University of Glasgow | School of Computing Science

Towards Human-like Reasoning in Constraint Programming; A Comparative Study of Minizinc Models for Solving the Skyscraper Puzzle

Akash Ananda Kumar Murali

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

30th August 2023

Abstract

This project explores the interaction between human reasoning and computational problem-solving through skyscraper puzzles. This project mainly focuses on the field of constraint programming (CP), which conventionally depends on algorithmic guessing to find solutions, which differs from human logic that relies more on deductive reasoning. The main question guiding this study is: can a CP solver be designed to solve a skyscraper puzzle which resembles human-like reasoning? To investigate, we developed two distinct models using minizinc with various sets of constraints. The first model uses the classic CP approaches, but the second model uses the table constraint for rule-based deductions to reduce guessing. The primary metrics for evaluating both models are node structure, depth of search tree, and computational solve time. The finding shows that the table constraint model aligns more like human reasoning and also performs more efficiently. This project proposes a promising future study focused on augmenting the human-like reasoning abilities of artificial intelligence systems in constraint programming.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Akash Ananda Kumar Murali Signature: Akash Ananda Kumar Murali

Acknowledgements

I would like to thank my supervisor, Dr. Ciaran McCreesh, for his support and guidance throughout the project. Also, I am thankful for a wonderful time living in Glasgow and studying at the University of Glasgow.

Contents

Chapter 1

Introduction

In the field of Artificial Intelligence (AI)[12] and Constraint Programming(CP)[1], one of the most interesting and challenging questions is how computers can think and solve problems like humans. AI is getting used in various applications, from natural language processing to robotics. This is significant not just because of computational power, but also in the adaptive and context sensitive way of humans approaching problems.

Constraint programming is a subset of AI that focuses on solving combinatorial problems by establishing a set of rules or 'constraints' that defines the parameters of the problem. Even though CP is powerful and efficient, its deterministic and formulaic way of solving problems often don't match with ways humans approach problem-solving. This difference is clear when solving puzzles, which frequently requires strategies, intuitions, and insights that are not easily replicated in CP algorithms.

One such example that captures the essence of this difference is the Skyscraper puzzle[6] that requires both logical deduction and spatial reasoning. While CP can be used to solve it effectively, the methods used often involve 'guessing'. This difference in between how humans solve, and machine calculation raises interesting questions about the adaptability and limitations of current CP methods. In this study, our main objective is to implement two different minizinc[4] models using constraint programming which solves skyscraper puzzles and evaluates both models to find which model replicates more like human reasoning.

The structure of this report is as follows:

- Chapter 2: Background information about constraint programming, MiniZinc, and the Skyscraper puzzle, as well as approaches to puzzle-solving by both humans and computers.
- Chapter 3: Implementation of two minizinc models to solve the Skyscraper puzzle using constraint programming. The first model uses basic algorithmic constraints, while the second uses table constraints to mimic human-like reasoning.
- Chapter 4: Evaluation of the two models against a set of metrics including guessing, node structure, and execution time. The aim is to understand which model more closely resembles human-like puzzle-solving.
- Chapter 5: Conclusion of key findings and suggests the future work.

Chapter 2

Background

This chapter gives necessary background information for this study. Solving puzzles shows how good the human brain is at rational thinking and seeing patterns. Over time, computer scientists have developed machines that can do certain tasks, like solving puzzles. Some of the ways this gap is closed are by using Constraint Programming (CP) and tools like Minizinc. In this section, we'll discuss these topics, which will set the stage for how machines and people solve the Skyscrapers puzzle differently.

2.1 Constraint Programming (CP)

Constraint Programming (CP)[1] is a highly effective computational approach for solving combinatorial issues. CP is based on the idea of expressing problems as a set of constraints on variables and then depending on a solver to identify solutions that satisfy these constraints.[1] Because this idea of solving is general, many areas use it: interactive graphic systems, operation research (scheduling), molecular biology (DNA sequencing), business applications (option trading), and electrical engineering (computing optimal circuit layouts, to name a few.[2]

Basics of Constraint Programming

Variables in the CP can represent an undetermined value. Consider the Sudoku puzzle, which many computer scientists are likely to be familiar with. Each empty cell in the Sudoku grid represents a variable, holding a space for a possible number.

Constraints are the conditions that solutions must follow. Rules are like "each row must have numbers from 1 to 9 with no repetition" or "each 3x3 grid must have distinct numbers from 1 to 9" which serve as constraints in Sudoku.[1]

Solver is an algorithmic engine that searches for solutions to problems based on the variables and constraints. In the case of Sudoku, a solver would try to fill the grid in such a way that all of the constraints are met. To solve the problem efficiently, the solver uses methods like backtracking, constraint propagation, and clever heuristics.[2]

Three basic qualities which describes the benefits of Constraint Programming[2,3]:

1. Two-Phase Approach: The Constraint Programming process is two stages: defining the problem as a constraint and solving it.

