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 r(n-1) = 0
                r0 = NULL
        else
                r0 = 0
Pi!=0: do forever
        if r(i-1) = 1
                ri = 0
        else
                ri = 1
```

Now let me try to explain how the algorithm works.

First, I'll start with how $P_{i!=0}$ works.

P_i looks at the previous processor's register r_{i-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 P_i. So, P_i completes the edge by assigning $r_i = 0$ to establish the edge completely from both ends.
2. Else P_i initiates a new edge with P_{i+1} by assigning $r_i = 1$.

Now we'll look at how P_0 works.

P_0 looks at the previous processor's register r_{n-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 P_0 tries to form an edge with P_{n-1} by setting $r_0 = 0$.
2. 1 means that the processor is trying to make an edge with P_0. So, P_0 completes the edge by assigning $r_0 = 0$ 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 P_0 sets itself to NULL since in this algorithm P_0 will only form an edge with P_{n-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 r_{n-1} =

1. 1: P_0 sets $r_0 = 0$ and becomes matched.
2. NULL: P_0 sets $r_0 = 0$ goes into waiting.
3. 0: P_0 becomes NULL and becomes single.

Now when P_1 executes we can be sure that r_0 would never be 1. So r_1 becomes 1.
When P_2 executes we can be sure that r_2 would become 0 and form an edge with P_1.
As you can see, this will continue until it reaches P_{n-1} and when

1. i is even, $r_i = 0$
2. i is odd, $r_i = 1$

So,

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