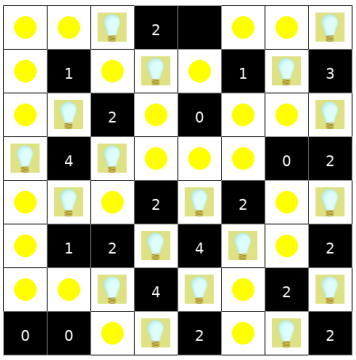
The solving time graph illustrated in Figure 11 shows a gradual increase in solving time as the grid size expands. This increase is consistent with the expected behavior of solving algorithms for NP-complete



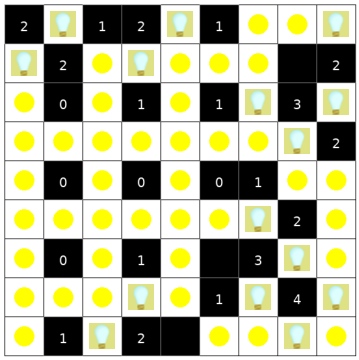
(a) 6×6 solved example grid



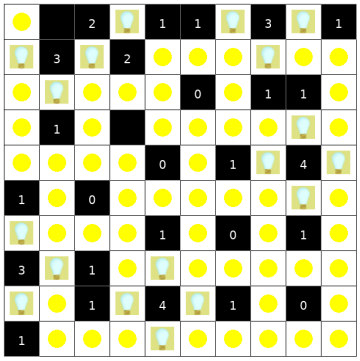
(b) 7×7 solved example grid



(c) 8×8 solved example grid



(d) 9×9 solved example grid



(e) 10×10 solved example grid

Figure 10: Solved grids of unsolved examples of Figure 8 for grid size 6×6, 7×7, 8×8, 9×9, 10×10 using ASP human strategy-based solver

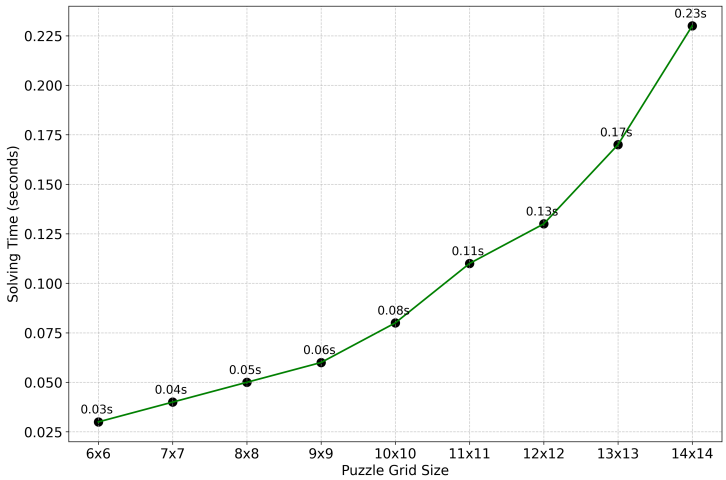


Figure 11: Relationship between Akari puzzle grid size and average solving time by ASP human strategy-based solver, demonstrating a gradual increase in solving time with larger grid sizes.

problems like Akari, as larger instances generally require more computational resources to solve efficiently. The graph shows that the solving times remain relatively low across all grid sizes. While the solver remains efficient for all tested grid sizes, the average computational time grows steadily with puzzle complexity. The nearly linear increase in solving time across the range of grid sizes highlights the scalability of the solver and its suitability for practical use, even when puzzle dimensions increase.

9.4 Human Solver-Embedded Puzzle Construction Redesign

The analysis of solver limitations revealed an insight that certain puzzles generated by the initial constructor were unsolvable using the implemented six human strategies. This disconnect between puzzle generation and human-like solving capabilities limited the practical utility of the system. To address this issue, instead of generating puzzles that may or may not be solvable using human strategies, the constructor is redesigned to integrate human solving strategies directly into the construction process.

The Human Solver-Embedded Puzzle Construction Mechanism ensures that all generated puzzles are not only valid and unique but also solvable using the six identified human strategies. The redesigned process of generating Akari puzzle grids involves the following steps:

- Generate a valid candidate puzzle with nicer configuration through the ASP constructor
- Verify both uniqueness and human solvability in parallel
- Accept only candidates with unique solutions that are solvable using the six human strategies
- Create visual representations of both the solved and unsolved puzzles
- Collect and analyze construction performance metrics

The first step remains the same as the previous generator. It begins with the initialization of an 'AkariPuzzleGenerator' class, which sets the size of the puzzle grid. The Clingo logic programming system then creates a random puzzle configuration using a random seed to ensure diversity in puzzle generation. This phase produces a candidate Akari puzzle grid that serves as the starting point for verification.

The key enhancement in this redesigned approach is the parallel verification of both uniqueness and human solvability. When a candidate puzzle is generated, it undergoes two simultaneous verification processes:

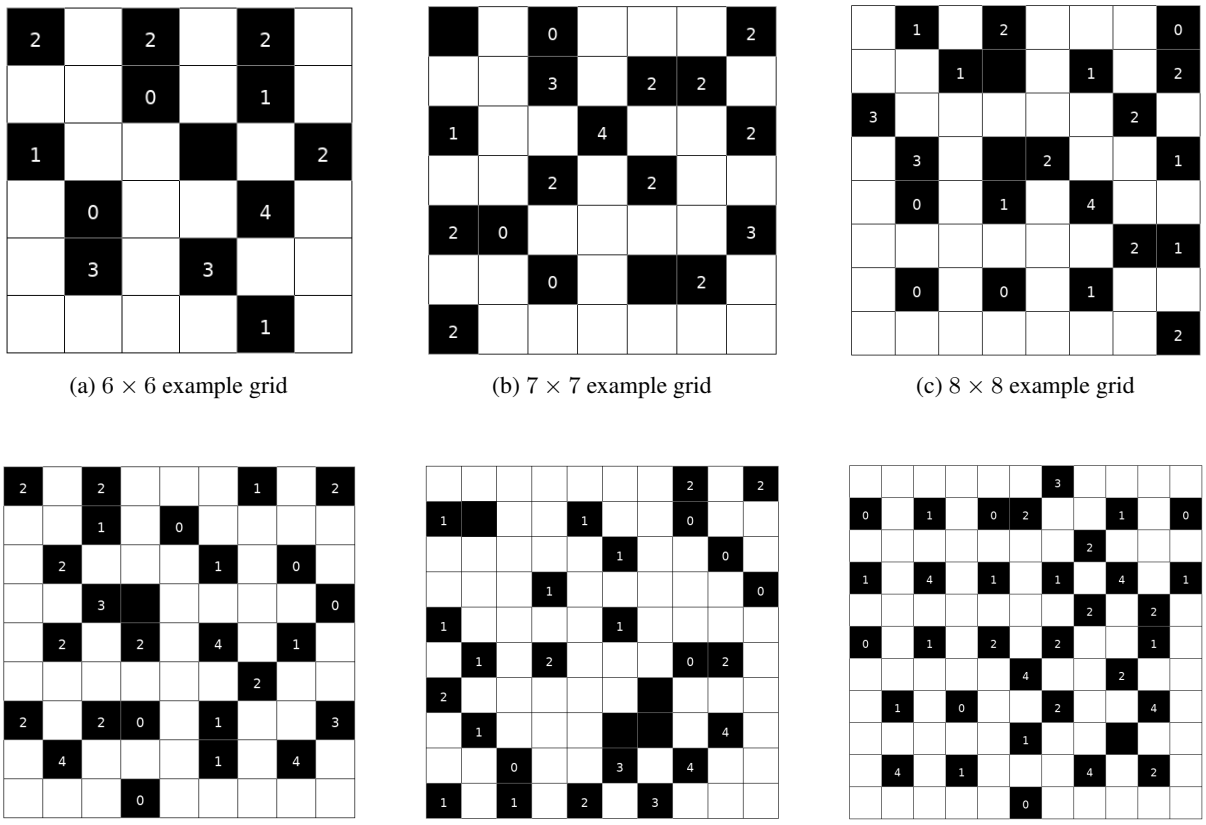
- The uniqueness verification through model counting
- The solvability verification using the human strategy-based ASP solver

For uniqueness verification, the candidate puzzle(in unsolved state) is taken as input into the independent human strategy-based ASP solver mentioned in Section 8.1, and the number of solution models is counted. The human strategy-based solver attempts to solve the puzzle using only the six implemented human strategies. A candidate puzzle is accepted only if it satisfies both criteria: having exactly one solution in the model counting step and being completely solvable by the human strategy-based solver. If either condition fails, the puzzle is discarded, and the process returns to the ASP constructor to generate another candidate. This cycle continues until a puzzle meeting both requirements is found. Once a suitable puzzle is identified, the PIL [9] is used to create visual representations of both the solved and unsolved versions. The images display the grid layout with black cells as walls, and in the solution version, illuminated cells are highlighted with bulb images placed at the positions of lights. By integrating the human strategy-based solver directly into the construction process, this redesigned mechanism eliminates the need for formatting puzzles as ASP facts for further usage, as the puzzles are already verified to be solvable using only the six human strategies. This streamlines the overall process while ensuring that all generated puzzles align with human-solving capabilities. The performance metrics continue to be

collected during this process, measuring the time required for puzzle generation, taking the verification time for both uniqueness and human solvability into account. This data provides valuable insights into the efficiency of the integrated approach across different puzzle sizes.

9.5 Refined Constructor Perfmance

The redesigned construction mechanism represents a significant advancement over the previous design. It ensures that all generated puzzles are guaranteed to be both uniquely solvable and solvable with the human solver using standard logical deduction. But this imposes fundamental constraints on practical grid size limits. Figure 12 shows examples of the reconstructed valid, unique, as well as human-solvable puzzles for grid sizes: 6×6, 7×7, 8×8, 9×9, 10×10, and 11×11. Figure 13 illustrates the solution states of the example grids of Figure 12. Each of these puzzles has a unique solution and can also be solved by using the six human solving strategies only.