2. High Flexibility: A constraint model is highly versatile since limits may be readily added, removed, or changed.

3. The presence of built-ins: Constraint solvers provides a variety of built-in-functions and algorithms for solving constraint models.

The relation between puzzles and CP is obvious. Puzzles are a set of limitations that must be met, and CP provides the tools to navigate these limits systematically and efficiently. Using CP provides a structured and clear way of understanding the mechanics of puzzle-solving from a computational aspect for this study.

2.2 MiniZinc

MiniZinc is a high-level and declarative constraint modeling language.[4] It acts as a bridge which allows users to describe a wide variety of combinatorial problems without committing to a specific solver or its low-level details.[5]

Benefits of MiniZinc:

- MiniZinc abstracts away any solver-specific features which allows users to focus on describing their problem in clear and straightforward language. This level of abstraction provides more natural problem representation which is similar to how one will describe the problem verbally or numerically.
- MiniZinc has the unique advantage of remaining independent of any particular solver. After a problem has been defined in minizinc, several solvers can be utilized to discover a solution. This interchangeability gives flexibility and allows users to take advantage of various solvers.[5]
- MiniZinc was developed for constraint programming, its design enables for integration with other solution paradigms such as Integer Linear Programming (ILP) and Boolean Satisfiability.[5]

2.3 Skyscraper puzzle

Skyscraper is a logic-based puzzle inspired by the rich history of Japanese puzzles.[6] The Skyscraper puzzle is similar to other grid-based puzzles such as Sudoku. However, its distinct constraints and clues distinguish it.

Rules and Structure of the Puzzle:

Skyscraper puzzle's gameplay relies around a grid, usually of n × n dimensions. The goal is to fill this grid with numbers representing skyscraper heights, which typically range from 1 to n. The clues given on the grid's edges indicate the number of visible skyscraper buildings from that vantage point, which initiates the problem.[6] The following are the rules:

- Each row and each column must contain all the numbers from 1 to n, with no repetition.

- A number in each puzzle which represents the height of a skyscraper.
- The clues on the edges tell how many skyscrapers are visible when looking inwards from that direction. Higher skyscrapers block the view of lower ones. So, a clue of "1" means only the tallest skyscraper (*n*) is visible, while a clue of "*n*" would mean they descend stepwise from the border *(1, 2, 3, ..., n)*.[6]

Example of the Skyscraper Puzzle:

Consider a simple 4 x 4 Skyscraper puzzle[7] as shown in figure 2.1 with given clues on the edge of the grid to solve the puzzle.

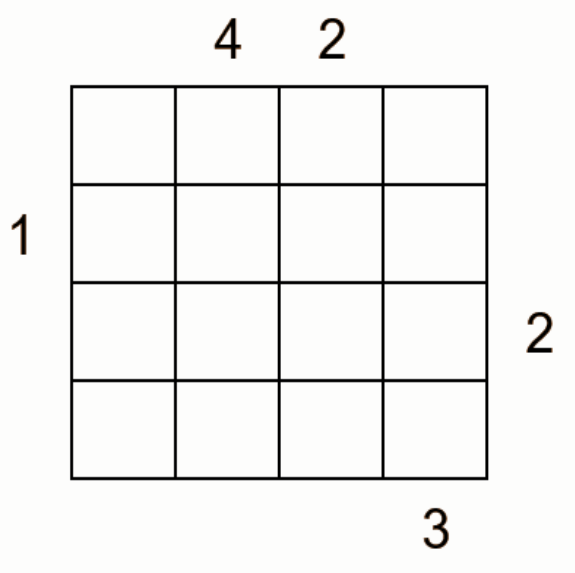

Figure 2.1: Unsolved Skyscraper puzzle

Understanding Visibility Times:

Visibility times refers to the number of skyscrapers visible when viewing across a row or down a column. And taller skyscrapers block the view of shorter ones. Let's use this approach to understand the given clues from the figure 2.1:

Top Clue of "4": The second column from the left has a top clue of "4". This means when looking from the top, all 4 skyscrapers should be visible in increasing order of their heights. This leads to only one possible arrangement: 1,2,3,4.

Left Clue of "1": The second row from the top has a left clue of "1" that means we'd only see the tallest skyscraper of a building of height 4.

Bottom Clue of "3": Looking upwards from the bottom on the fourth column, three skyscrapers are visible.

