Applied Logic Weekly Assignment Report

Bohdan Tymofieienko B.S.
b.tymofieienko@student.fontys.nl
4132645

Mikolaj Hilgert M.H.
m.hilgert@student.fontys.nl
4158385

December 20, 2021

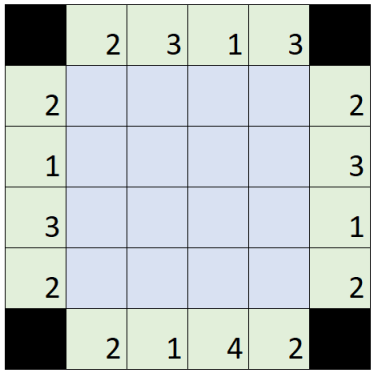
Contents

1 Week iii. Skyscraper and 3 water jug problem.

1.1 Skyscraper

A Skyscraper puzzle consists of a square grid with some exterior 'skyscraper' clues. Every square in the grid must be filled with a digit from 1 to 4 (or 1 to whatever size the grid is) so that every row and column contains one of each digit. To represent this problem in *Z3*, we must set out the grid and describe the objective in terms of constraints, Each section will describe a constraint, following by implementation and explanation.

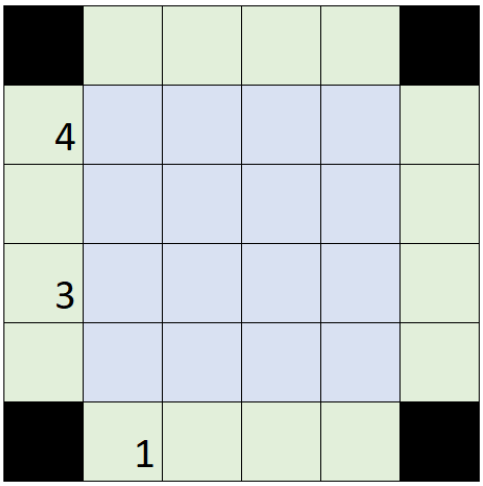
For our approach to the problem, we create a grid that holds an inner grid. For the sake of brevity a 4x4 skyscraper example will be explained. In this case, the inner grid in-fact is 4x4, however the encapsulating grid is in fact 6x6. As it needs to specify the 'hint' that is used to solve the puzzle.



As can be seen from the above graphic. The black squares are ignored as they are not directly adjacent to any of the inner grids columns or rows. The light green squares are the values of the 'hints' which specify how many skyscrapers can be seen from that square. Each green square specifies the amount to either an adjacent row or column. The vertically placed green squares specify the rows. And the horizontal specify the blue column.

The puzzle may increase in difficulty, it's possible that not all hints are given as in the table above. But rather only a few strategic hints are given, as a result the *Z3* program has to be able to solve the puzzle without being given all the hints, as well as the program should be easily expandable from a 4x4 inner grid upwards. This is something that is discussed later.

For the purpose of explanation of all elements, A hard difficulty 4x4 grid will be explained.



Firstly, we must define the board function, inner and outer grid sizes and hints.

```
(declare-fun B (Int Int) Int)

(declare-const Grid Int)
(declare-const Size Int)

;Assert hints
(assert (= (B 1 5) 4))
(assert (= (B 1 3) 3))
(assert (= (B 1 2) 1))

;Input inner-grid size
(assert (= Grid 4))

;Assert outer-board size
(assert (= Size (+ Grid 2)))
```

The first constraint that has to be set, is that of the possible heights of the skyscrapers that can be placed within the inner grid.

```
(assert
    (forall ((row Int) (col Int))
        (ite
            (and
                (< 1 row Size)
                (< 1 col Size)
            )

            (<= 1 (B row col) Grid)

            (ite
                (<= 1 (B row col) Grid)
                true
                (= (B row col) 0)
            )
        )
    )
)
```

What is defined that, for all rows and columns, within the entire grid, the values must be between 1 and the inner grid size. In this case 4. As per: **(< = 1 (B row col) Grid)**.
Another fact is that if say there is no value given to the hint cells, then that hint is set to 0, so that *Z3* does not set values that would cause a fallacy. This ensures that *Z3* uses only the correct hints to find the values so that they satisfy the model. This is seen in the ITE statement, where if the row and column fall within the inner grid, then they may be of values [1,4]. Else if its in the outer grid, another ITE checks whether a value has already been assigned (in the case of hints), if it has, its kept as it was. If not, its set to 0 so it does not contradict later conditions.

