Solving Smullyanic Dynasty and Canal View Puzzles with Z3

Fabio Somenzi

September 16, 2025

Abstract

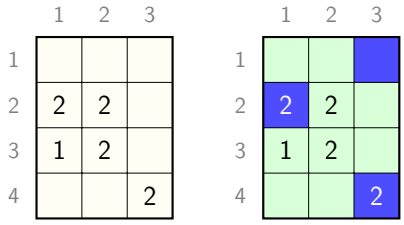
This document describes the programming project for the Fall 2025 edition of ECEN 2703.

Contents

1 Introduction

In our treatment of logic, we have learned how to solve knights-and-knaves puzzles. We have also seen how to apply logic to the Minesweeper problem. The Z3 tutorial that you read shows one way to solve Sudoku puzzles by encoding the constraints that every row, column, and 3×3 box of the grid must contain the digits 1 through 9. For our programming project, we take on two *pencil puzzles*, in which the solver has to shade squares of a rectangular grid so as to satisfy given constraints. As we shall see, our two puzzle genres have a lot in common.

Example 1. In a *Smullyanic Dynasty* puzzle[1] we are given a rectangular grid. Some squares contain nonnegative integers, as in the 4×3 grid below.



The solver needs to shade some squares according to the rules below. A shaded square is a *knave*, while an unshaded square is a *knight*. The numbers in the squares are the *clues*.

1. Shaded squares cannot share an edge. However, they may share a corner. This is the *separation* rule.

2. The unshaded squares form an edge-connected region. That is, every unshaded square must share an edge with some other unshaded square in such a way that one can go from every unshaded square to any other unshaded square. (Two squares that only touch in a corner are not edge-connected.) This is the *connection* rule.

3. A clue is truthful if, and only if, it is in a square that is a knight.

4. A truthful clue counts the number of knaves in the *domain* of its square. The domain of a square consists of the square itself and the up to eight squares surrounding it. The domain of a clue is the domain of the square containing it.

The solution to the puzzle above is shown to its right. (We say "*the* solution" because, in well-designed puzzles, the solution is unique.) If this is the first Smullyanic Dynasty puzzle you've ever come across, you may want to cover the solution and see if you can re-discover it. In Section 2, we'll talk about "tricks of

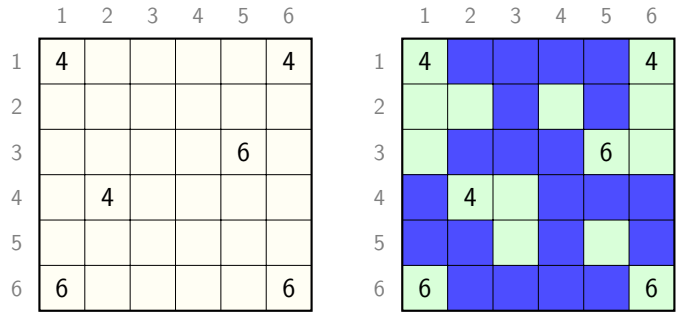
[1] The creator of this puzzle genre, who goes by the name of zotmeister, meant his invention to be a homage to Raymond Smullyan. The reason he gave for using the word "dynasty" leaves something to be desired.

the trade" that help humans to tackle puzzles of this genre. In any event, even if a puzzle baffles you, it is straightforward to check that a purported solution obeys the rules. (You should try it on the solution above.) Showing that a solution is unique is more involved.[2]

Example 2. In a Canal View puzzle, we are given a rectangular grid like the one below, due to Eric Fox. The solver needs to shade some squares so that they form a connected region—the canal—with no 2×2 "pools." (No 2×2 region may be completely shaded.)

Some squares contain numbers. Those squares must not be shaded. The numbers in them (the clues) indicate how many squares of the canal are visible from them, both horizontally and vertically. An unshaded square blocks the view. So, for example, `r1c1` (Row 1, Column 1) sees no canal squares vertically because `r2c1` is unshaded.

A square may also contain a question mark. (Our example does not have any, but some of the puzzles your program will solve do.) A square with a question mark may not be shaded, but imposes no constraint on the number of canal squares visible from it. This genre was invented by Prasanna Seshadri.



You may have noticed that the two shading puzzle genres we are going to tackle have one important rule in common, which we call the `connection` rule: either the unshaded squares, or the shaded squares must form an edge-connected region of the grid. We'll start our project with Smullyanic Dynasty, which will teach us how to deal with such connectedness constraints. This ability will make the task of solving Canal View puzzles easier.

These notes aim to be self-contained. If you want to know more, Knuth [4] and Hearn and Demaine [3] are good general references on puzzles, though neither book covers our puzzles. Knuth covers solution techniques, while Hearn and Demaine focus on computational complexity.

[2] Canal View was shown to be NP-complete and, in fact, ASP-complete in [5]. Simplifying a bit, ASP completeness implies that proving uniqueness of the solution is also computationally hard. To the best of my knowledge, no such results have been published for Smullyanic Dynasty.

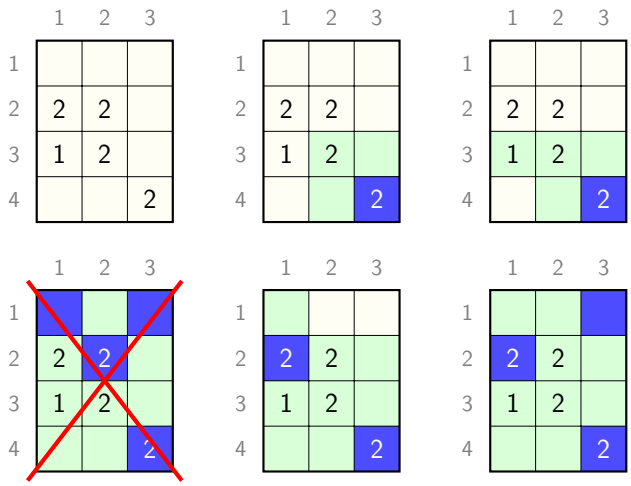


Figure 1: Stages in a solution of the Smullyanic Dynasty puzzle of Example 1.

2 Solving Smullyanic Dynasty Puzzles

While in principle one could write a solver for Smullyanic Dynasty puzzles without having ever tackled one puzzle, some familiarity with the manual solution process will definitely help. Hence, in this section, we look at some of the basic techniques on which human solvers rely. There are a few observations whose application helps us solve an example of Smullyanic Dynasty puzzle like the one of Example 1.

- Throughout the solution, we'll be looking for squares that must be shaded as well as for squares that must not be shaded. We'll use pale green to mark the latter. Whenever we shade a square, we mark the squares that share an edge with it as "unshaded." (That is, we paint them green.)
- Whenever we shade a square, we should check whether other squares must be painted green to let all unshaded squares connect in one region.

These observations often help a solver to make progress toward the solution. In Figure 1 we apply them to the solution of the puzzle of Example 1.