Based on these basic insights, we can fill the grid[7] to solve the puzzle as shown in figure 2.2

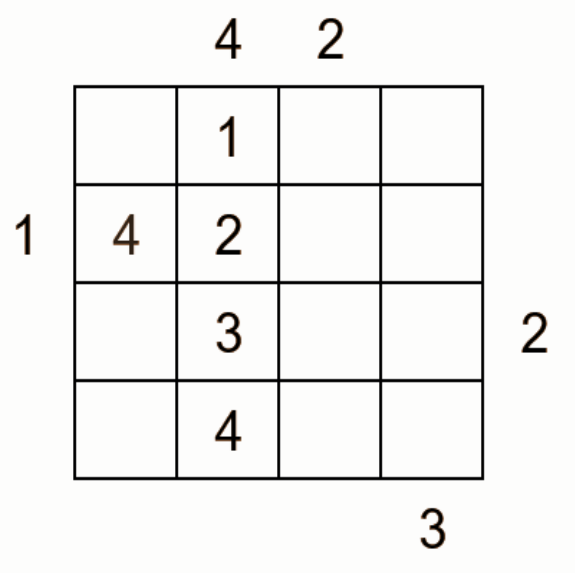

Figure 2.2: solving the puzzle based on visibility times

With the information derived from the visibility clues and the requirement that each row and column must include the numbers 1-4 without any duplicate values, the remaining sections of the puzzle[7] can be solved and the final puzzle looks as shown in figure 2.3.

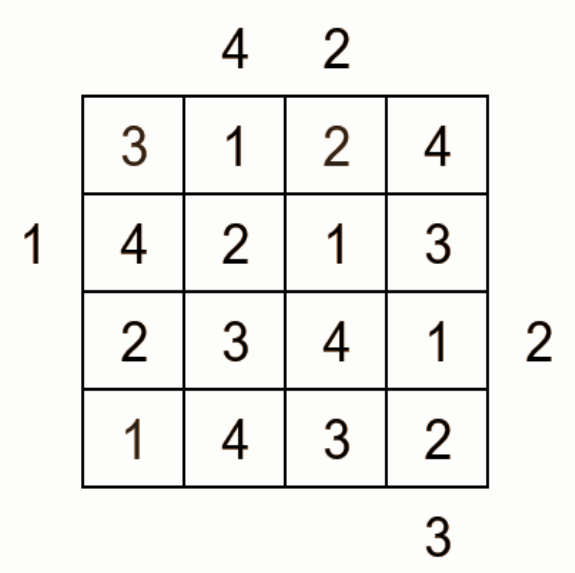

Figure 2.3: Solved Skyscraper puzzle

Why the Skyscraper Puzzle for this study: The Skyscraper puzzle problem gives a unique kind of logic and visuals. Unlike other standard number placement puzzles, Skyscraper puzzle needs solvers to think spatially and also visualize the varying heights of "skyscrapers". This interesting logic made skyscraper puzzles a choice of acting as a benchmark to understand and compare human and computer puzzle-solving strategies.

2.4 Human vs. Computer: The Puzzle-Solving Approach

Lets see how human and computer will approach the puzzle like skyscraper:

Human Approach to Skyscraper Puzzles

Humans approach the Skyscraper puzzle in different ways:[8,9]

Intuition: Mostly based on earlier experience with similar problems, seasoned players frequently develop an intuitive understanding of where numbers might fit. Patterns recognized over time gives this intuition.

Strategy: Through repeated gameplay, players understand and develop the effective tactics or sets of rules. For example, if they see a clue of '4' on a 4x4 Skyscraper puzzle, they may quickly put a '1' next to it because that is the only way to see all four buildings.

Trial and error: Some positions may be guessed, especially in tough puzzles, and then the implications of those guessess are calculated. And If there are contradictions, then the guessess is to be incorrect and then another guess is tried.

Computer Approach to Skyscraper Puzzles

Computers, on the other hand, aren't concerned about the puzzle's context. Their strategy is based on:[8,9]

Algorithms: A well-defined collection of instructions that ensures the existence of a solution. For example, constraint propagation is a common strategy for narrowing down possible options.

Brute Force: Computers test all possible combinations until a solution is found. However, optimized algorithms which will usually eliminate the need for brute force approach.

Optimization: using the advanced algorithms and heuristics, computers can quickly get rid of wrong solutions and efficiently find the true answer.

A comparative perspective:

The way humans and computers approach the problems differently, but the logic is a link between the two. The intuition of an experienced player and the algorithms of a computer are both based on logic. However, where humans make mistakes or take longer due to complexity of the problem, computers are always accurate and quick.

Summary

In this background, we explored the concepts of constraint programming (CP), capabilities of Minizinc, structure of skyscraper puzzle and unique puzzle solving approach of humans and computers. In the next chapter, we will implement the minizinc models which solve the skyscraper puzzle.

Chapter 3

Implementation

In this chapter, we'll dive into the technical aspects of developing two different models which solve the Skyscraper puzzle: the Algorithmic Model and the Table Constraint Model. And in the next evaluation chapter, we will discuss why the second model and comparison analysis.

3.1 Algorithmic Model

The Algorithmic Model solves the skyscraper puzzle in a systematic strategy using MiniZinc. And by using basic concepts of constraint programming, the model ensures accuracy by creating the puzzle grid, its constraints, and applying the given clues.

