EE 382N: Distributed Systems **Fall 2017**

Lecture 1: Oct 13

Lecturer: Vijay Garg *Scribe: Marina Thomas*

1.1 Termination Detection and Diffusing Computation

It is an important problem in distributed computing to develop efficient algorithms for termination and deadlock detection. Both termination and deadlock are stable properties and therefore can be detected using any global snapshot algorithm.

1.2 Diffusing Computation

1. Every process is either active or passive
2. Initially all processes are passive except for the special process called environment.
3. A passive process can become active only if it receives a message
4. Only active process can send messages
5. Once the job is done, the process becomes passive

Definition 1.1 *Termination - Happens when all processes are passive and all channels are empty.*

Algorithms for many problems can be structured as diffusing computations. For example, the shortest paths from a processor in a networkcan be structured as diffusing computations.

Below a distributed shortest-path algorithm is used to illustrate the concepts of a diffusing computation. There are a bunch of airports and the 'special' airport is Austin. Assume that we are interested in finding the shortest path or the least cost to fly from Austin(AUS) airport to all other airports.

Every airport maintains the cost of the shortest path from the Austin to itself as known to it currently, parent or the predecessor of itself in the shortest path from the AUS to itself.

Austin(the special airport/environment) knows how much it will take to fly to itself. Cost of flying from Austin to itself is 0.

AUS starts up the diffusing computation by sending the cost of the shortest path to be 0 using a message type path. Austin will send ("path",0) i.e (path, cost)

Any airport that receives a message from AUS of type path with cost c determines whether its current cost is greater than the cost of reaching AUS plus the cost of reaching from AUS to itself. If that is indeed the case, then it has discovered a path of shorter cost and it updates the cost and parent variables. Further, any such update results in messages to its neighbors about its new cost. The algorithm is shown in Figure 1.1 amd Figure 1.2.

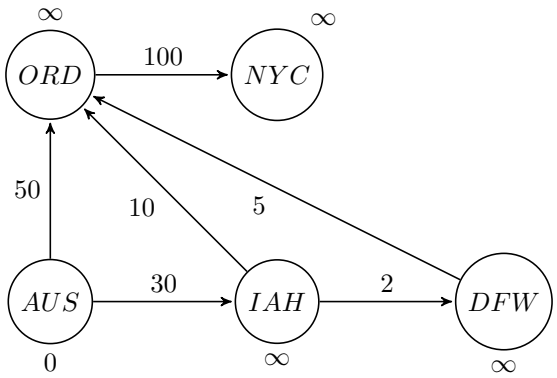


Figure 1.1: Shortest route - initial state

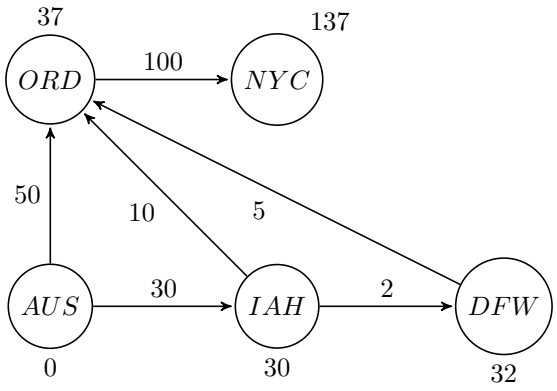
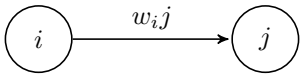


Figure 1.2: Shortest route

The initial cost at all other airports is initialized to ∞

When an airport(j) receives a path message, it will check

if cost[i] + w_ij ¡ cost [j]
  then cost[j] = cost[i] + w_ij;



The algorithm works fine, but no process ever knows when it is done, that is, the cost variable will not decrease further.

How do you know that the estimate is correct? → When all processes are passive and all channels are empty.

The goal is to come up with an algorithm that can detect this termination and how we can extend the diffusing computation to detect termination.

Is this predicate stable? All processes are passive and all channels are captured. Hence it is stable.

1.2.1 Dijkstra and Scholten's Algorithm

P_i is green iff P_i is passive and all its outgoing channels are empty

Then, TERM $\equiv (P_1\ is\ green) \wedge (P_2\ is\ green) \wedge (P_3\ is\ green) \ldots \wedge (P_n\ is\ green)$

This is conjunctive predicate.

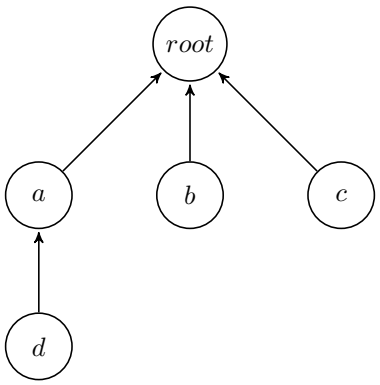
$P_i\ is\ red\ \equiv P_i\ is\ notgreen$

Maintain a set T st it contains all red nodes $\vee$ processes.

T= {} $\rightarrow$ TERM

I need to maintain this set in a distributed manner. T is going to be maintained as a spanning tree.

root≡environment
Every node is going to have a variant parent



If a node received a message, and if parent is null, make sender the parent.
If a node has a parent, don't reset parent. Eventually all nodes will turn red and will be part of the tree.

Code:

$\underline{P_i}$:

var parent : id	initially null init passive for all
var state:	active, passive
var D : int;	#of messages sent - # of ack received init 0

For every channel if a message is sent, a signal (ack) is received

On receiving a program message from P_j

```
if(parent == null) then
        parent := j;
```

How to leave from this tree?
The joining node does not send the signal to the parent the first time

```
if (parent == null) then
```

```
      parent := j
      state = active
else
     send signal to P_j
     On receiving signal
     D:= D-1
```

On sending message (only if state == active)
D := D+1 ==¿

When do you turn green?

```
if state == passive && D==0: && parent != null
        send signal to parent
        parent := null
```

A leaf node that has left the tree can join back the tree at some other point if made active by any other node.

Environment detects termination when D=0

If two process sends message to same node, it accepts one and send signal back to the other.

Overhead = number of program messages