As in the solution of classic Knights and Knaves problems, we try to reach contradictions from assumptions we make. A contradiction tells us that the assumption that led to it is false. In this puzzle, if `r4c3` were truthful, it would be trapped because the two shaded neighbors, which cannot share an edge, could only be `r3c3` and `r4c2`. The connection rule gave us a start! We should also notice that we have discovered a useful trick: A 2 in a corner is always shaded.

Now, if `r4c3` is shaded, its clue must be false. Since the separation rule prevents more than two knaves in the domain of `r4c3`, there must be exactly

one: `r4c3` itself. This takes us to the second grid from the left in the top row of Figure 1.

Suppose `r3c1` is a knave. Then, it has more than one knave in its domain. However, its domain is a proper subset of the domain of `r3c2`. Having two or more knaves in `r3c1`'s domain would give `r3c2` more than two shaded neighbors. This cannot be because we already decided that `r3c2` is a knight. Besides, it would result in two knaves next to each other. Hence, `r3c1` is a knight. The rightmost grid in the top row of Figure 1 shows the current state of affairs.

We now focus on `r2c2`. Observe that its domain includes the domain of `r2c1`. If `r2c2` were a knave, it would have more than 2 knaves in its domain because, otherwise, `r2c1` would also be a knave. That would give adjacent knaves, which is against the rules. However, the only way to place three knaves in the first two rows of the grid traps `r1c2`, as shown in in the leftmost grid in the bottom row of Figure 1. Hence, `r2c2` is a knight, while `r2c1` is a knave to make the clue in `r3c1` truthful. This is shown in the center grid in the bottom row of Figure 1.

We can see that `r1c2` must be unshaded in a couple of ways. One the one hand, shading `r1c2` would trap `r1c1`. On the other hand, if `r1c2` were shaded, `r2c1` would be truthful, in spite of being a knave. Finally, `r1c3` must be shaded to put a second knave in the domain of `r2c2`, and that completes the solution.

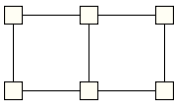
Note how the systematic application of rules and observations derived from them eliminated the guesswork: we never had to say, "let's shade this square and see what eventually happens." We did consider the consequences of making wrong decisions, but the resulting contradictions emerged immediately. Many puzzle enthusiasts subscribe to the idea that a good puzzle is one that can be solved by an experienced human solver without any *bifurcation*. In Computer Science, we'd say, "without backtracking."

It should be clear that the one we followed is one of the possible sequences of steps that may be taken to solve the puzzle. For example, when we focused on `r2c2`, we may have considered the consequences of leaving `r2c1` unshaded instead. Our solution path does illustrate, though, the type of (often subtle) reasoning that puzzles often require of human solvers. Mechanical solvers, like Z3, may not have been taught all the tricks; hence, they may resort to backtracking when a human solver wouldn't. However—fortunately for us—they have brute force to spare.

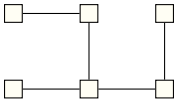
3 Spanning Trees

Our first objective is to write a Python program that uses Z3 to (mechanically) solve Smullyanic Dynasty puzzles. To that effect, it is useful to think of the grids on which we play these puzzles as *graphs*. Every square of the grid is a *vertex* of the graph. Two squares that share an edge in the grid are connected by an *edge* in the graph.[3] A 2×3 grid seen as a graph is shown below.

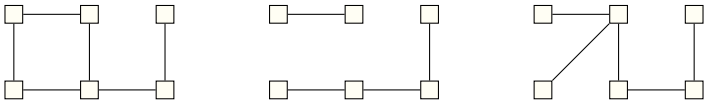
[3]The reason why we use the words "vertex" and "edge" in describing graphs has to do with the early applications of graphs to polyhedra (like the cube or the tetrahedron) that



A *tree* is a connected graph with no loops. Continuing our example, the following is a tree.



This tree has the same set of vertices as the graph above and connects all of them using (some) edges of the graph. We call such a tree a *spanning tree* for the graph. The following are *not* spanning trees for our graph. The one on the left is not a tree because it has a loop; the one in the middle is not a tree because it does not connect all the vertices; and the one on the right is a tree, but it is not a spanning tree of our graph because it has an edge that is not in the given graph.



A spanning tree for a graph exists if, and only if, the graph is connected: if there is no path between vertices u and v of a graph, no such path may be created by removing edges from the graph. However, is a graph is connected and has loops, we can repeatedly remove an edge from one loop until none are left. Spanning trees are important for our puzzle solvers because we'll ask Z3 to guarantee connectedness of a grid region by producing a spanning tree for the corresponding (sub)-graph of the grid graph.

A course that covers graph algorithms would likely introduce you to one or more ways to compute a spanning tree of a graph. Depth-first and breadth-first searches of a graph, for example, can return a spanning tree. All spanning trees of n vertices contain $n-1$ edges. However, if including edge e_i in the tree incurs cost c_i, then a spanning tree of minimum cost can be computed efficiently by either Kruskal's or Prim's algorithms. All these algorithms are beautiful and worth knowing about, but for our purpose, we need something different. The reason is that Z3 has to simultaneously choose which squares to shade and show that the remaining squares are all connected.[4]

This is the key observation: Given a tree, we can choose a vertex as the *root* of the tree. All other vertices have a *parent* vertex in the tree: the next vertex on their path to the root. In our solvers, we choose the tree by assigning a parent to each vertex to be connected. We want the path from a vertex to its parent, then to the parent's parent, and so on, to end at the root. For this to happen we must prevent the path from forming a loop.

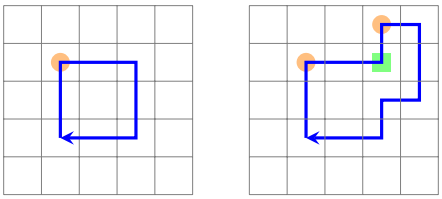
have—you guessed it—vertices and edges.

[4] For Canal View, it will be the shaded squares that have to remain connected.

4 Preventing Loops

In Smullyanic Dynasty, we need to make sure that the parent links of the unshaded squares form no loop. In our solver, we need to specify appropriate constraints for Z3. We adopt an approach based on [1, 2], whose main ideas we summarize here.

Let's look at the two loops below and suppose that we go around them in *clockwise* fashion, as shown by the arrows. (We'll stick to clockwise travel throughout.)



The loop on the left contains four turns, while the loop on the right contains eight turns. We identify these turns by the directions before and after the turn. Let's focus on two types of turns: up-then-right (marked by orange circles) and right-then-up (marked by green squares). Let's call these turns *notable*. In the loop on the left, there is one up-then-right turn and no right-then-up turns. In the loop on the right, there are two turns of the first type and one turn of the second. A few more examples will convince you that in every (clockwise) loop there is always one more up-then-right turn than there are right-then-up turns. So, the number of notable turns is always odd. Intuitively, the turns of the second type counter the attempt of the turns of the first type to close the loop. Since the loop closes, the turns of the first type must have the last word.[5]

It is also not difficult to see that one cannot take two up-then-right turns without an intervening right-then-up turn. Hence, the difference between the numbers of notable turns of the two types is always 0 or 1 along the loop and is 1 at the end. This suggests that we can track this difference with just one bit. (More details and actual proofs of all these claims are found in [1, 2].)