Initialization and Puzzle Grid Setup:

```
include "globals.mzn";
```

The "*globals.mzn*" file is included here to provide a set of predefined global constraints to help in solving the puzzle.

```
int: N = 4;
```

The size of the puzzle grid is set to 4, indicating a 4x4 grid for the Skyscraper puzzle.[6]

```
array[1..N] of int: top_clue = [0,0,0,0];
array[1..N] of int: bottom_clue = [2,1,0,0];
array[1..N] of int: left_clue = [0,4,2,0];
array[1..N] of int: right_clue = [3,0,0,0];
```

These array representations initialize the clues given for the puzzle. Those clues indicate how many skyscrapers are visible from a specific direction. And the value of 0 indicates that no clue is given for that direction.

```
array[1..N, 1..N] of var 1..N: grid;
```

The puzzle grid is represented as a 2D array of variables, with each cell taking value ranging 1 to N.

Essential Constraints

The core of a constraint programming model is its constraints, which determine the valid arrangements and combinations, and satisfy the problem's rules. In the Skyscraper puzzle, vital constraints ensure the solution's validity and also follows the puzzle rules. These essential constraints are key to our model.

Ensuring Uniqueness in Rows and Columns:

```
constraint forall(i in 1..N) (
  alldifferent(row(grid, i)) /\
  alldifferent(col(grid, i))
);
```

The *alldifferent* function ensures that a collection of values, such as those in a row or column, are distinct from one another. In the Skyscraper puzzle, this function ensures no two skyscrapers of the same height exist in the same row or column. This constraint is required for the puzzle's correctness.

Visibility Constraint:

The fundamentals of the skyscraper puzzle depend heavily on visibility constraints. They specify how many skyscraper buildings are visible from a given edge of the grid. A taller skyscraper hides the shortened skyscraper behind it. This distinction is important because it influences the position of skyscrapers within each row and column.

```
constraint
  forall(i in 1..N)(
    (top_clue[i] = 0 \/ sum(j in 1..N-1)(
      bool2int(grid[j,i] < grid[j+1,i] /\ forall(k in 1..j)(grid[k,i] <= grid[j+1,i]))
    ) = top_clue[i] - 1)
```

Let's analyze this visibility constraint for the top direction and which is same applied for all other direction:

Purpose: This constraint ensures that the number of visible skyscrapers for each column a matches the given clues (or its unconstrained if the clue is 0).

The condition: top_clue[i] = 0 verifies if there is a clue given or not for that column. If there is no clue given, then the visibility constraint does not apply.

Counting Visible Skyscrapers: The summation is the important part of this constraint. The condition (grid[j,i] < grid[j+1,i]) checks for each row j , if the next skyscraper is taller than the current one. The nested *forall* makes sure that all previous skyscrapers

are shorter than or equal to the next skyscraper. If these conditions are satisfied, then the next skyscraper becomes visible.

Bool2int: This function transforms a boolean either true or false result to an integer value 1 or 0. As a result, every time a skyscraper is visible, the sum increases by 1.

Clue comparison: The total visible skyscraper from the sum should be the same as the clue minus 1. The subtraction occurs due to the method of counting the visibility.

Additional Constraint

In the Skyscraper puzzle, while the essential constraints assuring on row and column uniqueness and satisfying the visibility criteria, more constraints that can be added. These additional constraints are not necessary to get the correct solution for the puzzle, but they can help speed up the solving process, improving efficiency, and also might suggest another way of approaching the problem.

```
if top_clue[i] = 1 then
        grid[1,i] = N endif
```

This rule of constraints can help find the solution space faster. The solver can discard many other configurations because it knows that the cell must have the tallest skyscraper.

```
if top_clue[i] = N then
        increasing([grid[j,i] | j in 1..N]) endif
```

When this clue is present, the solver does not need to consider other configurations for that row or column. It can organize the skyscraper in increasing order.

Solving and Output Formatting

solve satisfy;

In MiniZinc, the solve satisfy statement initiates the solver to search for a solution that satisfies all of the given constraints.

```
output ["| " ++ join(", ", [show(grid[i,j]) | j in 1..N]) ++ " |\n" | i in 1..N];
```

The output format is developed with human readability and displaying the solution in a grid structure.

3.2 Table Constraint Model

This table constraint model starts with the basic concepts of the grid, dimensions, and clues. It diverges from the original, though, by including a table-based method.[10] This involves precomputing various puzzle grid permutations and pairing them with their relevant visibility clues.

Visible Clue Function:

The visible clue function is an innovative addition to the second model. Its primary objective is to determine the number of skyscrapers that are visible when looking from a specific direction along a row or column, based on the given heights.

```
function int: visible(array[int] of int: arr) =
 let {
   array[1..N] of int: max_up_to = [max(j in 1..i) (arr[j]) | i in 1..N];
 } in
 sum([if i = 1 \/ arr[i] > max_up_to[i-1] then 1 else 0 endif | i in 1..N]);
```

The Input: The function is given an array, arr, which contains a sequence of skyscrapers' heights.

