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

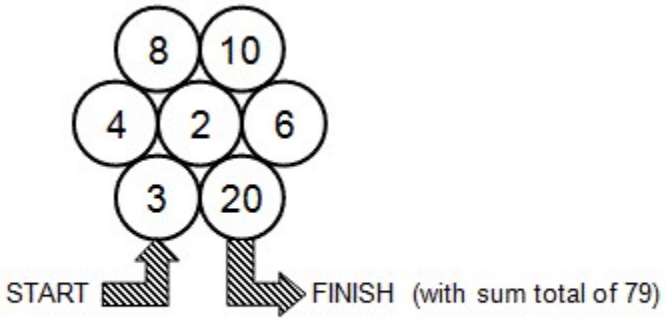
January 16, 2022

Contents

1 Week v. Prime circles problem Cannibals and Missionaries problem

1.1 Prime circles problem

In the provided challenge the target is to travel from "Start" to "Finish" having the accumulating sum always being prime.



Sum Totals = **3** _ _ _ _ _ _ _ _ _ _ **79**

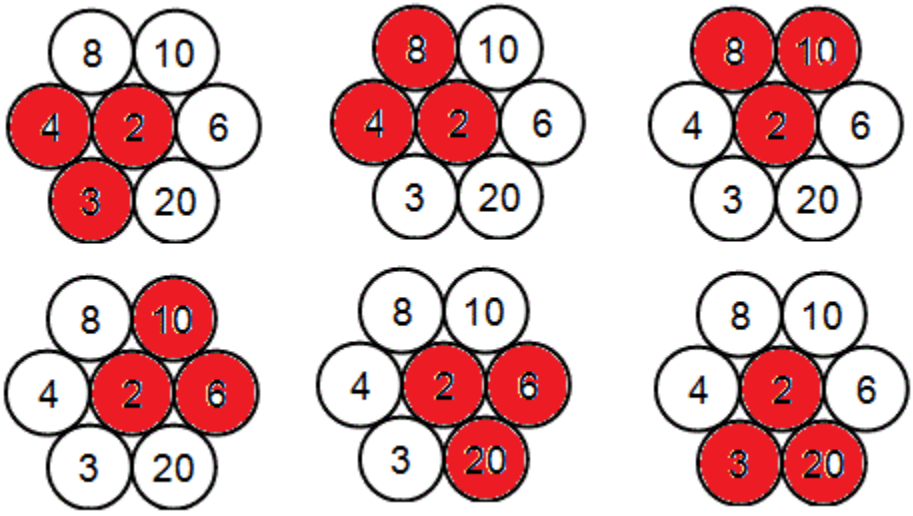
The set of rules is the following:

1. **The starting circle is 2.**
2. **The ending circle is 79.**
3. **Transitions between two circles are allowed provided two circles are adjacent.**
4. **The result of the transition is adding the value of the target circle to the accumulating sum.**
5. **For all the states accumulating sum is a prime number**
6. **'Ping-ponging' between circles (i.e., returning immediately to the circle that you have just left) is not allowed.**

To solve this problem with $Z3$, in fact, we need to have two observation at each period of time: accumulating sum and current circle. Knowing that only traveling between adjacent circles is allowed we have to specify that with $Z3$ explicitly. Before that, let's introduce the way to define the adjacent circles.

One way to do that would be to define an adjacency list. In essence, adjacency list is an set S of sets A where each A_x contains elements which are mutually transitive. Meaning it is possible to travel from any element of set A_x to any element of set A_x without breaking the rules.

As you can observe in the following diagram, there exists six adjacency lists with 3 elements each.



In $Z3$ we can define it as the following statement:

```
(declare-fun Adj (Int Int) Int)

(= (Adj 1 1) 2) (= (Adj 1 2) 3)  (= (Adj 1 3) 4)
(= (Adj 2 1) 2) (= (Adj 2 2) 8)  (= (Adj 2 3) 4)
(= (Adj 3 1) 2) (= (Adj 3 2) 8)  (= (Adj 3 3) 10)
(= (Adj 4 1) 2) (= (Adj 4 2) 10) (= (Adj 4 3) 6)
(= (Adj 5 1) 2) (= (Adj 5 2) 20) (= (Adj 5 3) 6)
(= (Adj 6 1) 2) (= (Adj 6 2) 3)  (= (Adj 6 3) 20)
```

To keep track of observations at each period of time we define the following two-dimensional array. Where first index is a **Step** and the second index is the **Level** of observation. Level 1 is the accumulative sum and Level 2 is a current circle.

```
(declare-fun Result (Int Int) Int)
```