The next important condition to define is that all rows must be distinct from one another. This is done by passing applying the following logic, for a column any two rows must be distinct. Applying this within a **forall** ensures that this applies for all columns. Implication in this case states that next statement applies only to entries within the bounds.

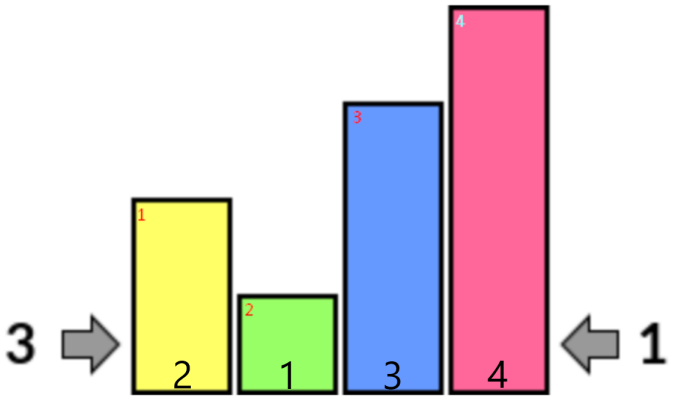
```
(assert
    (forall ((col Int) (row1 Int) (row2 Int))
        (=>
            (and
                (< 1 col  Size)
                (< 1 row1 Size)
                (< 1 row2 Size)
                (distinct row1 row2)
            )
            (distinct (B col row1) (B col row2))
        )
    )
)
```

This same flow is applied to ensure that for all rows no two columns are repeated.

```
(assert
    (forall ((row Int) (col1 Int) (col2 Int))
        (=>
            (and
                (< 1 row  Size)
                (< 1 col1 Size)
                (< 1 col2 Size)
                (distinct col1 col2)
            )
            (distinct (B col1 row) (B col2 row))
        )
    )
)
```

The final condition that has to be defined is the one of skyscraper visibility. A hint of value 1 means only one skyscraper is visible when viewing the row or column from the direction of the clue. Similarly, if a hint is 2, that means from that position only two can be seen, and so on. This does **not** mean however, that the number coincides with the amount of sky scrapers in the row/column as higher skyscrapers block the view of lower skyscrapers located behind them.

There are many approaches to defining this rule in *Z3*. The aforementioned example is not ideal for explaining as all hints define a row or column only from one side. So the approach is explained with the following example of a row:



As depicted above, the hint on the left is of value 3 meaning from that position it may only see 3 different skyscrapers, whilst from the right looking left you may just see one as that one is the tallest. As mentioned before, from left to right you do not see four as the tower at position 1 (depicted by the small numbers) is taller than the one at the second position and thus its 'hidden' from view.
To turn this into a condition, a function is defined that has takes a parameter of two adjacent towers and discerns which one is larger and returns that value.

```
(define-fun FindTaller ((a Int) (b Int)) Int (ite (> a b) a b))
```

This is then used when we define a function that checks the visibility in a row.

```
(define-fun CheckHeightOfRow ((row Int) (hint Int) (e1 Int) (e2 Int)
(e3 Int) (e4 Int)) Bool
    (=>
        (<= 1 (B hint row) Grid)
        (=
            (+
                1
                (ite (> (B e2 row) (B e1 row)) 1 0)

                (ite (> (B e3 row) (FindTaller (B e2 row)
                (B e1 row))) 1 0)

                (ite (> (B e4 row) (FindTaller (B e3 row)
                (FindTaller (B e2 row) (B e1 row)))) 1 0)
            )
        (B hint row)
        )
)
```

The above function works as such, a row number is input as well as the corresponding hint and the potential elements in said row. Then we ensure that the hint falls within the bounds of the board. Also, it assures that each hint is equal to the amount of elements that satisfy the visibility condition. Visibility depends on whether the tower is larger than the maximum of all predeceasing towers.

