An SMT-Based Solver for the Aquarium Logic Puzzle

Matthew Casey and Pranav Phadke

Abstract

This paper explores an SMT-based approach to solving the logic puzzle Aquarium. In Aquarium, a player attempts to fill several blocks, designated “aquariums”, on a rectangular grid by following numerical hints listed next to the rows and columns. We describe a program in Python that takes in a starting board and returns a solution, or lack thereof. The program uses the starting board to generate $n \times m$ variables, where n is the width of the board and m is the height of the board. The state, filled or empty, of the variables, is determined by their relative position within an aquarium and its corresponding row and column hints. The program ensures correctness by making sure the sum of variables in a row or column with filled states, represented by the number 1, corresponds to its row or column's hint. As a result, this program computes a correct solution to a given board; however inputting a representation of each board is time-consuming; thus, future variations of this solver must incorporate a more efficient means to typing in initial board states. All source code is found at https://github.ccs.neu.edu/mattcasey02/CS2800-Final-Project.

1 Introduction

Aquarium is a logic puzzle played on a rectangular grid divided up into designated blocks called aquariums. A player attempts to solve the puzzle by “filling” in the aquariums to a certain water level, or leaving them empty. Figure 1 shows a grid with a single aquarium completely filled. In any given aquarium, the water level must be consistent throughout the width of an aquarium. Figure 2 shows an aquarium with a consistent water level while Figure 4 shows an aquarium with an inconsistent water level. Additionally, all water in an aquarium must have either water or a barrier below it. Figure 3 shows an aquarium that doesn't follow this rule since it has floating water.

In practice, a player refers to the numerical hints associated with columns and rows to cross out squares that cannot be filled with water while filling in those that must be filled. For example, figure 5 shows a correct first move. By examining the last row and its hint "1", the player would recognize that only the first square in that row could be filled. Filling in any other square in that row requires the rest of the aquarium to be filled as well. Figure 2 shows the resulting board, which violates the row's hint of "1". Further, by filling in the first square of the last row, the player can cross out the rest of the squares in that row. Additionally, the first move also satisfies the hint in the first column. Thus, the player can cross out the rest of the squares in that column. Because the first row has a hint of 3 and one of the squares in the first aquarium in that row is crossed out, the player can deduce that the last three squares of that row must be filled. Additionally, due to the filling property of aquariums, the last two squares in the second row must be filled as well. The culmination of these steps is seen in Figure 6.

By following such logical steps while satisfying the provided hints, the player can solve an Aquarium puzzle board. However, on boards of larger sizes, solving these puzzles takes substantially longer due to the vast amount of hints a player has to keep track of at all times. The purpose of this project was to use a constraint based approach to solve Aquarium game boards like a human player but at a much faster rate. This allows for quick and accurate solutions to larger and more complex game boards.