Let's see how to convert this idea into constraints for Z3. We associate one Boolean variable to each square of the grid. We then impose constraints saying that, if the loop turns up-then-right or right-then-up in that square, the turn-tracking variable of this square must have opposite value to that of its predecessor along the path. In the remaining 10 ways in which the loop goes through a square, the turn-tracking variable must have the same value as the variable from the predecessor square. The turn-tracking variables of the shaded squares will take arbitrary values and will be of no concern to us.

The chain of constraints around a loop forces the turn tracking bit to change an odd number of times: If there are $n > 0$ up-then-right turns, there are $n - 1$

[5] In the choice of the two turns to consider notable, we follow [1]. In general, we could choose any two turns that, taken in sequence, produce no change in direction. For example, down-then-left and left-then down. Also note that, over an entire lap, there are four more right turns than there are left turns.

right-then-up turns, and $2n - 1$ is always odd. This means that the parity constraints around a loop are unsatisfiable! This guarantees that there are no loops. In Section 5 we'll see how to incorporate this technique in our encoding of the Smullyanic Dynasty puzzles. We'll use the same approach for Canal View puzzles too.

4.1 A Simpler Solution, Which is not as Good

There are alternatives to tracking turn parity to guarantee the absence of loops. Perhaps the simplest is to associate an integer variable to each square of the grid, which, if a square is unshaded, gives the distance of the square from the root along the spanning tree. Every unshaded square that is not the (unique) root must have an unshaded neighbor whose distance is one less than its own distance. Of course, we specify the constraints and ask Z3 to assign values to these variables.

This encoding works well for small puzzles. However, as the grid grows larger, the range of the integer variables grows. (The number of digits grows logarithmically with the number of squares.) The other problem of this approach is that, while bounded integers can be encoded in propositional logic, the resulting constraints are more difficult to grok for Z3 than pure propositional constraints like the ones we have discussed earlier. Experiments confirm that our choice works better overall.

5 Modeling Smullyanic Dynasty Puzzles

We now discuss how to model a Smullyanic Dynasty puzzle for Z3. With minor adaptations, what we say applies to other SAT and SMT solvers, but we'll focus on our favorite tool.

For an $m \times n$ grid, with square $(0, 0)$ in the top-left corner, we associate six Boolean variables to each square of the grid.[6]

- $x_{i,j}$: if it is true, then square (i, j) is unshaded; otherwise, it is shaded.
- $h_{i,j,1}$ and $h_{i,j,0}$: they encode the horizontal parent link according to the following scheme:
  - `(False,False)`: no horizontal link
  - `(False,True)`: the horizontal link points right
  - `(True,False)`: the horizontal link points left
  - `(True,True)`: forbidden
- $v_{i,j,1}$ and $v_{i,j,0}$: they encode the vertical parent link according to the following scheme:
  - `(False,False)`: no vertical link

[6] Recall that, technically, these are *constant symbols*.

  - `(False,True)`: the vertical link points up
  - `(True,False)`: the vertical link points down
  - `(True,True)`: forbidden

- $p_{i,j}$: the turn parity of square (i, j) that was discussed in Section 4.

Later, we'll see that we also need a single Boolean variable, r, to select the root of the tree. Let's start with the constraints on the x variables.

- No two shaded squares should share a side: if squares (i, j) and (k, ℓ) are neighbors, $x_{i,j} \vee x_{k,\ell}$.

- If square (i, j) contains a clue, and it is unshaded, the number of shaded squares in the domain of (i, j) should equal the clue. If, however, square (i, j) is shaded, the two numbers should be different. This can be encoded as a pseudo-Boolean constraint.

If these constraints on the x variables are satisfied, but the constraints on the remaining variables are ignored, we get a *weak* solution. In a weak solution, the unshaded squares may form more than one connected region.

We now describe the constraints on the h and v variables. We need to distinguish three cases: The root of the spanning tree (which is unshaded), the other unshaded squares, and the shaded squares. First, let's consider an unshaded square that is not the root. Every such square must have exactly one outgoing edge, pointing to its parent in the tree. Let `absent(H[i][j])` be the predicate that is true if, and only if, there is no horizontal edge out of square (i, j) (and similarly for `absent(V[i][j])`). Also, let `legal(i,j)` be true if, and only if, neither the h variables, nor the v variables for square (i, j) have the forbidden value `(True,True)`. Then, if square (i, j) is unshaded, it must satisfy `legal(i,j)` and `absent(V[i][j])!= absent(H[i][j])`. In addition, if the horizontal parent pointer of unshaded square (i, j) points left, then $j > 0$ should hold and $x_{i,j-1}$ should be true (i.e., square $(i, j - 1)$ should also be unshaded). Likewise, for the other three directions.

Finally, we need to prevent two unshaded squares from pointing at each other. For example, if square (i, j) points left (which means that $j > 0$) then square $(i, j - 1)$ should not point right. (These "short loops" are not prevented by the p variables we are going to discuss shortly.)

For the root of the tree, we first observe that we can limit its position to the first two squares of the first row. This is because no two adjacent squares can both be shaded. We cannot do better than that in general: there are Smullyanic Dynasty puzzles in which the square $(0, 0)$ is shaded, puzzles in which $(0, 1)$ is shaded, and puzzles in which neither is shaded. However, when both $(0, 0)$ and $(0, 1)$ are unshaded, we can let them both be roots because, being adjacent, they are connected. So, we avoid using Boolean variables to let Z3 decide where the root will be. We may impose the constraint that a root should have no outgoing edges, but it is not necessary.

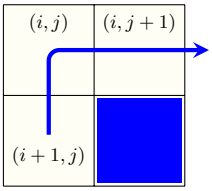
The h, v, and p variables of the shaded squares are, in principle, unconstrained. It may be convenient, though, to forbid the `(True,True)` value for all edges, regardless of a square's shading.

As we said, we associate to each square a Boolean variable $p_{i,j}$, which switches value whenever the loop takes an up-then-right or right-then-up turn. We do not insist on the particular values of the p variables: only on their relations to the neighboring variables if they belong to unshaded cells. In particular, if the values of all the p variables are flipped, from a valid solution we get another valid solution to the constraints (while the solution to the puzzle does not change).