The Local Variable - max_up_to: The function contains the computed array max_up_to. This array keeps account of the maximum height seen until a given index in the input array. For example, the max_up_to array for the input array [1,3,2,4] would be [1,3,3,3]. Information Technology Visibility:

Visibility in Computing: The function checks whether the height of each skyscraper in the sequence (arr[i]) is greater than the greatest height seen in the preceding sequence (max_up_to[i-1]). It's always visible if it is the first skyscraper (i=1). Otherwise, it becomes visible if its height exceeds the tallest seen thus far. The function counts all visible skyscrapers.

Computation of Permutations and Their Visibility:

When we solve a Skyscraper puzzle for grids like 4x4, we are not just filling in numbers but effectively arranging numbers in rows and columns based on certain visibility criteria. This arrangement is a permutation.

The skyscraper heights for a 4x4 grid range from 1 to 4. There are 4! (Or 24) distinct permutations. Such permutations are the foundation for possible solutions. By comparing visibility criteria against each permutation, we can decide which permutations are valid solutions.

```
array[1..24, 1..(N+1)] of int: perms_visibility = [| 1,2,3,4, visible([1,2,3,4]) |
                                                    1,2,4,3, visible([1,2,4,3]) |
                                                    1,3,2,4, visible([1,3,2,4]) |
                                                    1,3,4,2, visible([1,3,4,2]) |
                                                    1,4,2,3, visible([1,4,2,3]) |
                                                    1,4,3,2, visible([1,4,3,2]) |
                                                    2,1,3,4, visible([2,1,3,4]) |
                                                    2,1,4,3, visible([2,1,4,3]) |
                                                    2,3,1,4, visible([2,3,1,4]) |
                                                    2,3,4,1, visible([2,3,4,1]) |
                                                    2,4,1,3, visible([2,4,1,3]) |
                                                    2,4,3,1, visible([2,4,3,1]) |
                                                    3,1,2,4, visible([3,1,2,4]) |
                                                    3,1,4,2, visible([3,1,4,2]) |
                                                    3,2,1,4, visible([3,2,1,4]) |
                                                    3,2,4,1, visible([3,2,4,1]) |
                                                    3,4,1,2, visible([3,4,1,2]) |
                                                    3,4,2,1, visible([3,4,2,1]) |
                                                    4,1,2,3, visible([4,1,2,3]) |
                                                    4,1,3,2, visible([4,1,3,2]) |
                                                    4,2,1,3, visible([4,2,1,3]) |
                                                    4,2,3,1, visible([4,2,3,1]) |
                                                    4,3,1,2, visible([4,3,1,2]) |
                                                    4,3,2,1, visible([4,3,2,1]) |];
```

The dimensions of the array *perms_visibility* are *[1..24, 1..(N+1)]*, Which means there are 24 rows (for each permutation) and *N+1* columns (N for skyscraper heights, *+1* for visibility).

Each row represents a distinct permutation of the heights 1 through 4, followed by the visibility clue for that permutation.

The visibility is calculated using the visible function for permutation [1,2,3,4] and thus the row in the array would look like: [1,2,3,4,X], where X is the visibility clue.

This approach of precomputation has some advantages:

- Instead of recalculating visibility for every alternative arrangement during the solving process, the solver can quickly refer to the precomputed visibility clue, which speeds up the solving.
- The computational complexity is reduced through the process of simplifying the problem into a lookup method. The solver does not need to repeat iterations to check visibility, rather it checks if a given permutation is feasible based on the given clues.

Handling row and column table:

```
% for Rows
constraint forall(i in 1..N) (
  (left_clue[i] = 0 \/ table([grid[i, 1], grid[i, 2], grid[i, 3], grid[i, 4], left_clue[i]], perms_visibility)) /\
  (right_clue[i] = 0 \/ table([grid[i, 4], grid[i, 3], grid[i, 2], grid[i, 1], right_clue[i]], perms_visibility)) /\
 all_different(grid[i, 1..N])
);


% for Columns
constraint forall(i in 1..N) (
  (top_clue[i] = 0 \/ table([grid[1, i], grid[2, i], grid[3, i], grid[4, i], top_clue[i]], perms_visibility)) /\
  (bottom_clue[i] = 0 \/ table([grid[4, i], grid[3, i], grid[2, i], grid[1, i], bottom_clue[i]], perms_visibility)) /\
 all_different([grid[j, i] | j in 1..N])
);
```

Left Clue: If left_clue[i] is not 0, which indicates that the current row's arrangement must be present in the perms_visibility table.

Right Clue: similar to the left clue but the sequence of the row is reversed to ensure clear visibility from the right perspective.

Top Clue: The principle applied is similar to the left clue, but the column's arrangement is considered.