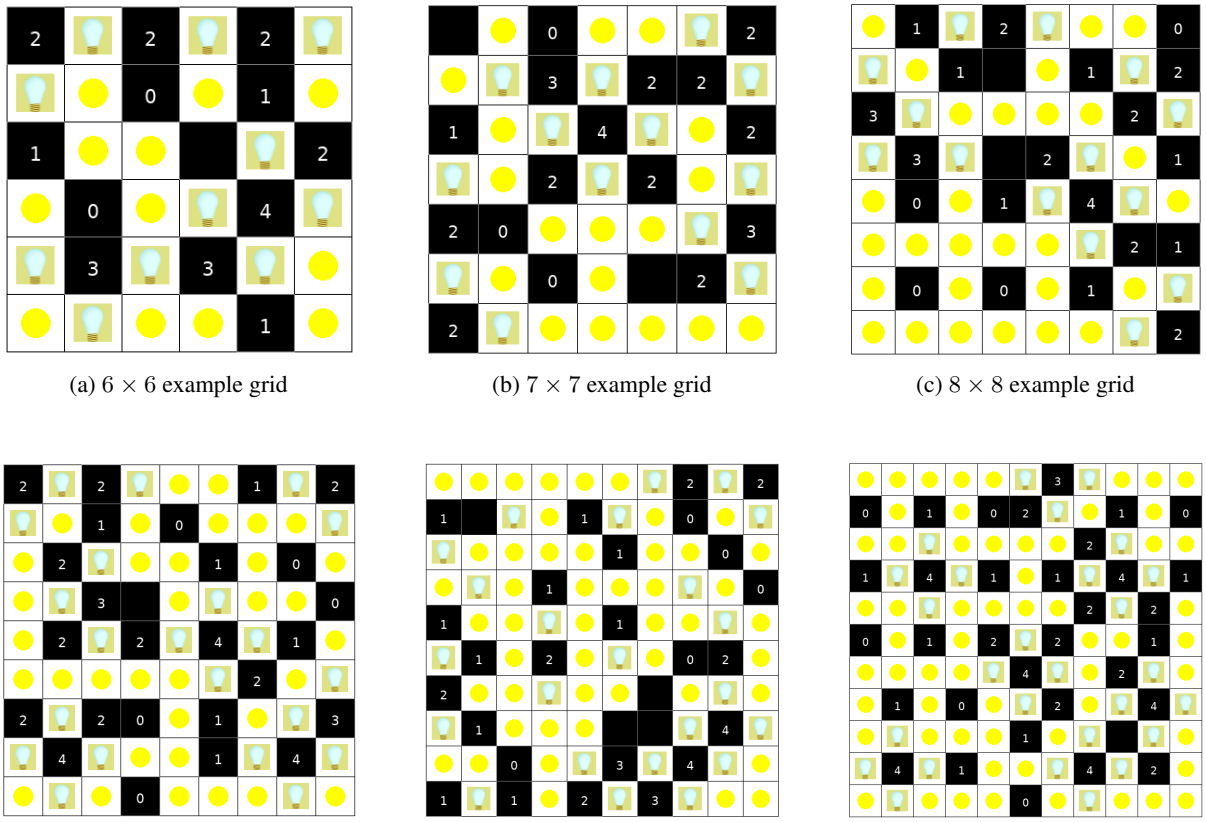
(a) 6×6 example grid (b) 7×7 example grid (c) 8×8 example grid

(d) 9×9 example grid (e) 10×10 example grid (f) 11×11 example grid

Figure 12: Example grids of unsolved puzzles constructed by the refined human solver-embedded puzzle constructor for each grid size 6×6, 7×7, 8×8, 9×9, 10×10, 11×11

Table 3 presents the average construction times for puzzles generated using the redesigned Human Solver-Embedded Puzzle Construction Mechanism. The previously designed constructor could generate puzzles up to 14×14 grids (with a 39-minute average construction time). But the redesigned system stops at 11×11 due to a 47-minute average construction time. This reflects the added complexity of ensuring that solutions align with human strategies.

Figure 14 illustrates the comparison of the performance metrics between the refined constructor and the previous constructor (Table 1). This reveals a significant computational cost increase across grid



(a) 6×6 example grid (b) 7×7 example grid (c) 8×8 example grid

(d) 9×9 example grid (e) 10×10 example grid (f) 11×11 example grid

Figure 13: Solved grids of unsolved examples of Figure 12 constructed by the refined human solver-embedded puzzle constructor for each grid size 6×6, 7×7, 8×8, 9×9, 10×10, 11×11

sizes due to the added human solvability verification step. This increase is especially significant for larger grids, where the time required becomes several minutes or even close to an hour. For instance, the construction time for the 10×10 grid rises from 5.81 seconds (original) to 6 minutes 39.19 seconds, grows by nearly 80 times. For the 11×11 grid, the increase was nearly 100 times compared to the original constructor. This alignment between puzzle generation and human solving capabilities creates a more cohesive and engaging puzzle-solving experience.

Puzzle Grid Size	Average Construction Time
6×6	0 min 1.54 sec
7×7	0 min 4.10 sec
8×8	0 min 7.67 sec
9×9	0 min 28.94 sec
10×10	6 min 39.19 sec
11×11	47 min 3.89 sec

Table 3: A table with average construction times for five different grid sizes with the redesigned human solver-embedded puzzle constructor

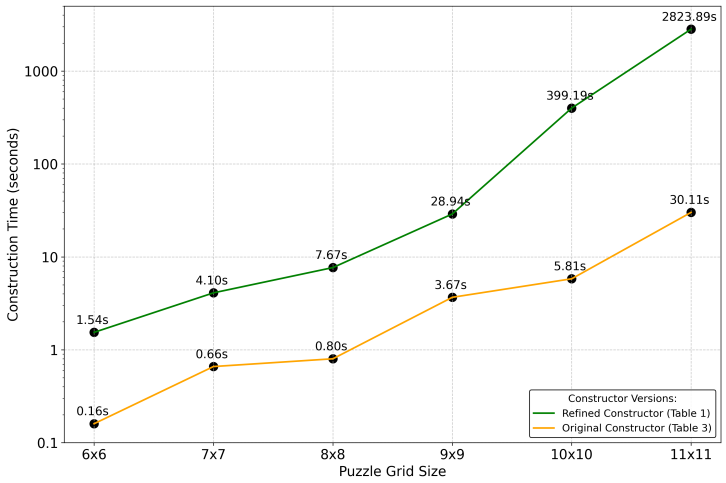


Figure 14: Comparing the Relationships between Akari puzzle grid size and construction time from the refined human solver-embedded puzzle construction, and the previously designed constructor illustrating the exponential increase in computational complexity as grid size increases

With the refined human solver-embedded puzzle constructor, the feasible grid size for practical puzzle generation is reduced due to the increased computational cost. As shown in Table 3, the average construction time rises sharply with grid size. For 10×10 grids, the average time required is 6 minutes 39.19 seconds, which is already substantial for generating multiple puzzles. For 11×11 grids, the average construction time increases to 47 minutes 3.89 seconds, making it impractical for regular use. Based on these results, it can be concluded that $n = 10$ is the largest grid size that remains reasonably feasible for puzzle construction with the refined approach. Attempting to generate larger grids would require a disproportionately long time. This limits the practicality of the constructor for larger grid sizes.

