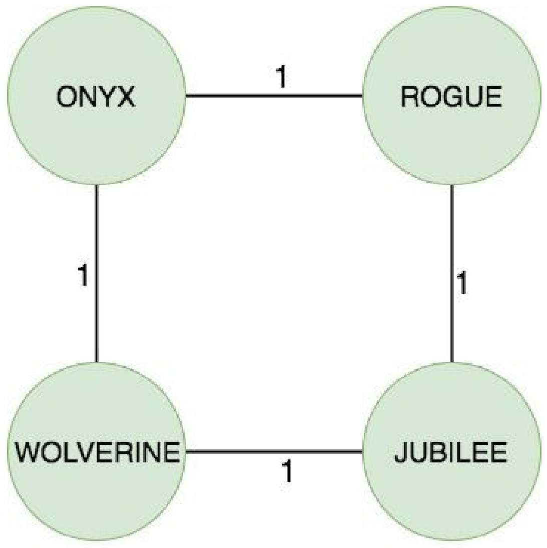
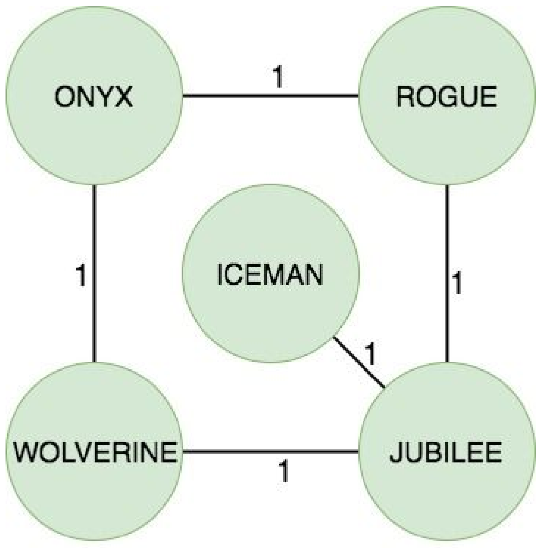
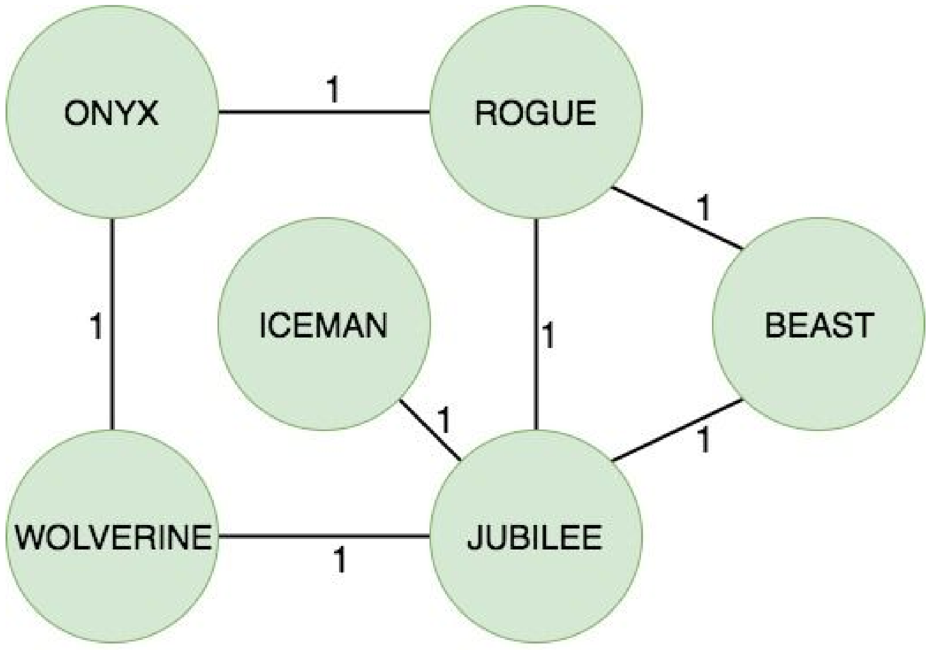
Graphs →



4 Nodes



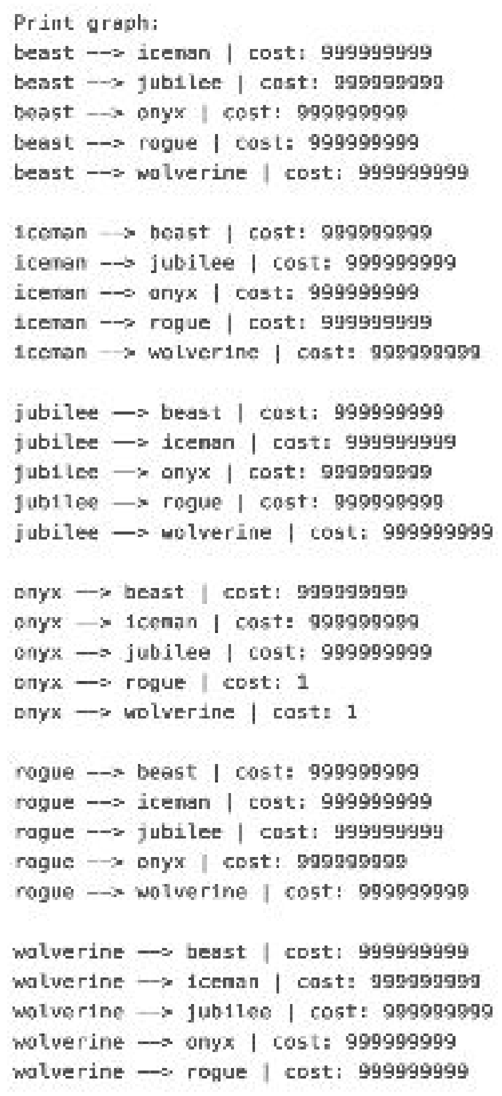
5 Nodes



6 Nodes

We have used the above graph configurations as input to the distance vector routing algorithm.