The resulting table might be interpreted as next

Step	**1**	...	...	...	...	...	...	...	...	...	...	**N**
Sum	3	...	...	...	...	...	...	...	...	...	...	79
Circle	3	...	...	...	...	...	...	...	...	...	...	20

Making the following assertion we assure that at each step *(except the last one)* two constraints apply:

1. **For each step there exists such a two elements in some adjacency set, so that one of them is equal to the predecessor element *(an origin)* and another one *(the target)* added to the accumulative sum results in a prime number .**

2. **For all two steps except first one the predecessor circle is not equal to the ancestor circle. So given current circle is a and predecessor circle was b the ancestor circle for a can be anything but b. This restricts "ping-ponging".**

```
(forall ((step Int))
    (=> (<= 1 step (- MaxStep 1))
        (and
            (exists ((adjSet Int) (originCircle Int) (targetCircle Int))
                (and
                    (<= 1 adjSet 6)
                    (<= 1 originCircle 3)
                    (<= 1 targetCircle 3)
                    (distinct originCircle targetCircle)

                    (= (Result step 2) (Adj adjSet originCircle))
                    (= (Result  (+ step 1) 2) (Adj adjSet targetCircle))

                    (= (Result (+ step 1) 1) (+ (Result step 1)
                    (Adj adjSet targetCircle)))

                    (isPrime (Result (+ step 1) 1))
                )
            )
            (=> (not (= step 1))
                (not (= (Result (- step 1) 2) (Result (+ step 1) 2)))))))
```

Function 'isPrime' declares that the number must be prime to suffice the model. It does so by essentially saying that there exist no two distinct factors for the number both greater than 1.

```
(define-fun isPrime ((Input Int)) (Bool)
    (and (> Input 0) (not (exists ((y Int) (x Int))
     (and (= Input (* x y)) (< x Input) (< y Input) (> x 1) (> y 1)))))
)
```

1.1.1 Conclusion

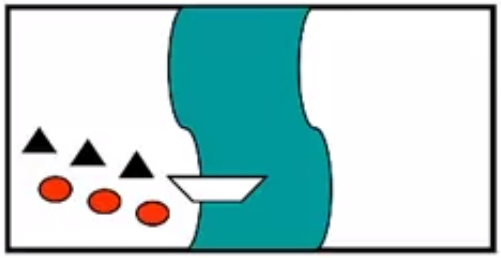
The resulting output can be interpreted as following:

Step	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Sum	3	5	13	17	19	29	37	41	43	53	59	79
Circle	3	2	8	4	2	10	8	4	2	10	6	20

As can be observed from the table, accumulative sum at each step is indeed a prime number. By looking at the second level *(Circle)* you may construct a path. Additionaly that there is no "ping-ponging" in the provided output. The last visited circle is 20 and the accumulative sum in the last step is equal to 79.

1.2 Cannibals and Missionaries problem

This problem is a variation of the famous Wolf, goat and cabbage problem. In this variation of the problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, that the missionaries present on the bank cannot be outnumbered by cannibals. The boat cannot cross the river by itself with no people on board.



The image above visually describes this problem, as we may interpret the red dots as the cannibals and the filled triangles as the missionaries. As can be seen all are on the left side, and the point is to transport all to the right side, without causing a 'tragedy'.

There are some important conditions that we have to explicitly establish:

1. Boat can carry upmost two passengers.
2. There may not be more cannibals than missionaries together at any time.
3. A boat must have at-least one passenger to cross sides.
4. A person/s may only cross if the boat is on their side.

To solve this problem, we decided the interpret the problem as a table, namely, per step we specify 3 missionaries and 3 cannibals. And their position is represented with either a 0 (meaning on the right side) and 1 (meaning the left side). The example of step 1 (initial step) is shown below... (s stands for step.)

	M			C		
s	1	2	3	4	5	6
1	0	0	0	0	0	0

With the tabular approach explained, we may start defining the problem within *Z3*. First, we define the board and ending constant;

```
;(step,person)->side
(declare-fun B (Int Int) Int)

; timepoint at which the required end-state is reached
(declare-const N Int)
```

The first condition to be defined is the one that the boat may only move upmost two passengers per move. This is defined via a forall within a large assert and.