This same operation is done for column, except the arrangement of elements in the grid is different:

```
(define-fun CheckHeightOfCol ((col Int) (hint Int) (e1 Int) (e2 Int)
(e3 Int) (e4 Int)) Bool
    (=>
        (<= 1 (B col hint) Grid)
        (=
            (+
                1
                (ite (> (B col e2) (B col e1)) 1 0)

                (ite (> (B col e3) (FindTaller (B col e2)
                (B col e1))) 1 0)
                ...
            )
        (B col hint)
```

Having the functions, an assertion must be made for all rows. Meaning that the aforementioned functions apply for the whole grid. For each row the values are ascending and descending in order to provide order as for example a hint on the left row applies to the next 4 towers to the right of the hint. And on the same row the hint on the right side applies to the 4 towers to its left. As defined in our grid. The same is done for columns.

```
(assert
    (forall ((index Int))
        (=>
            (< 1 index Size)
            (and
                (CheckHeightOfRow index 1 2 3 4 5)
                (CheckHeightOfRow index 6 5 4 3 2)
                (CheckHeightOfCol index 1 2 3 4 5)
                (CheckHeightOfCol index 6 5 4 3 2)
            )
        )
    )
)
```

1.1.1 Conclusion

To then get values of the table, we must check satisfiability and get values of the inner grid. $Z3$ provides an output.

4	1	2	3	4	
	3	4	1	2	
3	2	3	4	1	
	4	1	2	3	
	1				

Finally, the point of extendability. This solution is easily extendable to larger sizes. All that has to be done is Grid const has to be defined to say 5. Which results in a 5x5 inner graph. The next thing that must be changed is that for the two height checking functions another parameter for the new element and ITE must be added where you include the added tower in the comparisons. As well as in its corespondent assertion the new elements has to be added. As present in the 5x5 example.

NOTE: The full examples for 4x4 and 5x5 are included in the submission. The output could be interpreted in a table by writing values from the bottom upwards, per column.

1.2 Three jugs puzzle

Water pouring puzzle (*or three jugs puzzle*) is a famous problem. In essence this is a class of puzzles where the target is to evenly split the water between two jugs. In our particular case the rules are as follows:

1. There are jugs 8, 5 and 3 liters each respectively.
2. First jug is full, others two are empty.
3. Each step consists of pouring water from one jug to another and stops when either the source jug is empty or the destination jug has become full.

Target is to have 4 liters in first jug, four liters in second jug and third jug is empty. In other words, water is split evenly between first and second jugs.

In order to solve this problem with $Z3$ we will first deduce a set of constraints from the original rules. Before that, to keep track of jug volume in each step let's define a 2-dimensional array of integer numbers where first index is a jug's number and second index is a step. It could be also interpreted as a table.

	First jug (8L)	Second jug (5L)	Third jug (3L)
1	8	0	0
...	...	...	...
n	4	4	0

Table 1: Desired input (first step) and desired output (step n)

List of constraints can be formulated in the following way:

1. **Each column has a maximum value.** (*Volume of a jug*)
2. **There exist no row where value in a row exceeds column's maximum.** (*Meaning each column has a maximum, and all the values in this column are less or equal than the maximum value.*)
3. **There exist no row where number of coinciding elements is equal to amount of columns.** (*Two rows cannot be the same. Thus, loops are avoided.*)
4. **For all rows sum of entries in a row is equal to 8.** (*Total volume of water in jugs stays the same. No water is spilled.*)
5. **In first row where value of entry one is 8 , entry two is 0, entry three is 0.** (*First jug is full, other two are empty. Desired input.*)
6. **There exist a row where value of entry one is 4, entry two is 4, entry three is 0.** (*First and second jugs have 4 litres each, third jug is empty. Desired output.*)
7. **For all two *adjacent* rows exactly *two* values are changed and exactly *one* remains unchanged.** (*Definition of a step. Since it's only possible to pour from exactly one jug to exactly one jug we have to add this constraint.*)

8. **For two *adjacent* rows R_1 and R_2 the sum of changed entries of R_1 equals to sum of changed entries of R_1 and either one of changed entries of R_2 is the maximum of a column or a 0.** (*Total amount of water after pouring is the same. Thus we need to assure if sums are equal. Pouring stops either if destination is full (maximum of a column) or origin is empty (zero)*)

1. To assure each column has an upper bound (volume), we will declare three constants and assign values.

```
(assert (= Jug1_Max 8))
(assert (= Jug2_Max 5))
(assert (= Jug3_Max 3))
```

2. To assure that jug cannot contain more than it's volume we add next statement. Implication states that following statement only applies to steps within the bounds.
Remark: 'MaxStep' is the step with permutation [4,4,0]

```
(assert (forall ((step Int))
        (=>
        (<= 1 step MaxStep)
        (and
            (<= 0 (B 1 step) Jug1_Max)
            (<= 0 (B 2 step) Jug2_Max)
            (<= 0 (B 3 step) Jug3_Max)))))
```

