**Self-stabilizing maximum matching on a Dijkstra ring**

**States**

The following are the **3** states that each processor can take with regards to mutual exclusion algorithm done on a directed Dijkstra ring.

1. **0** indicates that the processor is forming an edge with its anti-clockwise neighbor.
2. **1** indicates that the processor is forming an edge with its clockwise neighbor.
3. **NULL** indicates that the processor is single.

**Algorithm**

P0: do forever

if rn-1 = 0

r0 = NULL

else

r0 = 0

Pi!=0: do forever

if ri-1 = 1

ri = 0

else

ri = 1

Now let me try to explain how the algorithm works.

First, I’ll start with how Pi!=0 works.

Pi looks at the previous processor’s register ri-1 to see with whom the processors want to make an edge with.

1. 1 means that the processor is trying to make an edge with the next clockwise neighbor which is Pi. So, Pi completes the edge by assigning ri = 0 to establish the edge completely from both ends.
2. Else Pi initiates a new edge with Pi+1 by assigning ri = 1.

Now we’ll look at how P0 works.

P0 looks at the previous processor’s register rn-1 to see with whom the processor wants to make an edge with.

1. NULL indicates that the processor is not yet tied to any other processor so P0 tries to form an edge with Pn-1 by setting r0 = 0.
2. 1 means that the processor is trying to make an edge with P0. So, P0 completes the edge by assigning r0 = 0 to establish the edge completely from both ends.
3. 0 means that the processor has already established an edge with its other neighbor. This happens only when the number of processors is odd. So P0 sets itself to NULL since in this algorithm P0 will only form an edge with Pn-1.

**Proof for self-stabilization**

Firstly, since we are using a Dijkstra ring, we can utilize the mutual exclusion property that only processor runs its code at a time and next processor that runs is its clockwise neighbor.

Now, I’ll try to prove that every execution that starts with the root and once it reaches the root again will result in a safe state. If that’s the case, every execution before the root can be considered non harmful (and useless) and this will be proved below.

Let’s start with the root.

If rn-1 =

1. 1: P0 sets r0 = 0 and becomes matched.
2. NULL: P0 sets r0 = 0 goes into waiting.
3. 0: P0 becomes NULL and becomes single.

Now when P1 executes we can be sure that r0 would never be 1. So r1 becomes 1.

When P2 executes we can be sure that r2 would become 0 and form an edge with P1.

As you can see, this will continue until it reaches Pn-1 and when

1. i is even, ri = 0
2. i is odd, ri = 1

So,

1. If n is even then, rn-1 would become 1 and the root would become 0 forming the edge and reaching the safe state.
2. If n is odd then, rn-1 would become 0 and the root would become NULL reaching the safe state.

As we can see above in both cases, we reach the optimal solution and safe state in one complete iteration + root execution.