9.6 Case Study: 10x10 Puzzle Solution

This section represents a detailed case study of solving the 10x10 Akari puzzle from Figure 12e. The steps demonstrate how the solver applies various human solution strategies to progress from the initial unsolved state to the complete solution. This also provides insight into the logical progression of solving

an Akari puzzle using formalized human strategies.

In the first step shown in Figure 15a, the forced empty strategy is being applied for the cells with clue 0. With this strategy applied, cells (2,9), (3,8), (4, 9), (7, 2), (7, 6), (8, 3), (8, 5), (8, 7), (9, 2), (9, 4), (10, 3), and (10, 5) are marked as no possible bulb placements. The cells that can't have any light placements are marked with a cross sign. Then, in the second step in Figure 15b, the diagonal exclusion strategy is applied. According to that, cells (5, 8), (9, 10), (10, 7), and (10, 9) are also marked with the cross sign as no possible bulb placements in these cells. In the third step shown in Figure 15c, the restricted clue strategy is applied. According to that, blubs are placed in cells (5, 9), (6, 10), (7, 1), (7, 9), (8, 8), (8, 10), (9, 1), (9, 7), (9, 9), (10, 2) and (10, 8). After these bulb placements, the 2 clue of the cell(9, 6) becomes restricted, and so a bulb is placed in the remaining adjacent cell(9, 5). As clue 2 in cell(5, 10) already has 2 bulbs in adjacent cells, cell(4, 10) is marked as no possible light placement following the clue saturation-based exclusion. To become satisfied, the 1 clue in cell(3, 10) becomes restricted to only possible bulb placement in the adjacent cell(2, 10). So a bulb is placed on this cell. Because of this bulb placement, the 1 clue in cell(1, 10) becomes satisfied, and the other cell(1, 9) is marked with the cross sign as no bulb placement is possible there following the clue saturation-based exclusion.

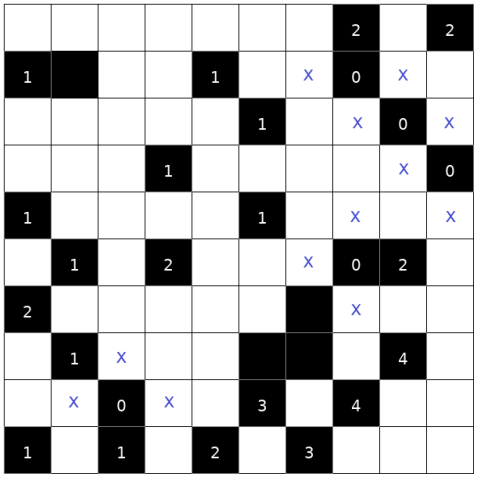
In the remaining unlit part of the grid, following the limited light options strategy, a bulb is placed in cell(1, 6) in Figure 15d. This bulb placement satisfies 1 clues in cells (1, 5) and (2, 6). So no bulb can be placed in the other available adjacent cells (1, 4), (2, 5), (2, 7), and (3, 6) following the clue saturation-based exclusion strategy. This makes the 2 clue in cell(1, 7) restricted to only possible bulb placement in the adjacent cell(1, 8) to satisfy this clue. Similarly, clue 2 in cell(4, 6) also became restricted with only two possible adjacent available cells (4, 5) and (4, 7) to be satisfied. So bulbs are placed in these cells to satisfy the 2 clue. These placements make the 1 clue in cell(4, 4) satisfied, and so the other available adjacent cells (3, 4) and (4, 3) can't have bulb placements using the clue saturation-based exclusion strategy. So these are marked with a cross sign. Now, with the limited light options strategy, a bulb is placed in cell(6, 2). This satisfies 1 clues in cells (5, 2), and (6, 3). So the available adjacent white cells (4, 2) and (6, 4) can't have any bulb placements using the clue saturation-based exclusion strategy. In this scenario, the 1 clue in cell(6, 5) becomes restricted with one possible adjacent white cell to satisfy it. So a bulb is placed in cell(6, 6). A bulb is placed in cell(8, 4) due to the limited light options strategy. Clue 1 in cell(1, 2) also has only cell(1, 3) as the available empty adjacent cell. To satisfy this 1 clue, a bulb is placed in cell(1, 3). After step 4, the only unlit cells in the puzzle are cells (2, 4) and (3, 2). Finally, two bulbs are placed in these two cells following the limited light options strategy in step 5 (Figure 15e). This makes the puzzle grid completely illuminated, satisfying all the rules of Akari. Thus, the solved state is reached, which is exactly the same as Figure 10e.

This step-by-step solution of the 10x10 Akari puzzle demonstrates the effectiveness of the ASP human strategy-based solver. Throughout the solving process, the applications of various strategies are observed. Each step brought the puzzle closer to its unique solution, mimicking the logical progression a human solver might follow. This case study validates the solver's ability to handle complex puzzles by applying necessary human strategies.