The constraint on the p variables at square (i, j) have the following form: "If there is an incoming edge from above, and an outgoing edge to the left, then $p_{i,j}$ and $p_{i-1,j}$ should be the same. Of course, we need to consider all 12 possible combinations of incoming and outgoing edges, and ensure that the p variable we compare to $p_{i,j}$ exists. (In our example, i should be greater than 0. It is also a good idea to skip this constraint if $j = 0$ because it is impossible for a square in the first column to have a parent link pointing left.) There are 12 cases to consider instead of 16 because, once we have chosen the direction of the incoming edges (in one of 4 ways) we can choose the direction of the outgoing edge in one of the remaining 3 ways. Recall that we forbid edges that point at each other. Therefore, those 4 cases are not possible. If there is no edge from a neighbor into (i, j) the p variable of the neighbor is not (directly) constrained by the value of $p_{i,j}$.

The constraints on the p variables are important for the unshaded squares, but we can apply them regardless of the x variables.

In each of the 12 cases for which we write a constraint, we know which square the path comes from and which square it visits next. Hence, we know whether it takes an up-then-right turn, it takes a right-then-up turn, or does something else. In the first two cases, we impose the constraint that $p_{i,j}$ should have the value opposite to the p variable of the predecessor; in the last case, we impose the constraint that $p_{i,j}$ has the same value as the t variable of its predecessor. For the 4 combinations of incoming and outgoing edges that are forbidden, we specify no constraints on the p variables. As an example, consider this fragment of grid:



For these values of the x, h, and v variables, the constraint on the p variables should be,

$$(p_{i,j} \neq p_{i+1,j}) \wedge (p_{i,j+1} = p_{i,j}) \ .$$

Suppose $\mathrm{up}(i, j)$ is a function of the h and v variables that is true if, and only if, the arrow from square (i, j) points up. Likewise, for $\mathrm{down}(i, j)$, $\mathrm{right}(i, j)$ and

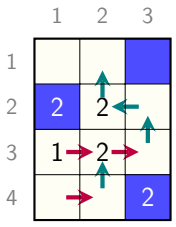


Figure 2: Possible values of the p variables for the puzzle of Example 1. The color of the arrows encodes the value of the parity variables. The tree rooted at $(0, 0)$ has just one node.

left(i, j). We need the constraints,

$$(\text{up}(i+1, j) \wedge \text{right}(i, j)) \rightarrow p_{i,j} \neq p_{i+1,j}$$
$$(\text{right}(i, j) \wedge \text{right}(i, j+1)) \rightarrow p_{i,j+1} = p_{i,j} \quad .$$

We also need other constraints: for each square, one constraint for each of the 12 combinations of corner values that are possible at that square. In practice, we want to loop over all squares and add up to 12 constraints for each of them. Why do we say "up to?" Because, for example, in the top row of the grid, the loop cannot enter or leave a square from above. Trying to refer to the square below a square in the bottom row would cause an out-of-range access, which is not nice. Perhaps even worse, trying to refer to the square above a square in the top row results in a row index of -1, which causes Python to access the *last* row of the grid.

We can also fix the value of the p variable for the root of the tree, though it is not necessary.

For the puzzle of Example 1 ($m = 4$, $n = 3$), Figure 2 shows possible values. The arrows obviously describe the values of the h and v variables. Each arrow is colored according the value of the square's p variable: red means true and teal means false. To reduce clutter, the values of the p variables are not shown for either the shaded squares or the root. Notice that the solver could have computed a different spanning tree, or reversed the assignment of truth values to the p variables. For us, a puzzle with unique solution will be one for which the shading cannot be changed without violating the constraints. We will not insist on the uniqueness of the assignment to h, v, or p variables.

6 Solving Canal View Puzzles

Let's see how to solve the puzzle of Example 2. We start from the last row of the grid, with the two sixes. To satisfy the constraint due to the six in Row 6, Column 1, we need to shade six squares chosen from Column 1 and Row 6. Since clues are always unshaded, there are only 4 squares available in Column 1. Hence, we need at least two shaded squares from Row 6 and

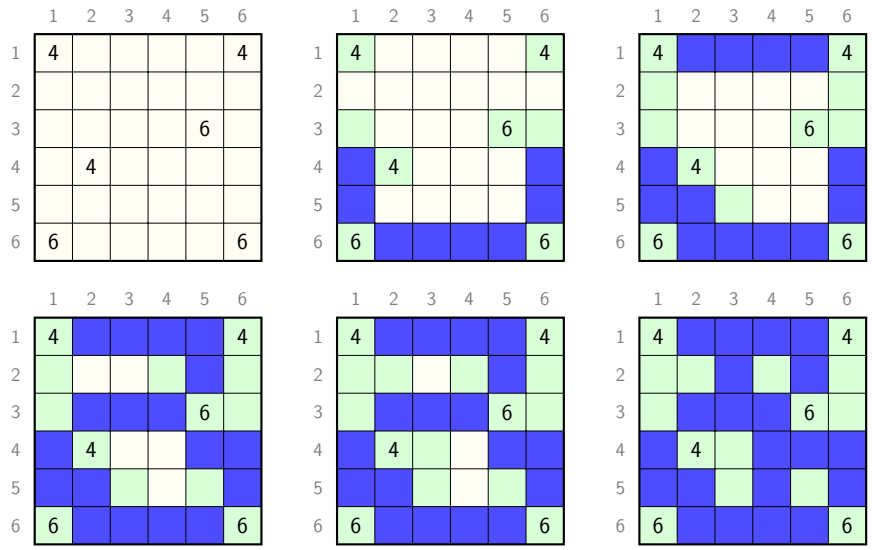


Figure 3: Stages in a solution of the Canal View puzzle of Example 2.

`r6c2` and `r6c3` must be shaded. With symmetric argument applied to `r6c6`, we conclude that `r6c4` and `r6c5` should be shaded. But then, each six in the last row sees four canal squares in Row 6 and should see exactly two squares in its column. This takes us to the middle grid of the top row in Figure 3.

We now apply the same type of reasoning to the first row of the grid. At least 3 of the 4 shaded squares seen from `r1c1` must come from Row 1. Likewise, at least 3 of the 4 shaded squares seen from `r1c6` must come from Row 1. But then, there are 4 shaded squares in Row 1, which means that `r2c1` and `r2c6` must remain unshaded.

We may also notice that the canal squares in Column 1 can only connect to the other canal squares through `r5c2`, which must be shaded as a consequence. We must then leave `r5c3` unshaded to avoid a 2×2 pool in the canal. This takes us to the rightmost grid of Figure 3.

We now focus on the 6 in `r3c5` and observe that we cannot shade both squares immediately below it, lest we should create a 2×2 pool in the canal. Therefore, the six shaded squares seen from `r3c5` must be the three squares to its left, the two squares above it, and the first square below it. To avoid pools, we must then leave `r2c4` and `r5c5` unshaded, giving us the leftmost grid of the bottom row in Figure 3.

It's time to look at the 4 in `r4c2` and notice that `r4c2` already sees 4 canal squares. Hence, `r2c2` and `r4c3` must remain unshaded. This results in the middle grid of the bottom row of Figure 3. Now, we only need to notice that all remaining squares must be shaded to give a connected canal.