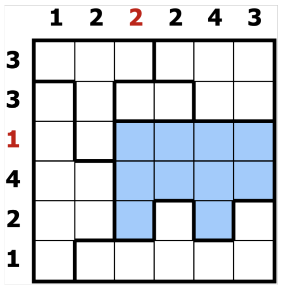


Figure 1: Example of a Grid With a Completely Filled Aquarium

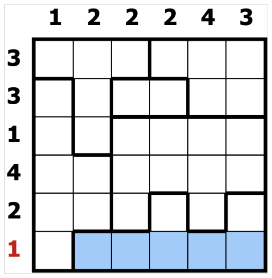


Figure 2: Example of an Aquarium With a Consistent Water Level

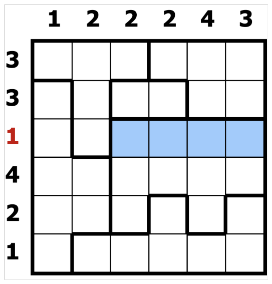


Figure 3: Example of an Aquarium With Floating Water

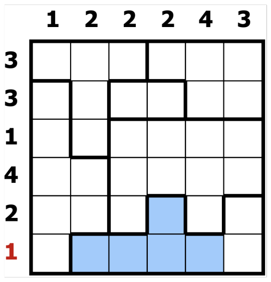


Figure 4: Example of an Aquarium With an Inconsistent Water Level

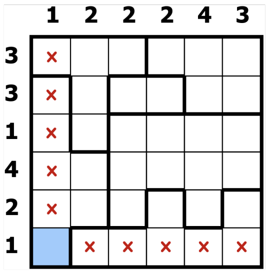


Figure 5: Example of a Correct First Move

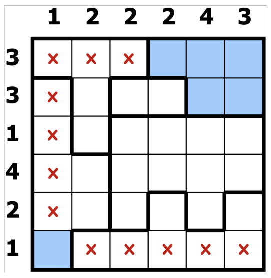


Figure 6: Example of A Board in Progress of Being Solved

2 Related Work

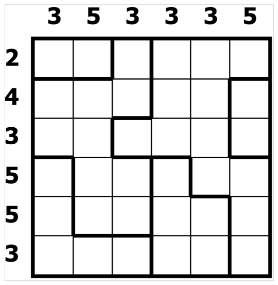
There exist various SAT/SMT based solvers for popular logic puzzles. These types of solvers are well suited for logic puzzles since these puzzles have easy to define rules which can be expressed as boolean formulae. Such solvers work best for "one-player" puzzles, such as Sudoku and Aquarium, due to their shallow depth of possible moves. Examples of problems solved using SAT/SMT algorithms include Sudoku, Dog Cat & Mouse, and Eight Queens. The solutions for these puzzles, using an SMT solver, are available as part of the documentation of the Z3 library [1]. Our extensive search of prior art revealed no SAT/SMT solution to the Aquarium puzzle. Since Aquarium is a logic puzzle with a small set of defined rules, it is an ideal candidate for the SAT/SMT approach.

3 Game Board Encoding

When determining an encoding, it is important to consider what information is key to understanding the game. An encoding should contain the information required to solve the puzzle. Sudoku, for example, requires the location and value of squares that have already been filled. The board size is always 9x9 so it is unnecessary to pass in all the squares whose values have not yet been determined.

There are two key pieces of information required to solve aquarium. The first is the number of filled cells in each row/column. This information is written as the top row and left column in the puzzle. The second key piece of information is the location of the borders of all the aquariums. These borders are represented by dark black lines, while the borders between cells within an aquarium are represented with thin black lines. The dark blacks can be represented by 1s and the thin blacks by 0s.

We choose to represent any given Aquarium puzzle as a list of lists. This way, our encoding matches the appearance of the aquarium puzzle. Although translation from a puzzle to this encoding must be done manually, the task is made easier by the similar appearance of the two. An example of this encoding as applied to a 6x6 board is show above in Figure 7.



```
((3,5,3,3,3,5),             Column filling rules
 (1,1,1,1,1,1),             Top horizontal walls
 (2,1,0,1,1,0,0,1),         Vertical walls in row 1
 (1,1,0,0,0,1),             Walls between rows 1&2
 (4,1,0,0,1,0,1,1),         Vertical walls in row 2
 (0,0,1,0,0,0),             Walls between rows 2&3
 (3,1,0,1,0,0,1,1),         Vertical walls in row 3
 (1,0,1,1,0,1),             Walls between rows 3&4
 (5,1,1,0,1,1,0,1),         Vertical walls in row 4
 (0,0,0,0,1,0),             Walls between rows 4&5
 (5,1,1,0,1,0,1,1),         Vertical walls in row 5
 (0,1,1,0,0,0),             Walls between rows 5&6
 (3,1,0,0,1,0,1,1),         Vertical walls in row 6
 (1,1,1,1,1,1))             Bottom horizontal walls
```

Figure 7: Example Encoding of a 6x6 Board

The first sublist encodes the number of filled cells in each column. The second sublist is the encoding for the top horizontal walls in the board. Since the top border in every puzzle is a solid black line, it is always a list of 1s with length equivalent to the width of the board. The third sublist is the encoding of the vertical walls in the first row, using 1 for a dark border and 0 for a light one. Additionally, the first item in this sublist represents the number of cells in the first row that must be filled. The remaining sublists alternate between representing a row of horizontal walls and a row of vertical walls.

Using this information, a player can both deduce the solution, or lack thereof, to any given Aquarium puzzle. The actual solution to the puzzle can be expressed as a list of lists containing 0s and 1s representing not-filled and filled cells respectively.

4 Constraints for Solving

Our program follows 5 rules to determine satisfiability.

1. Every defined variable, corresponding to a square on the board, has a value of either 1 or 0. This ensures that the variables hold a binary state corresponding to true and false, or "filled" and "empty."

2. The sum of all variables in a row equals the corresponding row's numerical hint. This ensures the amount of filled tiles in a row is consistent with the board's hint.

3. The sum of every variable in a column equals the corresponding column's numerical hint. This ensures the amount of filled tiles in a column is consistent with the board's hint.

4. A filled cell either has a wall beneath it or the cell beneath it is also filled. This ensures the depth aspect of Aquariums is consistent throughout the grid (i.e. no floating water is allowed).

5. The water level in an aquarium is consistent, even when walls separate cells that are in the same aquarium. This accounts for normal and edge cases to the consistent water level rule.