Bottom Clue: Similar to the top clue but the column being reversed to accommodate visibility from the bottom.

Finally, the *all_different* constraint ensures that each number within a row or column is distinct, hence leads to the Skyscraper puzzle's fundamental rule.

Now that the visibility function, permutations, and table constraints have been explained in detail, the model is complete, and the solver is ready to find a correct solution. In our first model, we have discussed how to call on the solver and to display the output.

Chapter 4

Evaluation

In this chapter, we'll use established metrics to evaluate the performance and attributes of the two developed models. The purpose is to provide an understanding of strengths and limitations of each model in solving the skyscraper puzzle.

4.1 Evaluation Metrics

4.1.1 Guessing

Human-like Guessing:

When humans solve puzzles like skyscrapers, they mostly rely on educated guesses based on given clues and their own cognitive processing. These guesses are then validated with the rest of the puzzle constraint. This "trial and error" approach reduces the solution space and especially when logical steps are not obvious. However, repeated wrong guesses can lead to longer solving time.

Solver-generated Guessing:

Solvers, like humans, do guessing when they are unable to further deduce the solution only based on constraint propagation. Solvers, on the other hand, have advantage of these guesses at speed and ruling out wrong answers. The number of guesses done by a solver gives the understanding into how direct or guess-heavy a solution path is. Fewer guesses imply a more direct problem solving approach.

4.1.2 Node Structure from MiniZinc

The node structure produced by MiniZinc Geocode Gist is a solver configuration that can be seen as a search tree.[11] Each node represents a partial solution or guess, and branching happens when multiple options are explored.

- Depth of Guessing: Deeper trees mean more levels of guessing happened.
- Breadth of Solutions: The breadth of the tree indicates how many different solutions or guesses were happened at each level.

Understanding the process of the solution space and how models traverse it can be achieved by the tree structure. Also, comparing both models through the tree structure can give information about their different efficiencies.

4.1.3 Execution Time

Even though execution time is a direct measure of how fast a model finds an answer, it's essential to look at this metric with the others. Two models might have comparable execution times, but one might take the direct path with less guesses, while the other might take a longer path with more guesses.

It's important to realize that we worry about the number of guesses not because of execution time but because fewer guesses frequently correlate with more human-like problem-solving.

4.2 Evaluation of the Algorithmic Model

To understand the performance of the 4x4 skyscraper puzzle algorithmic model we developed in the implementation chapter, it's important to analyze the output solving statistics provided by minizinc solver:

%%%mzn-stat: failures=2
%%%mzn-stat: initTime=0.016032
%%%mzn-stat: nodes=6
%%%mzn-stat: peakDepth=3
%%%mzn-stat: propagations=386
%%%mzn-stat: propagators=55
%%%mzn-stat: restarts=0
%%%mzn-stat: solutions=1
%%%mzn-stat: solveTime=0.004664
%%%mzn-stat: variables=93
%%%mzn-stat-end
%%%mzn-stat: nSolutions=1
%%%mzn-stat-end

Guessing (Failures)

Failures: 2

In a constraint solver, guessing is represented by the number of failures. Each failure is about a wrong guess made by the solver, which then backtracks to the previous state and tries different options. Two failures for the algorithmic model showed that the model was efficient, but there were times it couldn't deduce the next step only by logic and needed to guess.

Execution Time

solveTime: 0.004664 seconds

The Execution time is an obvious metric to check for efficiency. In this instance, the algorithmic model was solved in about 4 milliseconds. It's an impressive speed that shows

the model is efficient for 4x4 puzzle grids. However, the puzzle's complexity and the grid size can significantly impact this time.

In this study, our primary focus is on understanding the solver's decision-making process for the Skyscraper puzzle, mainly how it mirrors human intuition and reasoning. While execution times and scalability are often valued metrics, we won't emphasize them as they don't correspond with our main objective.

Node Structure Analysis

In minizinc, using geocode gist solver configuration, can visualize the complete node structure for the developed model as shown in figure 4.1