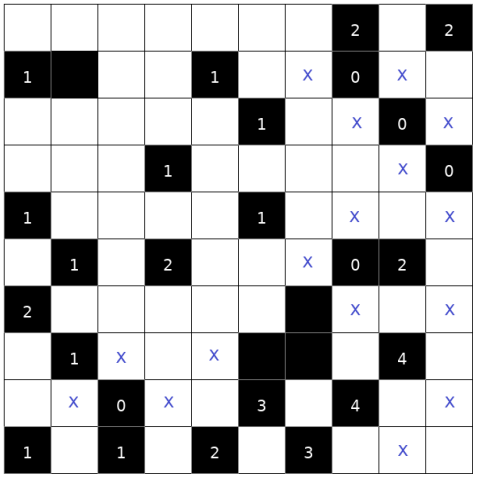
9.7 Comparative Analysis with an Existing Akari Puzzle Generator

An established open-source portable puzzle collection by Simon Tatham et al. [11] includes a C++ implementation of the Akari/Light Up puzzle constructor. The Akari puzzle constructor developed in this thesis is fundamentally different from this implementation. Although both systems aim to generate unique and human-solvable Akari puzzles, the puzzle construction methodologies, solver integration, and verification approaches differ significantly.

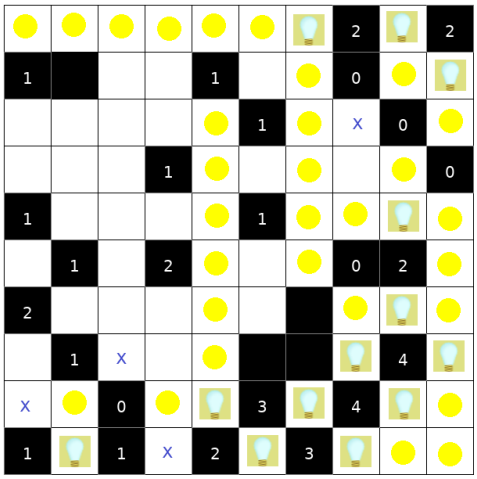
The ASP-based approach presented here enables a declarative formulation of both the puzzle construc-



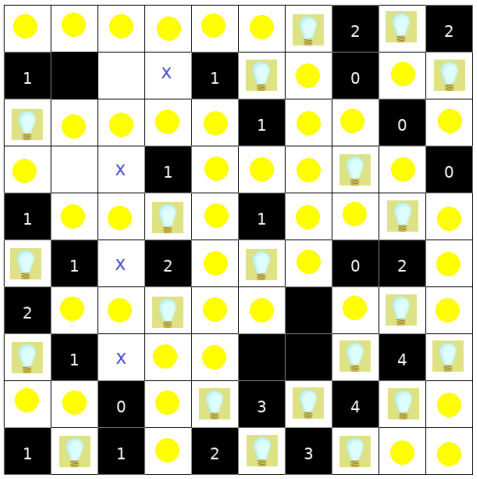
(a) Step 1



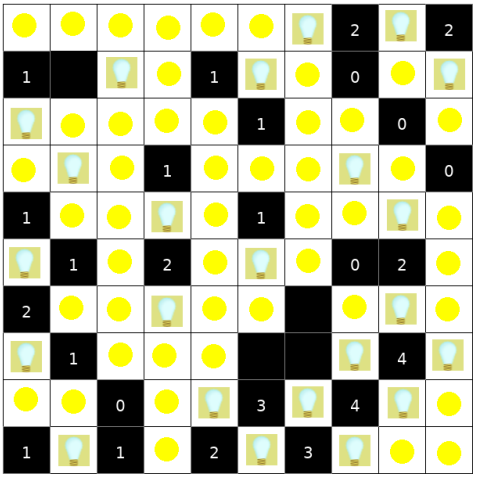
(b) Step 2



(c) Step 3



(d) Step 4



(e) Step 5

Figure 15: Step by step solution of the 10×10 unsolved grid example of Figure 12e using human solving strategies

tion process and the solving process. The construction workflow steps include: the placement of black cells on the grid according to specified constraints, followed by the assignment of clues, and finally the placement of light bulbs. In contrast, the Akari generator in Simon Tatham's Portable Puzzle Collection employs an imperative, heuristic-driven approach implemented in C++. The construction process starts by randomly placing black squares with symmetry constraints, then fully populating the grid with lights. Redundant lights are then iteratively removed while maintaining full illumination and avoiding conflicts. Clues are derived later from the final bulb placement.

The developed ASP-based method verifies the uniqueness of the solutions using a model-counting approach with Python integration (ensuring that only puzzles with exactly one solution are accepted). Here, human solvability is embedded into the construction process by integrating a solver that implements six formalized human strategies, guaranteeing that each generated puzzle can be solved logically without resorting to guessing. This dual verification – uniqueness and human-solvability – ensures that the resulting puzzles are both challenging and accessible to human solvers. Whereas in their approach, uniqueness is enforced through a backtracking solver that counts solutions, discarding puzzles with multiple solutions. Their solver employs a mix of deterministic deduction and limited recursion, allowing for controlled guessing in harder puzzles. The difficulty is adjusted by limiting or permitting recursion during the solving phase. Their implementation also includes optimizations such as randomized removal order and heuristic prioritization, which enable efficient puzzle generation for standard grid sizes.

The primary distinction between the two approaches lies in their construction philosophy and verification rigor. The ASP-based method offers formal guarantees of uniqueness and human-solvability through explicit constraint modeling, but at the cost of computational efficiency. The exponential growth in construction time with increasing grid size is a general characteristic of Akari puzzle generation. It becomes more apparent in the ASP-based implementation because using a general-purpose logic programming solver like Clingo will not be as efficient as specialised C++ code. The model-counting-based uniqueness checking approach might also play a role in this. The ASP-based algorithm gives practical feasibility up to 12×12 grids for the original constructor and up to 10×10 grids for the human-solver-embedded version. The C++ implementation is significantly faster in puzzle generation speed and scales better than this in practice, using heuristics and limited backtracking to balance uniqueness and difficulty without formalizing human-solving strategies to the same extent. But this efficiency comes at the expense of readability and flexibility. Unlike the declarative ASP-based approach, the C++ code is harder to interpret and extend. For example, incorporating new human-solving strategies in ASP can be achieved by adding a few new rules and/or constraints, whereas doing so in the C++ code would require considerable effort and a deep understanding of its procedural logic.