We may wonder why we considered the clues in the order we did. It turns out that we followed the general heuristic of looking for the most constrained

ones. These often are the largest and smallest clues and, among the largest clues, those in the corners or along the edges of the grid.

7 Modeling Canal View Puzzles

The encoding of Canal View problems is similar to that of Smullyanic Dynasty puzzles. We associate the same Boolean variables as in Smullyanic Dynasty to each square of the grid: $x_{i,j}$ is true if, and only if, Square (i, j) is unshaded. In Canal View, moreover, we know that $x_{i,j}$ is true whenever (i, j) holds a clue.

The meaning of the h, v, and p variables is identical in the two puzzles, except that in Canal View they are used to enforce that all squares whose x variables are *false* form a single region by guaranteeing that they have a spanning tree.

A more substantial difference has to do with what the clues count. In Smullyanic Dynasty, a clue that is not '?' counts the number of shaded squares in a *fixed* set of squares (its domain). In Canal View, a shaded square in the same row or column as a clue, but not directly adjacent to that clue, counts toward the total only if all the squares between that square and the clue are shaded. Later in this section, we outline how to deal with this rule.

A final difference has to do with the root of the spanning tree. There is no guarantee that either `r1c1` or `r1c2` will be shaded. So, we are going to associate to each square one variable, $r_{i,j}$, that is true if that square is the root of the spanning tree. We then impose the pseudo-Boolean constraint that exactly one such variable is true. There is one possible optimization: if the puzzle has at least one positive clue, we can restrict the root to be one of its neighbors.

A clue must equal the sum of four numbers: the counts of the shaded squares in each cardinal direction up to the first unshaded square, or to the edge of the grid. Let's see how to count the shaded squares to the west of Square (i, j). The other three directions can be dealt with similarly.

If $j = 0$, the result is 0 because we are at the edge of the grid. If $j > 0$, we distinguish two cases: if $x_{i,j-1}$ is true, the result is, again, 0. If, however, $x_{i,j-1}$ is false, the result is 1 plus the number of shaded squares to the west of $(i, j - 1)$ (up to the first unshaded square, or to the edge of the gird). This gives a recursive formula that can be passed to Z3. For example, the number of shaded cells to the west of $(3, 2)$ is the value of the expression

```
If(X[3][1],
   IntVal(0),
   IntVal(1) + If(X[3][0],
                  IntVal(0),
                  IntVal(1) + IntVal(0)))
```

We have used `IntVal(0)` instead of a plain 0 to stress that this is an expression to be passed to Z3, because when we encode the puzzle, we don't know the values of the `X` variables.[7]

[7] Python would apply *coercion* (implicit type conversion) to a plain integer because it is added to a Z3 expression. Relying on coercion gives simpler code and causes Python to

An important consequence of this recursive approach is that the solver now has to deal with both Boolean and integer values. Therefore, for Canal View, we should use the general-purpose `Solver()` instead of `SolverFor('QF_FD')`.[8]

8 How Hard Are these Puzzles?

This section will make more sense once we have covered Section 3.3 of our textbook. On first reading, you may skim it, or even skip it.

We have discussed complexity classes for *decision* problems: those that have yes/no answers. We can derive decision problems from our puzzles by asking, "Does this puzzle have at least one solution?" Notice that we ask for "at least one" instead of "exactly one." These decision problems are the ones whose hardness we briefly discuss in Section 8.1. The uniqueness question, which is an important one for puzzle setters, is then the subject of Section 8.2.

8.1 Existence of a Solution

If we are given a purported solution to either a Smullyanic Dynasty puzzle or a Canal View puzzle, we can easily check whether it is indeed a solution. Verifying that each clue sees the right number of canal cells, for example, can be done in time linear in the size of the grid. (Do you see how?) For the connectedness checks, we can build a spanning tree of the canal cells in linear time. For Smullyanic Dynasty, similar checks apply. (The code described in Section 9 already verifies the solutions.)

All this means that both Smullyanic Dynasty and Canal View are in NP: a "yes" answer comes with a witness (namely, the solution to the puzzle) that can be checked in polynomial time.

Canal View has been shown to be NP-complete. The proofs of NP-hardness can be found in [5]. Figure 5 shows the puzzle that results from asking whether the graph of Figure 4 has a Hamiltonian cycle.

What does the NP-completeness of our puzzles mean for our project? Since no known algorithm for an NP-complete problem runs in polynomial time in the worst case, we should expect to come across puzzles that take an inordinate amount of time to solve. We should expect the transition from easy to very difficult to be abrupt, and we'll encounter puzzles of the same size that take very different amounts of time to solve. It is quite possible for a minor change in the way we write the constraints to make a large difference in the solution times of some puzzles. All this notwithstanding, most puzzles that are doable by humans are within the (easy) grasp of even a simple solver. Be prepared, but don't lose heart.

simplify $1 + 0$ to 1 before Z3 even sees the expression.

[8] We cannot use a solver for finite domains because Z3 cannot pre-determine an upper bound on the values of our `If` expressions. It is possible to overcome this limitation by introducing auxiliary variables to which an explicit upper bound is attached, but, to keep things simple, we won't pursue this idea in our project.

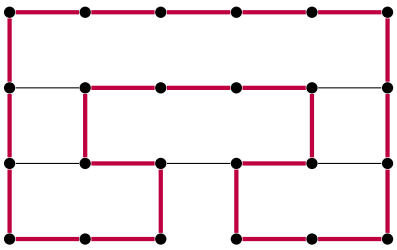


Figure 4: A grid graph with degree at most 3. (Every vertex is connected to at most three other vertices.) A *Hamiltonian cycle* is a closed path in the graph that visits each vertex exactly once. The only such path of this graph is highlighted.

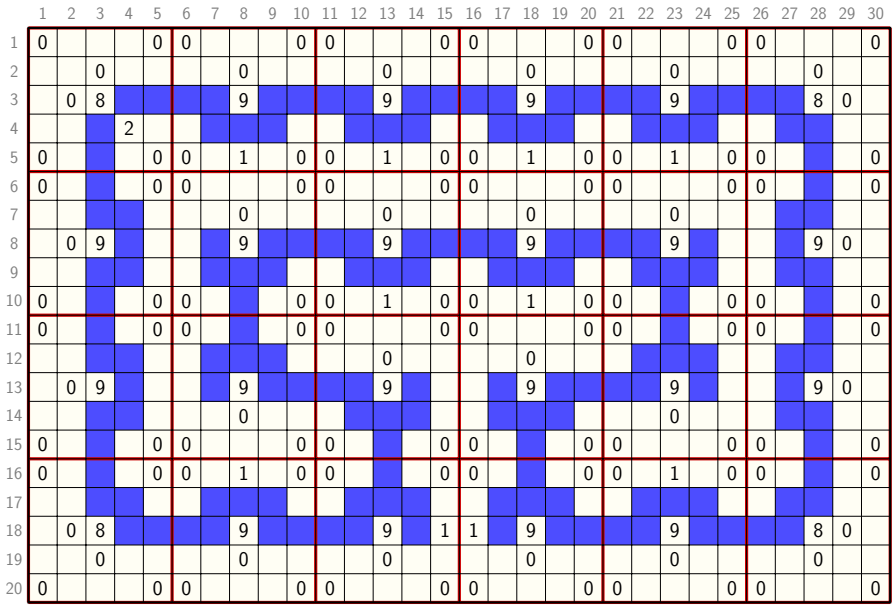


Figure 5: Reduction from Hamiltonian Cycle to Canal View applied to the grid graph of Figure 4. The 5×5 squares delimit the *gadgets* that are stitched together to encode the grid. Each gadget corresponds to one vertex of the grid graph. The Hamiltonian circuit found in this solution of the Canal View puzzle is the one highlighted in Figure 4.