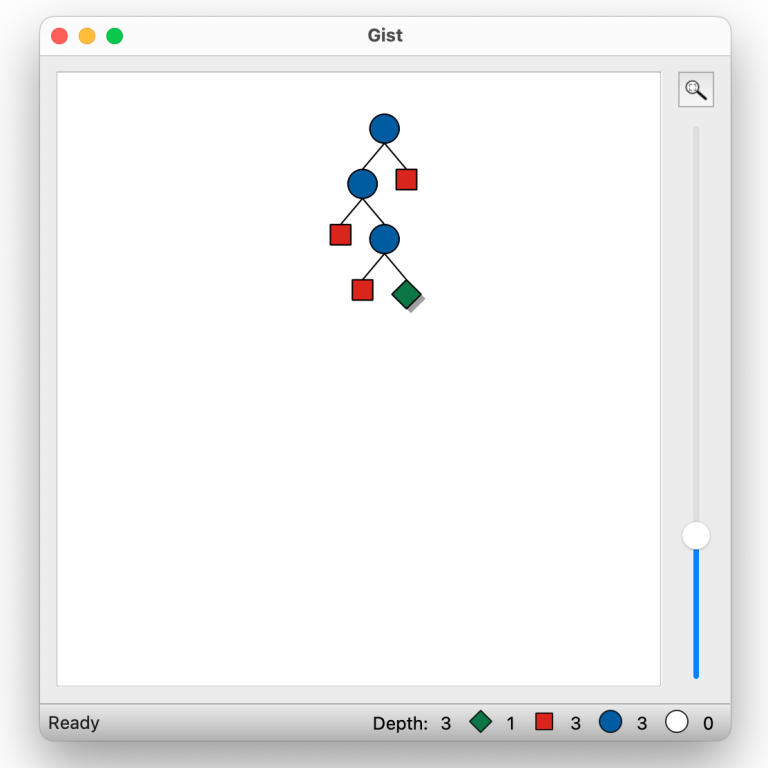

Figure 4.1: Node structure for algorithmic model

After going through the inspect features for each node, gist console provides the array like structure of what solver doing each node:

Array structure for the initial node:

grid = array2d(1..4, 1..4, [[2..4], {1,3}, {1..2,4}, [1..3], 1, 2, 3, 4, [2..4], {1,3}, {1..2,4}, [1..3], [2..3], 4, [1..2], [1..3]]);

Here, based on the additional constraint, the solver quickly filled the position of [1,2,3,4] and [4]. And brackets [2,4], denote ranges where the solver hasn't concluded the exact value for that position yet.

Array structure for the second node:

grid = array2d(1..4, 1..4, [4, 1, 4, [2..3], 1, 2, 3, 4, 2, 3, [1..2], [1..2], [2..3], 4, [1..2], [1..3]]);

In the second node, the solver has made some decisions. The grid starts to take shape with specific numbers assigned in the initial row. However, uncertainties still exist, such as [2..3] shows that the solver is still considering multiple possibilities for that position.

Continuing the same process for remaining nodes until reaching the final solution in the last green node:

grid = array2d(1..4, 1..4, [4, 3, 1, 2, 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1]);

The Algorithmic model is efficient but the presence of two failures shows some of guesswork. Although the number of nodes isn't too high, it would be optimal to reduce guesswork altogether which ensures a logical path to the solution. This reduction would not only accelerate solving time, but also ensure that the model is similar in behavior to a human solver which is navigating the puzzle through logic rather than chance.

Next, we'll evaluate the table constraint model to see how it compares and whether it can address some of the algorithmic model's limitations.

4.3 Evaluation of the Table Constraint Model

Table constraints are basically designed to mimic human reasoning by narrowing the scope of possible solutions at every step of the problem-solving process. This is similar to how a human would look at a skyscraper puzzle grid and eliminate possibilities based on visible clues.

The output solving statistics of table constraint model from implementation chapter:

%%%mzn-stat: failures=0
%%%mzn-stat: initTime=0.000729
%%%mzn-stat: nodes=1
%%%mzn-stat: peakDepth=0

%%%mzn-stat: propagations=31
%%%mzn-stat: propagators=0
%%%mzn-stat: restarts=0
%%%mzn-stat: solutions=1
%%%mzn-stat: solveTime=0.000104
%%%mzn-stat: variables=21
%%%mzn-stat-end
%%%mzn-stat: nSolutions=1
%%%mzn-stat-end

From above statistics, there are 0 failures, or no guessing was needed in the solving process. And execution time is more efficient than the algorithmic model.