In summary, while both generators produce unique and solvable Akari puzzles, the ASP-based approach prioritizes formal verification and strict adherence to implemented human-solving strategies, whereas the other implementation emphasizes efficiency and adaptive difficulty through heuristic and recursive techniques. The choice between these methodologies depends on whether the primary goal is formal correctness or practical generation performance.

10 Conclusions

In this thesis, I have presented a comprehensive study on the construction of unique Akari puzzles, ensuring that they are human-solvable using the human strategies implemented in this study. The whole process was executed using ASP with Python integration. Initially, I focused on developing an efficient puzzle constructor and a human strategy-based solver, leading to a redesigned constructor that ensures both uniqueness and human solvability.

We designed the puzzle constructor with Akari game rules as a problem validator and a solution validator with careful consideration of constraints implemented in ASP. The construction process verified the uniqueness of the generated puzzles by implementing the model-counting-based verification process. The constructor successfully generated 900 unique puzzles (100 of each grid size) across nine different grid sizes - 6×6, 7×7, 8×8, 9×9, 10×10, 11×11, 12×12, 13×13, and 14×14. The largest n for which the construction is feasible is identified as 12, with 3 minutes 17.78 seconds average construction time. Then I analyzed the constructor's performance, revealing an exponential increase in construction time as the grid size grew. This exponential growth aligns with the NP-completeness of Akari puzzle generation and highlights the computational challenges involved in creating larger puzzles.

Another significant part of this work is the formalization and implementation of human-solving strategies in ASP. I identified six strategies and translated them into ASP rules: Forced Empty Cell, Diagonal Exclusion, Clue saturation-based exclusion, Restricted Clues, Clue Patterns, and Limited Light Options. These strategies form the core of the human strategy-based solver developed in this research. The originally designed constructor produced many unsolvable puzzles for the human strategy-based solver. Although efficient for smaller grids, the solver performance declined progressively for larger grids, highlighting the inherent limitations in the strategy-based approach as puzzle complexity increases. The redesigned Human Solver-Embedded Puzzle Construction Mechanism addressed this by integrating dual verification:

- Uniqueness verification via model-counting
- Solvability verification via the human strategy-based solver.

As required, I achieved 100% solvability of the constructed puzzles through logical deductions without resorting to guessing, mimicking only the formalized human-solving techniques. But this introduced computational overhead. The average construction time increased exponentially with the refined approach. For faster puzzle generation, $n = 10$ is found feasible with the redesigned constructor, taking an average of 6.5 minutes to construct a puzzle, which was 5.81 seconds with the previously designed constructor.

With a case study of a 10×10 puzzle(unique) solution, I provided insights into the solver's step-by-step deduction process, illustrating how the implemented human strategies work in practice. This analysis not only validates the effectiveness of the chosen strategies but also offers a clear demonstration of the solver's logical approach.

A comparative analysis between the developed ASP-based Akari puzzle construction algorithm and solving framework, which I developed in this thesis, with an established implementation (in C++) by Simon Tatham et al. [11], highlighted the fundamental differences in construction philosophy, verification methods, and solver integration, providing context for the strengths and limitations of our proposed approach.

The thesis successfully met its primary objectives with the following step-by-step approach implemented:

- Develop an ASP-based Akari puzzle constructor capable of generating unique puzzles of varying grid sizes

- Create an ASP-based solver that employs six human strategies to solve the generated Akari puzzles
- Analyze the performance of both the constructor and solver across different puzzle sizes
- Refine the constructor design by embedding the human solver on the puzzle construction, which verifies the human solvability of the generated unique puzzles

The use of ASP has proven to be an effective approach in the whole process, offering a powerful framework for encoding complex logical constraints and strategies.

In conclusion, I have demonstrated the efficiency and flexibility of ASP in handling complex puzzle generation and solving problems. Our developed Akari puzzle generator provided a reliable method for creating controlled puzzle instances with guaranteed unique solutions and solvability with the implemented human solving strategies. Our implementation establishes a foundational framework for further research in automated human-solvable nice Akari puzzle creation, with potential applications extending beyond Akari to other logic puzzles and constraint satisfaction problems.

10.1 Future Work

In the current algorithm, while the exponential growth in runtime for larger grid sizes is to be expected given the NP-completeness of Akari, refining and expanding the current construction algorithm could help enable the faster creation of larger, human-solvable puzzles. Further work on this topic could focus on adding ASP optimization techniques and ensuring puzzle uniqueness in ASP itself, instead of the model-counting approach. Enhancing the used ASP construction algorithm might also include developing more sophisticated initial placements, implementing more dynamic constraints, introducing symmetry-breaking constraints, and adding more pattern recognition. Additionally, generating non-rectangular grids can also be explored with the constructor. Another form of Akari, called colored Akari (Akari RGB – where the lights are red, green, and blue), can also be explored with the required ASP rule implementations.

For the embedded human solver, the potential direction of future improvement is to employ more human-solving strategies implemented in ASP, potentially incorporating more advanced solving techniques to improve scalability. This could potentially handle larger grid sizes more efficiently. Another area of extension could be the development of a hybrid solver that combines the current logical deduction approach with machine learning techniques. Such a system could learn to recognize patterns or predict promising solving paths, guiding the ASP solver toward faster solutions. Additionally, future work could explore the adaptation of the solver for puzzles with multiple solutions, requiring more sophisticated constraint handling within the ASP framework.