Most people will agree that it is much easier to find a Hamiltonian cycle directly on the graph of Figure 4 than to solve the Canal View puzzle of Figure 5. In fact, anyone who is familiar with the definition of Hamiltonian circuit is likely to find one for the graph of Figure 4 in a matter of seconds. The greater difficulty of solving the puzzle is due, at least in part, to the fact that the reduction of Figure 5 produces, for a graph with n vertices, a grid with $25n$ cells. In proving NP-hardness, we only need to come up with a *polynomial* reduction scheme. A linear reduction like ours is certainly polynomial. Recall that reductions like this are not meant to provide practical ways to solve problems: they are only meant to prove that a polynomial algorithm for the candidate problem would yield polynomial algorithms for all problems in NP.

8.2 Uniqueness of the Solution

For most puzzles, including Smullyanic Dynasty and Canal View, the solution should be unique. We need to extend the basic framework of computational complexity, which talks about decision problems, if we want to talk about how hard it is to check whether a solution is unique. Unsurprisingly, this extension has been done.[9] The starting point is the definition of *function problems*, in which the result is a string instead of a yes/no answer. The function problem analog to SAT asks for a satisfying assignment to a propositional formula to be returned, instead of "yes." Among function problems, we are particularly interested in *counting problems*, in which the answer is the number of solutions to a function problem. The counting analog of SAT is #SAT, which asks the number of satisfying assignments of a propositional formula.

Function problems have their complexity classes. A function problem is in FNP if there is a polynomial-time, deterministic algorithm that, given x and y, checks whether y is a possible output for x. For counting problems, the analog of FNP is the $\#P$ class. If a decision problem is hard, the corresponding counting problem is also hard—because knowing the number of solutions means also knowing whether there are solutions. The converse is not true. There are easy decision problems that give rise to hard counting problems. For example, 2SAT is in P, but #2SAT is #P-complete.

Fix a decision problem, say, SAT. Checking whether a SAT instance has a unique solution is at least as hard as checking whether it has at least one solution: if we know that the instance has exactly one solution, we also know that it has at least one. Checking whether a SAT instance has exactly one solution is no harder than counting its solutions: if we have counted the solutions, we know whether their number is 1.

To make things more "interesting," though, the problem we are concerned with—whether a puzzle has a unique solution—is not exactly any of the problems discussed so far. Why? Because the question we want to answer is whether, given that a puzzle has a solution, it also has another one.

[9] Reading this section is not necessary to complete the project, but will improve your understanding of the big picture. Be warned that we are skipping over much fine print here.

Why does the knowledge that there is a solution affect the difficulty of checking whether there is another solution? Consider graph coloring, which is NP-complete when the number of colors is at least 3. Suppose, however, that we have computed one k-coloring of a graph. Then, we can cheaply obtain $k! - 1$ other k-colorings by permuting the colors.

So, a way to put the question of uniqueness is this: For puzzle type X, given a solution to a specific puzzle, is there a cheap way to determine whether there is another solution for the same puzzle that is applicable to all puzzles of type X? (X could be Canal View, for example.) Since our preferred method of checking for additional solutions involves calling the SAT solver again, it doesn't seem that we believe that such a cheap way exists for Canal View puzzles. In fact, we don't (unless P = NP). In the remainder of this section, we provide justification for our belief.

First, let's give a name to the problem we want to discuss. For a puzzle type P (e.g., Canal View) we denote by ASP P the problem of deciding whether, given an instance of P (e.g., a Slitherlink puzzle) and a solution to that instance, there exists another solution.[10] (ASP stands for *Another Solution Problem* [6]). ASP P, as we defined it, is a decision problem, which, for Canal View, has been proved NP-complete.

For some problems P, there are direct proofs that ASP P is NP-complete. As in the practice of NP-completeness, though, once a problem ASP P' is known to be NP-complete, we can try to reduce it to another problem ASP P to prove the latter is NP-complete. The key ingredient of such an approach is a *parsimonious reduction*, that is, a reduction that provides a bijection between the solutions to the two problems. This way, if we have an instance of P' and a solution to it, we can map those to an instance of P and one of its solutions. If we find that the instance of P has no other solution, since the solutions of the two problems are in one-to-one correspondence, we know that the instance of P' does not have another solution either. We need both the reduction from P' to P and the mapping from the solutions of P' to the solutions of P to be computable in polynomial time. Then, we can argue that, if we had a polynomial algorithm to solve ASP P, we would have a polynomial algorithm to solve ASP P'.

Parsimonious reductions have been used to prove that counting problems are #P-complete. Their role in the ASP context underlies the connection between these problems. The good news for those who work with another-solution problems is that many reductions used to prove NP-completeness results are parsimonious. Based on parsimonious reductions, Yato [7] has introduced the notion of ASP-complete problem, while [5] has proved that Canal View and many other puzzles are ASP-complete. If P is ASP-complete, counting its solutions is #P-complete and its decision version is NP-complete.

[10] While we limit our discussion to puzzles, ASP applies to function problems in general.

9 The Project

The goal of this project is to implement two Z3-based solvers: one for Smullyanic Dynasty and one for Canal View. The experience you gain with the Smullyanic Dynasty solver will help you tackle the Canal View solver, which, as we have seen, is a bit more challenging. Each solver consists of five files. A "starter kit" is available on Canvas. It contains, for each puzzle type:

- A file (`smullinputs.py` for Smullyanic Dynasty and `canalinputs.py` for Canal View) containing a function called `get_grid` that takes an integer i as input and returns the i-th puzzle from a predefined collection. A Smullyanic Dynasty puzzle is just a sequence of sequences of "optional" integers. (That is, each value is either an integer or `None`.) In a Canal View puzzle, each entry of the sequence of sequences is either -2 (empty square) -1 (clue of unknown value) or a nonnegative integer (a clue of known value). This file also contains a function called `check_grid` to check the integrity of a puzzle grid. Finally, it contains a function called `parse_command`, which reads the command line.