In table constraint model, the node structure is only one as shown in figure 4.2

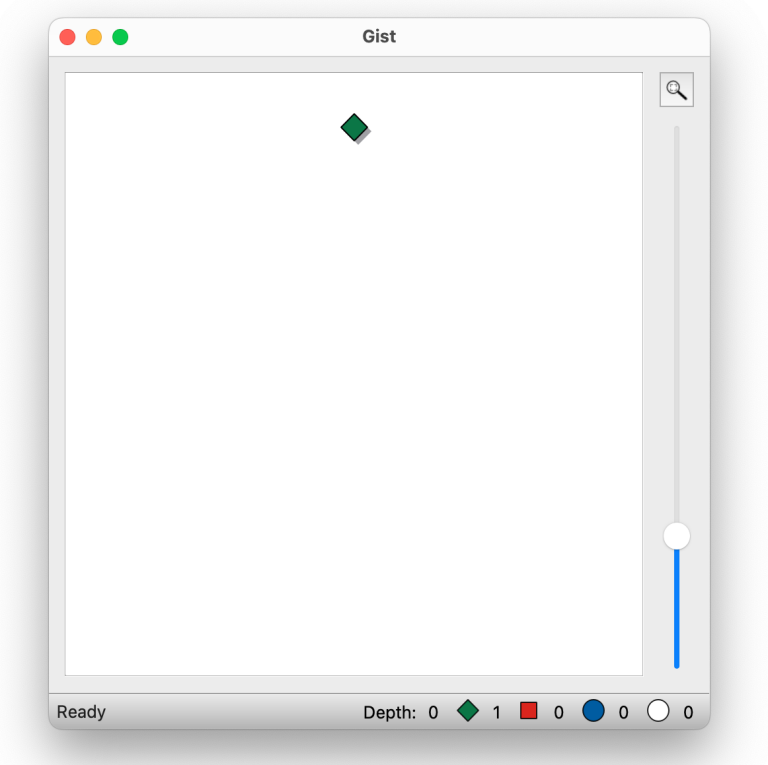

Figure 4.2: Node structure for table constraint model

With being only one node structure displayed by table constraint and so the inspection of array structure from green node would be final possible solution:

grid = array2d(1..4, 1..4, [4, 3, 1, 2, 1, 2, 3, 4, 2, 1, 4, 3, 3, 4, 2, 1]);

In constraint satisfaction problems, arc consistency[10] is a basic concept that reduces the search space significantly. In Model 2, the utilization of table constraints ensures that the constraints are arc-consistent at the time of initialization. When a constraint is arc-consistent, every value in the domain of each variable in the constraint can be extended to a feasible solution, reducing the number of guesses required.

Take metrics of Model 2 which has zero failures (%%%mzn-stat: failures=0) and only a single node (%%%mzn-stat: nodes=1). This can be directly attributed to the arc-consistency used by the table constraint, which filters out incorrect values before they can be inserted into the grid. Therefore, there is no need to make guesses.

Despite its efficiency, there are some situations where the table constraint will require guesswork:

- Multiple Solutions: When there is more than one valid solution to a puzzle, guessing may be required to go down any of the multiple valid paths.
- Incomplete Propagation: If the table constraint cannot achieve arc-consistency in a puzzle due to limitations in row or column reasoning, then guesswork will be required.
- Complex Constraints: The table constraint method alone wouldn't be enough if the puzzle has constraints that involve more than one row or column at a time.

In summary, the table constraint model not only improves solver metrics but also closely resembles human reasoning processes. Upon comparison with Model 1, it's proof that the table constraint model is the preferred choice for solving skyscraper puzzles, fulfilling the primary objectives of this study more effectively.

Chapter 5

Conclusion

In this study, we set out to understand and implement constraint programming (CP) models to solve the Skyscraper puzzle. And we developed Two distinct models: the Algorithmic Model and the Table Constraint Model. The primary objective was to identify which model aligns more closely with human-like reasoning in puzzle solving. And the Table Constraint Model was more like a human-centric approach, particularly in its efficient constraint propagation and reduced need for guesswork.

Future Work:

- We could expand the current implementation by using table constraint to more complex puzzles, thus assessing its scalability and flexibility.
- Our constraint models have potential to further enhance to more closely mimic human thought processes. This could lead to more artificial intelligence[12] systems which go beyond puzzle-solving.
- Also using advanced optimization techniques to both existing models. This would be useful when dealing with larger and complicated problem sets.
- Another possible option is to explore the synergistic effects of combining multiple constraint models, with the objective of universally efficient solving strategy.

By following these ways, we can extend the existing work and contribute more to the field of constraint programming and cognitive computing.

Bibliography

[1] F. Rossi, P. Van Beek, and T. Walsh, Handbook of Constraint Programming, Elsevier, 2006.

[2] Andrea Rendl, A Modeling Language for Constraint Programming, 2006.

[3] Krzysztof R. Apt, Principles of Constraint Programming, Cambridge University Press, 2003.

[4] Minizinc Documentation: https://www.minizinc.org/doc-2.7.6/en/index.html

[5] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. MiniZinc: Towards a standard CP modelling language. In Proceedings of the 13th international conference on principles and practice of constraint programming, 2007.

[6] Alex. Bellos, Puzzle Ninja: Pit Your Wits Against The Japanese Puzzle Masters, Book, 2017.

[7] Skyscraper example puzzle: http://www.ukpuzzles.org/forum/viewtopic.php?t=2268

[8] Alice M. Lynch, A human centered approach to logic puzzles, Conference: The Annual Symposium on Computer-Human Interaction in Play, 2021.

[9] Guillaume Escamocher, Barry O'Sullivan, Solving Logic Grid Puzzles with an Algorithm that Imitates Human Behavior, 2019.

[10] Mairy, Jean-Baptiste, Yves Deville, and Christophe Lecoutre, Smart Table Constraint, 2005.

[11] Bessiere, Christian, Bruno Zanuttini, and César Fernández, Measuring Search Trees, 2007.

[12] Mariam Khaled Alsedrah, Artificial Intelligence, 2007.

.