References

[1] Shih-Yuan Chiu, Shi-Jim Yen, Cheng-Wei Chou, and Tai-Ning Yang. A simple and rapid lights-up solver. In *2010 International Conference on Technologies and Applications of Artificial Intelligence*, pages 440–443, 2010.

[2] Sancho Salcedo-Sanz, Leopoldo Carro-Calvo, Emilio G. Ortíz-García, Ángel M. Pérez-Bellido, and José Antonio Portilla-Figueras. A nested two-steps evolutionary algorithm for the light-up puzzle. *J. Int. Comput. Games Assoc.*, 32:131–139, 2009.

[3] Igor Rosberg, Elizabeth Goldbarg, and Marco Goldbarg. Solving the light up with ant colony optimization. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 566–573, 2011.

[4] Bram Pulles. Analysis of akari – BSc. thesis, 2023.

[5] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[6] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, December 2011.

[7] Akari rules. https://en.wikipedia.org/wiki/Light_Up_(puzzle).

[8] Clingo. Python api documentation. https://potassco.org/clingo/python-api/5.4/.

[9] Pillow. Python imaging library documentation. https://pillow.readthedocs.io/en/stable/.

[10] Human strategies to solve akari. https://www.keepitsimplepuzzles.com/how-to-solve-an-akari-puzzle/.

[11] Simon Tatham and contributors. Simon Tatham's portable puzzle collection. https://www.chiark.greenend.org.uk/~sgtatham/puzzles/, 2004-2021. MIT License. Contributors include Richard Boulton, James Harvey, Mike Pinna, Jonas Kölker, Dariusz Olszewski, Michael Schierl, Lambros Lambrou, Bernd Schmidt, Steffen Bauer, Lennard Sprong, Rogier Goossens, Michael Quevillon, Asher Gordon, and Didi Kohen.

[12] QuillBot. Paraphraser. https://quillbot.com/paraphrasing-tool, 2025.

[13] Perplexity Pro. Ai-powered search and writing assistant. https://www.perplexity.ai, 2025.

A Python Clingo Integrations

A.1 Akari Puzzle Generator Implementation with Unique Solutions

A.1.1 Overview

This details the implementation of the Akari puzzle generator developed for this thesis. The generator produces controlled puzzle instances using ASP through the Clingo solver while ensuring each puzzle has exactly one solution discussed in Section 5. The implementation balances computational efficiency with quality assurance, using randomization techniques to ensure diversity in the generated puzzles while maintaining the constraint of solution uniqueness.

A.1.2 Pseudocode

While developing the constructor, the required dependencies were Clingo, PIL, and standard Python libraries: os, time, and random. The puzzle generation process uses two primary ASP encodings: "encode_puzzle.lp" and "encode_AkariRules.lp". The first ASP encoding contains rules and constraints for generating valid candidate Akari puzzle configurations, and the second encoding contains rules defining valid solutions to the candidate Akari puzzles. The following pseudocodes outline the full structure of the Akari puzzle generator:

Listing 12: Akari Puzzle Generator Class

```
Class AkariPuzzleGenerator:

  def initialize(size):
      // Constructor setting the puzzle grid size
      Set grid size to input size

  def generate_puzzle_configuration():
      // Generate a candidate puzzle configuration
      Create a random seed for puzzle diversity
      Initialize Clingo solver with a random seed
      Load puzzle encoding rules from external file
      Suppress output and solve for one model

      if no model is found:
          Return None

      Extract black cells without numbers
      Extract black cells with numbers and their constraints

      Return both sets of black cells

  def check_unique_solution(black_cells, black_cells_numbered):
      // Verify that the puzzle has exactly one solution
      Initialize Clingo solver to find up to 2 solutions

      Add facts for all black cells (with and without numbers)
      Load Akari rules from external file
      Suppress output and solve

      if more than 1 solution found:
          Return False, None, None
      elif exactly 1 solution found:
          Extract light positions from the solution
          Extract illuminated cells from the solution
          Return True, lights, illuminated
      else:
```

```
            Return False, None, None

    def generate_unique_puzzle():
        // Generate a puzzle with exactly one solution
        Loop until success:
            Generate puzzle configuration
            if configuration is None:
                Continue to the next iteration

            Check if the puzzle has a unique solution
            if unique:
                Return black_cells_numbered, black_cells, lights, illuminated
            else:
                Continue to the next iteration

    def create_puzzle_image(black_cells, black_cells_unnumbered, lights,
                                illuminated, filename, show_solution=True):
        // Create a visual representation of the puzzle
        Initialize image parameters (cell size, margins)
        Create a new blank image
        Initialize drawing context

        Define helper Function to get the appropriately sized font
        Load bulb image for light representation

        Draw grid lines

        if show_solution is True:
            Draw illuminated cells with yellow background
            Draw bulbs on illuminated cells

        Draw unnumbered black cells
        Draw numbered black cells with their constraint numbers

        Save the image to the specified filename

    def format_puzzle_asp(black_cells, black_cells_unnumbered, light):
        // Format puzzle as ASP facts for solving
        Create an empty list for ASP output

        For each unnumbered black cell:
            Add fact: black_cell_unnumbered(x+1, y+1)

        For each numbered black cell:
            Add fact: black_cell(x+1, y+1, number)

        For each light position:
            Add fact: light(x+1, y+1)

        Join all facts with newlines and return as a string
```

Listing 13: Main Function of the Constructor

```
def main():
  // Main Function to orchestrate puzzle generation
  Create output directories for different puzzle representations

  Initialize AkariPuzzleGenerator with the default size
  Set total puzzles to generate = 30
  Initialize an empty list for storing unique puzzles
  Initialize the list for timing measurements

  While unique puzzles count < total puzzles:
      Start timer
      Generate a unique puzzle
      End timer and calculate elapsed time
      Store elapsed time

      Extract puzzle components

      Create and save the solved puzzle image
      Create and save unsolved puzzle image

      Format the puzzle as ASP facts and save it to a file

      Add puzzle to the unique puzzles list

      Write timing information to file

  Calculate and write the average generation time to file
  Print completion message
```

A.2 Human Strategy-based Akari Solver

A.2.1 Overview

This section details the implementation of a human strategy-based Akari puzzle solver developed to ensure the solvability of the puzzles by humans. The solver takes puzzle configurations produced by the generator and computes solutions using a human strategy-based ASP through the Clingo solver, validating the solvability while collecting performance metrics.