3. To avoid repeating rows and as a consequence looping operations, we compare rows and negate the existential quantifier. Implication verifies that steps are within the bounds and not the same, then statement would fail.

```
(assert
    (not
        (exists ((step1 Int) (step2 Int))
            (and
                (<= 1 step1 MaxStep)
                (<= 1 step2 MaxStep)
                (distinct step1 step2)
                (and
                (= (B 1 step1) (B 1 step2))
                (= (B 2 step1) (B 2 step2))
                (= (B 3 step1) (B 3 step2)))))))
```

4. Since volume of water in the jugs is the same for all the steps, we have to add following statement.

```
(assert
    (not
        (exists ((step Int))
            (and
                (<= 1 step MaxStep)
                (not
                    (= (+ (B 1 step) (B 2 step) (B 3 step)) 8))
            ))))
```

5. Additionally we have to arrange the input. In the step 1 contains of jug are [8,0,0].

```
(assert (= (B 1 1) 8))
(assert (= (B 2 1) 0))
(assert (= (B 3 1) 0))
```

6. Similarly to previous one we have to define that at some point jugs are arranged in a way [4,4,0], or in other words our target point.

```
(assert
    (exists ((step Int))
        (and
            (<= 1 step MaxStep)
            (= (B 1 step) 4)
            (= (B 2 step) 4)
            (= (B 3 step) 0)
            (= step MaxStep)
        )
    )
)
```

7. To assure that exactly one entry remains unchanged in each of the two adjacent rows we count the amount of coinciding elements and compare with 1.

```
(assert
    (forall ((step Int))
        (=>
            (<= 1 step (- MaxStep 1))
            (= (+
                    (ite (= (B 1 step) (B 1 (+ step 1))) 1 0)
                    (ite (= (B 2 step) (B 2 (+ step 1))) 1 0)
                    (ite (= (B 3 step) (B 3 (+ step 1))) 1 0))
             1))))
```

8. Finally, the most important constraint takes place. In essence pouring water from one jug to another can be interpreted as a relationship between two rows in a column *(i.e.* $Step_1 = [8, 0, 0]$ $Step_2 = [3, 5, 0]$ *means we poured 5 liters from* $Step_1$ *to* $Step_2$*.)* To assure that operation is valid we have to check if the sums of **changed** elements are equal, so that $8 + 0 = 3 + 5$. Please note: we didn't mention 0 because it is **unchanged** in this step and thus we do not include it in the sum check. Additionally we need to add restrictions on amount of water poured in a step. We know that operation is valid either if source is empty or if destination is full. Consequently we need to assure that **changed** elements of $S2$ (result of pouring) are either 0 either maximum of a column they are placed.

As can be seen below we verify the sum and that one (or multiple) of the **OR** conditions is satisfied. Particularly either one of the jugs is empty or if one of the jugs is the maximum.

```
(define-fun PouredBetweenFirstSecond ((step Int)) Bool
    (ite
        (= (B 3 step) (B 3 (+ step 1)))
        (and (= (+ (B 1 step) (B 2 step)) (+ (B 1 (+ step 1)) (B 2 (+ step 1))))
        (or
            (= (B 1 (+ step 1)) 0) (= (B 1 (+ step 1)) Jug1_Max)
            (= (B 2 (+ step 1)) 0) (= (B 2 (+ step 1)) Jug2_Max))) false))
```

There are three possible scenario, pouring between jugs 1 and 2, between jugs 1 and 3 and between jugs 2 and 3.

```
(assert
    (forall ((step Int))
        (=>
            (<= 1 step (- MaxStep 1))
            (or
                (PouredBetweenSecondThird step)
                (PouredBetweenFirstThird step)
                (PouredBetweenFirstSecond step)))))
```

1.2.1 Conclusion

As a result we can see that $Z3$ gives a solution in 12 steps. We can interpret it as a table.

	First jug (8L)	Second jug (5L)	Third jug (3L)
1	8	0	0
2	3	5	0
3	3	2	3
4	0	5	3
5	5	0	3
6	5	3	0
7	2	3	3
8	2	5	1
9	7	0	1
10	7	1	0
11	4	1	3
12	4	4	0

Table 2: Solution in 12 steps

In general, by limiting 'Max Step' we can make $Z3$ solve it in less than 12 steps. The minimum possible amount of steps from our observations is 8.

```
(assert (<= MaxStep 8))
```

	First jug (8L)	Second jug (5L)	Third jug (3L)
1	8	0	0
2	3	5	0
3	3	2	3
4	6	2	0
5	6	0	2
6	1	5	2
7	1	4	3
8	4	4	0

Table 3: Solution in 8 steps