- A file (`smulldisplay.py` for Smullyanic Dynasty and `canaldisplay.py` for Canal View) containing implementations of the Unicode-based and `matplotlib`-based display functions.

- A file (`smullverify.py` for Smullyanic Dynasty and `canalverify.py` for Canal View) that contains a function called `verify` that takes a grid and a purported solution and checks whether the latter is indeed a solution to the puzzle described by the former. The function has an optional Boolean parameter (`allow_weak`) that makes it skip the connection check, so that you can use it already for the second milestone of the project. Your code must call `verify` for each solution it finds.

- A file (`smull.py` for Smullyanic Dynasty and `canal.py` for Canal View) containing a skeleton of the solver. This file is where most of your implementation work will take place. In particular, you will have to define variables, add constraints to the solver (in function `add_constraints`) and implement the `solve_and_print` function.

- A file (`smullrun.py` for Smullyanic Dynasty and `canalrun.py` for Canal View) to run your solver on all provided puzzles in one batch and collect the results in one file. The file has various options to control the experiment. Use the `-h` option to find out. The `-v` option, in particular, can be used to track progress.

The two solvers should take arguments from the command line that specify the following:

- `-p n` to request solution of the n-th puzzle (from the collection in the "inputs" file). The first puzzle is number 0.

- `-m` to request `matplotlib` output instead of Unicode.
- `-f n` to change the font size used by `matplotlib`.
- `-d` to request that the solver draw the unsolved puzzle (in the format determined by the `-m` switch) and quit.
- `-a` to request that the displayed solution include also the values of the p variables. (See Figure 2.) This option only takes effect when the `-m` option is also specified.
- `-s n` to request that at most n solutions be computed. By default, the solver should find all solutions and report their count.
- `-v` to control the amount of output produced by the solver. This option may be repeated (`-v -v` or `-vv`) to increase the verbosity of your program. It is up to you to decide what gets printed at each verbosity level. You should use this option to make the debugging of your programs more convenient.
- `-r` to cause the solver to report Z3's statistics.
- `-h` to cause the solver to print a short help message.

The parsing of the options and the (automated) generation of the help message is already available through `parse_command`. The functionality requested with those switches, of course, needs to be implemented.

10 Implementation Issues

Throughout this section, we assume that the solver object is referenced through a Python variable named `slv`.

10.1 Interacting with the Z3 Solver Object

Since all Z3 variables are Boolean, `slv = SolverFor('QF_FD')` is the recommended choice for Smullyanic Dynasty. As previously discussed, `slv = Solver()` is the simpler choice for Canal View.

When the solver has determined that the constraints are satisfiable, we can access the satisfying assignment by the `model()` method of the `Solver` class. We have to be careful, though, because Z3 may assign no value to certain variables. The event is pretty rare because, for a variable not to receive a value, there must be two equally valid solutions that only differ in the value of that variable. We still need to do something about it. Fortunately, Z3 can be asked to do *model completion*. Suppose you inquire the value of the Boolean variable `x` that did not get a value during the satisfiability check. Then `slv.model()[x]` returns `None`. Calling `slv.model().eval(x)`, however, returns `x`. Finally,

```
slv.model().eval(x, model_completion=True)
```

```
And(True, a)                  ->  a
And(False, a)                 ->  False
And(And(a, b), c)             ->  And(a, b, c)
Or(a)                         ->  a
And()                         ->  True
Or()                          ->  False
Not(Not(a))                   ->  a
Or(a, a)                      ->  a
And(a, Not(a))                ->  False
If(a, b, b)                   ->  b
If(a, True, False)            ->  a
If(a, False, True)            ->  Not(a)
Implies(a, a)                 ->  True
Implies(Not(a), a)            ->  a
Xor(a, Not(a))                ->  True
a == True                     ->  a
(a == a) == b                 ->  b
And(Or(a,b), Not(Or(b,a)))    ->  False
PbEq(((a,1),(b,1),(c,1)),0)   ->  Not(Or(a, b, c))
AtLeast(a, b, c, 1)           ->  Or(a, b, c)
AtLeast(a, b, c, 3)           ->  And(a, b, c)
AtLeast(a, b, True, 2)        ->  Or(a, b)
AtLeast(a, b, False, 2)       ->  And(a, b)
AtMost(a, b, c, 1)            ->  AtLeast(Not(a),Not(b),Not(c),2)

PbEq(((a,1),(b,1),(c,1)),1)   ->  PbEq(((a,1),(b,1),(c,1)),1)
a == (a == b)                 ->  a == (a == b)
And(a, Or(a, b))              ->  And(a, Or(a, b))
And(a, Or(Not(a), b))         ->  And(a, Or(b, Not(a)))
And(Or(a,b), Or(a,Not(b)))    ->  And(Or(a,b), Or(a,Not(b)))
```

Figure 6: Examples of results produced by `simplify`. Z3 applies a combination of constant propagation, rewriting in standard form, and simple heuristics. The last few lines of the table above show that `simplify` does not really do Boolean reasoning. Hence, it does not realize that `And(a,Or(a,b))` is equivalent to `a`. Of course, these limitations only apply to `simplify`. If we ask Z3 whether the constraint `And(a,Or(a,b))!= a` is satisfiable, the answer is "no."

will cause Z3 to add an arbitrary value for `x` to the model and return it. This is exactly what we want. It also leaves the model in the right state for when we later add a blocking clause to count the number of solutions. This approach is already implemented (for both puzzles) in the provided function `evaluate_model`.

If, for whatever reason, `slv.check()` returns `unknown`, a call to

```
slv.reason_unknown()
```

will return an explanation string, which can be printed. It may be a little cryptic, but it's usually better than nothing.

Finally, it may be convenient to call `simplify` on constraints before adding them to the solver. The simplification only applies simple rules. (See Figure 6 for a sample of what it can and cannot do.) However, if you call `print(slv)`,

perhaps to check whether the constraints you've added are the ones you meant, the simplified forms may be easier to read.

10.2 Visualizing the Solution

The ability to visualize the solutions to the puzzles is important, not only for the finished program, but also for the early "drafts." Images like the ones of Figure 2 are reasonably easy to understand. Good luck, however, interpreting the output of `print(slv.model())` without significant extra work. It's safe to say that carefully planning the visualization of the puzzle solutions is key to the success of your project.

Python has a powerful visualization library that some of you may have already used: `matplotlib`. Matplotlib does not ship with the standard Python distribution, but it can be installed easily. With its help one can draw the puzzle as given, the solved puzzle, or the puzzle with the arrows colored according to the values of the p variables. The code for these tasks is provided to you.

It is also possible to print a solution to the terminal window in which the program is running. Using Unicode characters[11] results in output that is easier to read than plain "ASCII art." This "low-budget" solution makes it easy to run a batch of puzzles, while saving the solutions in one text file.