Using a nested exist quantifier in a forall, we are able to assert that for every step, there exist 6 people, the exist with defined bounds for every person means there is no need to write out all of the

```
(forall ((step Int))
(=>
    (<= 1 step N)
    (and
    ; per step one or two people must take a boat, whilst 4/5 stay
    (exists ((p1 Int)(p2 Int)(p3 Int)(p4 Int)(p5 Int)(p6 Int))
        (and
        ; distinct people
        (distinct p1 p2 p3 p4 p5 p6)
        (<= 1 p1 6)
        (<= 1 p2 6)
        (<= 1 p3 6)
        (<= 1 p4 6)
        (<= 1 p5 6)
        (<= 1 p6 6)
```

Once the bounds are set for the parameters, we specify that across two consecutive steps, one or two people change sides (using the boat). Whilst four/five will not move.

```
        ; 1 or 2 change sides
        (or
            (and
                (distinct (B step p1) (B (+ step 1) p1))
                (distinct (B step p2) (B (+ step 1) p2))
            )
            (and
                (distinct (B step p1) (B (+ step 1) p1))
                (= (B step p2) (B (+ step 1) p2))
            )
        )
        ; 4 dont move
        (= (B step p3) (B (+ step 1) p3))
        (= (B step p4) (B (+ step 1) p4))
        (= (B step p5) (B (+ step 1) p5))
        (= (B step p6) (B (+ step 1) p6))
))
```

The next condition that has to be defined the one, where if cannibals outnumber the missionaries on any given side, then this will result in a tragedy, to have *Z3* find the solution to this problem, we need not define this behaviour, but rather we must define that for any step at any given side, the amount of cannibals may not be higher than the amount of missionaries on either side. We do this within the aforementioned forall clause, By calling two functions that differ in which side they check.

```
        ; no missinaries are eaten
        (SafeRSide step)
        (SafeLSide step)
    )))
```

-

These functions count the amount of missionaries vs cannibals at the given side, and ensure that the amount of cannibals is smaller or equal to the amount of missionaries. Who are p 1,2,3 on the table whilst cannibals are p 4,5,6. This however is only the case if there is at least one missionary on that side which is enforced by the implication and the atleastOneMiss function.

```
; check whether there are more or equal amount of missionaries on left side
; at given step
(define-fun SafeLSide ((step Int)) Bool
    (=> (atleastOneMiss 0 step)
        (<=
            (+
                (ite (= (B step 4) 0) 1 0)
                (ite (= (B step 5) 0) 1 0)
                (ite (= (B step 6) 0) 1 0)
            )
            (+
                (ite (= (B step 1) 0) 1 0)
                (ite (= (B step 2) 0) 1 0)
                (ite (= (B step 3) 0) 1 0)
            )
        )
    )
)
    ... SafeRSide similar but checking for 1 not 0
```

We must also specify that for all steps the all people may only be on the right or left side.

```
; for all the steps people may be on the right or left side
(forall ((x Int) (y Int))
    (=> (and (<= 1 x N) (<= 1 y 6 ))
        (<= 0 (B x y) 1)
    )
)
```

The next two conditions are tightly coupled and thus may be asserted together. They have to do with the behavior of the boat. We do this by asserting that there does not exist a step such that the step is even (meaning boat is on the right side) where at step+1 (next step) a person has traveled from the left to the right side.

What we found that is we start at an odd step, 1. then the boat would be on the left side, but then on step 2, the boat would move to the other side, as we know that per step at least one or two people must change sides. What we found, is that for every even step, the boat would be on the right side, and for every odd step the boat would be on the left. This means that there may not exist a case where the step is even, and at the next step a person goes from left to the right, as that would be impossible as we have established on every even step the boat is on the right side. We evaluate whether the step is even using the modulo operator seeing if the %2 operation yields 0.

```
(not
    (exists ((step Int))
        (and
            ; within bounds
            (<= 1 step (- N 1))
            ; step is even
            (isEven step)
            ; exists a person who goes left to right
            ; on an even step, (where boat is on right side)
            (exists ((person Int))
                (and
                    (<= 1 person 6)
                    (= (B step person) 0)
                    (= (B (+ step 1) person) 1)
                )
            )
        )
    )
)
```

The final thing that has to be added to the large and assert, is the limit on moves to tell *Z3* in how many moves it should be solving in. In this case it is known that it can be done in 12 moves. (in code its 13 because of N-1)

```
;limit on steps
(<= 1 N 13)
```

1.2.1 Conclusion

This is the interpreted output given by *Z3*, where M states for missionaries and C - cannibals:

	M		C			
s	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	1	1	1
5	0	0	0	1	1	0
6	1	0	1	1	1	0
7	0	0	1	0	1	0
8	1	1	1	0	1	0
9	1	1	1	0	0	0
10	1	1	1	0	1	1
11	1	1	0	0	1	1
12	1	1	1	1	1	1