A.2.2 Pseudocode

With similar dependencies mentioned in Section A.1.2, this process takes the puzzle configuration files as facts from the generator and integrates with the ASP encoding of the human strategy-based solver discussed in Section 8.1. The following pseudocodes outline the full structure of the Akari human strategy-based solver:

Listing 14: Utility Functions of the Solver

```
    def natural_sort_key(s):
        // Helper function for natural sorting of filenames (e.g., puzzle_1.lp,
    ↪ puzzle_2.lp, etc.)
        Split the string by numbers and non-numbers
        Return a list where numbers are converted to integers for proper sorting

    def solve_puzzle(puzzle_file, solver_file):
        // Core function to solve a puzzle using ASP
        Initialize Clingo solver
        Load puzzle file containing black cell constraints
        Load solver file containing Akari rules
        Ground the logic program
        Solve for one model
        if a solution found:
            Process and return the model
        else:
            Return None

    def process_model(model):
        // Extract puzzle elements from the ASP solution model
        Initialize empty lists for puzzle elements
        Initialize counters

        For each atom in the model:
            if atom is "black_cell":
                Extract coordinates and number
                Add to black_cells list
            elif atom is "black_cell_unnumbered":
                Extract coordinates
                Add to black_cells_unnumbered list
            elif atom is "lightbulb":
                Extract coordinates
                Add to lights list (solver's solution)
            elif atom is "light":
                Extract coordinates
                Add to generator_lights list (original solution)
            elif atom is "lit":
                Extract coordinates
                Add to illuminated list
            elif atom is "light_count":
                Store count of generator lights
            elif atom is "lightbulb_count":
                Store count of solver-placed lights

        Return all extracted elements
    def create_puzzle_image(size, black_cells, black_cells_unnumbered, lights,
    ↪ illuminated, filename):
        // Create a visual representation of a solved puzzle
        Initialize image parameters (cell size, margins)
        Create new blank image
        Initialize drawing context

        Define helper function to get the appropriately sized font
        Load light bulb image

        Draw grid lines

        Draw illuminated cells with yellow color
        Draw light bulbs on appropriate cells

        Draw unnumbered black cells
        Draw numbered black cells with their constraint numbers

        Save the image to the specified filename
```

Listing 15: Main Function of the Solver

```
    def main():
    // Main function to orchestrate puzzle solving
    Set up file paths and directories
    Create output directories if they don't exist
    Set puzzle size parameter
    Initialize timing measurement list

    Get all puzzle files and sort them in natural order
    Initialize counters and tracking lists

    For each puzzle file:
        Start timer
        Solve puzzle
        End timer and calculate elapsed time

        if a solution found:
            Extract puzzle elements from the solution

            Compare the solver's solution with the generator's solution:
                if light counts match and positions are identical:
                    Save the solution image to the output folder
                    Record as a uniquely solved puzzle
                    Log solving time
                else:
                    Save the solution image to the unmatched folder
                    Record as non-uniquely solved
        else:
            Record as an unsolvable puzzle

    Calculate and output statistics:
        Number of uniquely solved puzzles
        List of uniquely solved puzzles
        List of unmatched/unsolved puzzles
        Average solving time
```

Declaration of Authorship – Individual Thesis

I hereby declare that I am the sole author of this master's thesis entitled "Construction of Human-Solvable Akari Puzzles" and have not used any resources or technical tools other than those specified.

In addition, I declare that, in the preparation of this thesis, I have taken support from the Quillbot Paraphraser [12] to assist with paraphrasing and the generative AI tool – Perplexity Pro [13], to assist in language refinement throughout the document. All paraphrased and AI-generated content was critically reviewed, edited, and verified by me to ensure accuracy and adherence to scientific standards.

Furthermore, I declare that this master's thesis or any part of it has not been previously submitted for a degree or any other qualification at the TUD or any other institution in Germany or abroad.

Name: Urmee Pal
Matriculation Nr. 4865875

Urmee Pal Digitally signed by Urmee Pal Date: 2025.07.14 14:10:31 +02'00'

Dresden, 14 July, 2025

Place, Date

Signature

Acknowledgments

I would like to express my sincere gratitude to the members of my thesis committee, Prof. Dr. rer. pol. Markus Krötzsch and Dr. habil. Hannes Strass for their time and valuable contributions to the evaluation of my Master's Thesis. I am deeply thankful to my tutor, Dipl.-Math. Maximilian Marx, for his invaluable guidance, supervision, and continuous support throughout my research and the writing of this thesis. His constructive comments, expertise, and generous knowledge sharing have greatly helped me in shaping the final version of this thesis. I am truly grateful for the opportunity to work under his supervision.

I would also like to extend my heartfelt thanks to my family for their unwavering encouragement, understanding, and support throughout my academic journey. I owe an immense debt of gratitude to my mom Mrs. Shipra Paul, my dad Mr. Samir Paul and my brother Saikat Pal, who supported my decisions and encouraged my educational endeavours from the very beginning. Their belief in me has been a constant source of motivation.

Finally, I would like to thank myself for not giving up, for all the late nights, and for finding the motivation to keep going even when things got tough. The process has taught me a great deal about perseverance and self-belief.