In other respects, the Unicode solution is less satisfactory. For example, adding the values of the p variables to a diagram made with Unicode characters turns it into a mess.

In case you are wondering, the puzzle figures in this document were generated by the solvers in LaTeX format, using the Ti*k*Z package. This type of output is not required in your project: while the quality is the highest, you would need to install LaTeX and learn how to use it. Feel free to inquire if you are interested.

10.3 Debugging Your Project

Even before you run your program, you should consider using Python's typing hints in your code so that a type checker for Python like `mypy` may detect some problems. Programs annotated with type hints are easier to read, especially when you return to them after a hiatus. The annotations take some time to enter, but they also help you to answer the question, "What did I have in mind when I wrote this?"

Most Python programming environments provide a symbolic debugger—for example, an interface to `pbd`, the Python debugger— which allows you to step through the execution of your program, set breakpoints, and inspect variables. You may want to know how to run that debugger, even though you may not use it often.

Finally, for any program of reasonable complexity, the author(s) should plan how to make debugging easier from the beginning. We already discussed the importance of good visualization of the solution in Section 10.2. It is also a good

[11] https://en.wikipedia.org/wiki/List_of_Unicode_characters.

idea to have an easy way (e.g., from the command line) to control the amount of detail printed by your program. In standard mode, your program should only print the solution and the time taken, but when requested to be more verbose, it should report on whatever else you think may help you with the debugging. This is what the `-v` option of `parse_command` is for.

When debugging small examples, it may make sense to print the constraints added to the solver. This can be done in two ways. The simpler is

```
print(slv)
```

The limitation of this approach is that, if there are many constraints—or assertions, as they are called in Z3—it will not print all of them: at some point, it will print three dots (i.e., an ellipsis) and give up. So, if you want to see all the contents of the solver, you need to iterate through the assertions yourself. For example:

```
for assertion in slv.assertions():
    print(assertion)
```

10.4 Gathering Statistics

The `Solver` class has a `statistics()` member function that will report various quantities describing the behavior of the solver. Not all those quantities are meaningful to someone without detailed knowledge of Z3's functioning. However, you will quickly realize that a long run involves many *decisions* and *backjumps*, among other things.

The CPU time reported by `print(slv.statistics())` is the time taken by the last call to the `check()` method, not the cumulative CPU time spent by the solver. Besides, a significant fraction of the CPU time is spent in communicating the constraints to the solver.[12] Therefore, it is a good idea to measure the CPU time spent by your program from start to finish. Python's `time` package supplies all the necessary functions. Their use is illustrated by the skeleton solvers that you download as part of the Starter Kit.

11 Points and Milestones

There are four submissions. After the third and fourth submission, there will be code interviews. The first interview will cover the second and third submission. After the second submission, you will receive feedback, but the grade will be awarded after the interview. The whole project is worth 100 points. Each of the four submissions is worth full points if the code is fully functional and properly

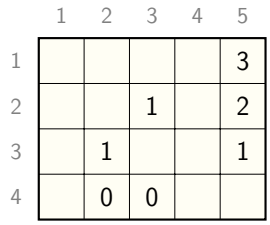
[12] Z3 carries out extensive—and relatively expensive—checks on those constraints before accepting them. Encoding our puzzles may produce many constraints. Hence, the time required to call `slv.add`, may no longer be negligible, as it was in the Z3-based homework problems.

documented, and the interview shows that you understand your code well. All files are submitted through Canvas.

- **September 10: 10 points.** This first stage of the project is essentially a sanity check. Add the puzzle below to the Smullyanic Dynasty collection (in `smullinputs.py`) as the last puzzle of the collection and make sure that you can display the grid to be solved by running

  ```
  python3 smulldisplay.py -pN
  ```

  where `N` is the index of the new puzzle. Do this with and without the `-m` option. (The file contains a small test program so that you can run it by itself with your Python interpreter.)

  You will need to modify `parse_command` to increase the maximum puzzle index. (On the contrary, `smullrun.py` is already set up to also run the puzzle you add.) Submit your modified `smullinputs.py` and the `.png` file you get by saving the `matplotlib` figure.

- **October 8: 30 points.** Submit a preliminary version of the Smullyanic Dynasty solver (`smull.py`). This preliminary version only needs to satisfy the constraints on the x variables. That is, no symbol appears more than once unshaded in any row or column, and shaded squares cannot share an edge. The shaded cells, however, do not need to form one connected region. The solutions obtained this way are sometimes called *weak solutions*. Your solver should accept all options described in Section 9, except for `-a`. All examples in `smullinputs.py` should be doable in a few seconds.

  Most puzzles have multiple weak solutions. Since your program should (in principle) be able to find all of them, the `-s` option will be very useful when the number of weak solutions is very large.

- **November 5: 25 points.** Submit your completed `smull.py`. Your program should implement all the constraints described in Section 5 and all the options described in Section 9. It should be able to solve all the puzzles in `smullinputs.py` in less than a minute each. This is a generous allowance: Your solver will likely solve most puzzles in less than one second. If it takes more than one minute to generate a solution, something is amiss, even if you run the experiments on a slow machine. Your solver should produce all solutions to the given puzzle unless the `-s` option is

specified. (Most puzzles have just one solution, but some have either 0 or more than 1.)

- **November 19: 35 points.** Submit your completed Canal View solver (`canal.py`). This final submission should feature a complete solver, with ability to display the values of the p variables when the `-a` option is specified and, in general, support for all the options described in Section 9.

The two finished solvers need about 2700 lines of Python. The starter kit gives you about 2000 of those lines (many of them used to describe the puzzles). Hence, expect to write, between September and November, about 700 lines of Python. That is not an awfully large number, but don't underestimate the task.

References

[1] T. Brock-Nannestad. Space-efficient planar acyclicity constraints - a declarative pearl. In *Functional and Logic Programming (FLOPS)*, pages 94–108, Mar. 2016.

[2] T. Brock-Nannestad. Space-efficient acyclicity constraints: A declarative pearl. *Science of Computer Programming*, 164:66–81, Oct. 2018.

[3] R. A. Hearn and E. D. Demaine. *Games, Puzzles, and Computation.* A K Peters, 2009.

[4] D. E. Knuth. *The Art of Computer Programming*, volume 4B. Addison Wesley, 2023.

[5] MIT Hardness Group, J. Brunner, L. Chung, E. D. Demaine, D. Hendrickson, and A. Tockman. ASP-completeness of Hamiltonicity in grid graphs with applications to loop puzzles. In *Fun with Algorithms*, pages 23:1–23:20, 2024.

[6] N. Ueda and T. Nagao. NP-completeness results for NONOGRAM via parsimonious reductions. Technical Report TR96-0008 may, Department of Computer Science, Tokyo Institute of Technology, 1996.

[7] T. Yato. Complexity and completeness of finding another solution and its application to puzzles. Master's thesis, University of Tokyo, Graduate School of Science, 2003.