As long as these 5 rules are met, a board must be to be satisfiable and the program will output a solution. No more rules are necessary as these 5 ensure the boards hints are satisfied and the water level and depth aspects of aquariums are met.

5 SMT-Based Solving

Since our encoding and our constraints both use the theory of integers, we choose to use an SMT solver to determine the satisfiability of a given Aquarium puzzle. The difference between an SAT and SMT solver is that an SMT solver allows the use of the theory of integers. This is important for this logic puzzle since we need to sum the numbers in each column and row (for constraints 2 and 3). Using an SAT solver, the same thing can be achieved but in a much less concise manner.

We choose to use the Z3 solver, a common SMT solver available in many languages. We use Python 3, and use the z3-solver package [1]. We begin by creating a list of list of integer variables. The Z3 solver will see these as variables and try assigning values to them such that the constraints it is given are met. We then encode the 5 constraints described in Section 4, using the built in abilities of the Z3 solver. The creation of each constraint is similar. We loop over the list of list of integer variables, and for each variable in the list we assign some constraint to it. Z3 has And, Or, and If built in to aid in the creation of these constraints. Further, Z3 has Sum built in which takes in a list and calculates its sum. This function is used to encode constraints 2 and 3, summing of rows and columns, respectively.

Once all the constraints are encoded, the constraints are passed into a solver which outputs whether the puzzle is satisfiable or not. If it is, the solver provides an example of a list of list of integers which satisfies the constraints. Since the output will look identical to the puzzle itself, translation back to the puzzle consists of filling in all cells with a 1 in the analogous location in the solver's output.

To check the correctness of the code, we manually encoded multiple interesting examples from www.puzzle-aquarium.com, ran the solver on them, and transcribed the results onto the website. For all puzzles tested the solver outputted a correct result. We also tested examples of impossible puzzles and ensured that the solver returned unsatisfiable. The examples chosen tested all the criteria outlined in Section 4 to ensure that all were transcribed to code correctly.

6 Discussion

The repository includes several working cases of the program on puzzles of varying sizes. Thus, we conclude that the program generalizes to any size board. Our program also generalizes to any rectangular Aquarium board, allowing users to create their own examples. The repository includes instructions on how one may transcribe an initial board state and run it within our program. A limitation to our solver is that transcribing boards takes a substantial amount of time as every single hint and aquarium border (and lack of border) needs to be transcribed into a list of lists of integers.

7 Conclusion

In this paper, we discuss our successful implementation of solving the logical puzzle Aquarium through the Z3 SMT Solver package for Python. We begin by discussing the rules to Aquarium such as its depth and width rules for individual aquariums that makes it difficult to solve. Afterwards, we examine several introductory steps a player may take in solving an example board to both help portray how one may attempt to solve the puzzle and emphasize the depth and width rules to aquariums. Section 3 discusses how to encode an aquarium game board. Due to the complex shapes of aquariums, we determine that the two important pieces of information required to solve the puzzle are the numerical hints provided to rows and columns and the specific locations of aquarium borders. As such, we transcribe aquarium borders, and the lack thereof, through binary bits and represent hints by their integer values.

After discussing the relevant information required for someone to solve the puzzle, we set up several constraints necessary for the program to solve any given Aquarium puzzle. Lastly, we discuss the reasoning as to why we choose an SMT-Based solver, known as the Z3 solver, and explain how the program is encoded. Since variables in our program's representation of the board have binary states, we use the theory of integers to sum rows and columns of variables and compare them to their corresponding hints. While creating a program for this game is possible through an SAT solver, it would be much less concise than our current program. In conclusion, the program is generalized to any size rectangular puzzle. However, a limitation to our program is the vast amount of time needed to transcribe an initial board into the solver.

8 Our Progression

When originally planning out how to solve this puzzle, we first looked towards attempting to encode the solver in ACL2. We were going to use defdatas to represent the different constraints and use the built in theorem solver of ACL2 to compute logical solutions to given boards. However, we had scrapped this idea after reconsideration and advice from Professor Hemann. We then thought of constructing this program in a language which we were much more familiar with - Java. But we quickly realized the limitations we had as encoding constraints in Java would most likely take significant time and nevertheless would involve complex classes and methods. After having scrapped these two ideas, we stumbled upon the example python program Professor Hemann uploaded. This used the SMT-based Z3 solving package in a way that seemed pretty simple to encode once the basics of the package were understood. Since we didn't have Python experience it took some time for us to understand the example code. When we finally understood how the Z3 package works, we were able to encode our Aquarium solving program.

9 References

1. https://github.com/Z3Prover/z3