Graph representation →



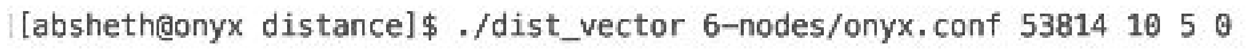
```
Print graph:
beast --> iceman | cost: 999999999
beast --> jubilee | cost: 999999999
beast --> onyx | cost: 999999999
beast --> rogue | cost: 999999999
beast --> wolverine | cost: 999999999

iceman --> beast | cost: 999999999
iceman --> jubilee | cost: 999999999
iceman --> onyx | cost: 999999999
iceman --> rogue | cost: 999999999
iceman --> wolverine | cost: 999999999

jubilee --> beast | cost: 999999999
jubilee --> iceman | cost: 999999999
jubilee --> onyx | cost: 999999999
jubilee --> rogue | cost: 999999999
jubilee --> wolverine | cost: 999999999

onyx --> beast | cost: 999999999
onyx --> iceman | cost: 999999999
onyx --> jubilee | cost: 999999999
onyx --> rogue | cost: 1
onyx --> wolverine | cost: 1

rogue --> beast | cost: 999999999
rogue --> iceman | cost: 999999999
rogue --> jubilee | cost: 999999999
rogue --> onyx | cost: 999999999
rogue --> wolverine | cost: 999999999

wolverine --> beast | cost: 999999999
wolverine --> iceman | cost: 999999999
wolverine --> jubilee | cost: 999999999
wolverine --> onyx | cost: 999999999
wolverine --> rogue | cost: 999999999
```

When the distance vector routing is initiated on any of the node, graph view of a particular node is generated as shown above.
Above figure is Onyx's view of the network.

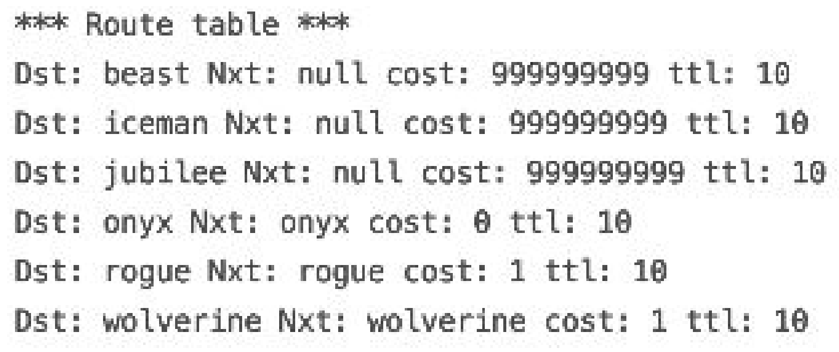
Data structure: Map of Map
Graph Map <key: source, value: Map <key: destination, value: cost> >

In the above data structure, the cost is the cost to go from source to destination.

Routing Table Representation →



```
[absheth@onyx distance]$ ./dist_vector 6-nodes/onyx.conf 53814 10 5 0
```



```
*** Route table ***
Dst: beast Nxt: null cost: 999999999 ttl: 10
Dst: iceman Nxt: null cost: 999999999 ttl: 10
Dst: jubilee Nxt: null cost: 999999999 ttl: 10
Dst: onyx Nxt: onyx cost: 0 ttl: 10
Dst: rogue Nxt: rogue cost: 1 ttl: 10
Dst: wolverine Nxt: wolverine cost: 1 ttl: 10
```

When the distance vector routing is initiated on any of the node, routing table is formed as shown above (as Onyx in the above example).

Data structure for Node: struct

```
// Routing table entry
typedef struct {
   char Dst[50];            // Name of the destination
   char nxtHop[50];         // Next hop along the route to get to destination
   int cost;                // Cost from current node to destination
   short ttl;               // Time to live
} route_entry;
```

Data structure for routing table: Map

Routing_table Map<key: destination, value: destination node>

NOTE → Both route table and graph at a particular node are stored as maps for faster access in O(1).

Describe multithreading components: Shared variables, mutex and condition variables →

Our multi-threaded design uses two threads, one to compute the routing table from the graph and one to send out period updates to its neighbors.
In order to safely synchronize the use of the routing table data structure we used a standard mutex lock. The lock is acquired for the routing table whenever it needs to be read/written and is released once the read/write operation is done.

Time to establish routes to all nodes during initialization (Convergence with all nodes up) →

Number of nodes	**Time to converge**
4 Nodes	2 Cycles
5 Nodes	3 Cycles
6 Nodes	8 Cycles

Note: The number of cycles it takes to converge also depends on which node comes up first.

Time to converge to a steady state after a node goes down →

Number nodes	**Time to converge**
4 Nodes	2 Cycles
5 Nodes	4 Cycles
6 Nodes	5 Cycles

Note: This table assumes that only one node has failed.

Your analysis should vary the Infinity and describe the effect that this variable has on the time it takes to converge →

The effect of the infinity value we choose determines when we consider the link to be a failure. For example if the value of infinity is 1000 and the cost of a link changes to greater than 1000 then the problem is that, the link whose cost changed is considered as dead, since it is has gone beyond the considered infinity value.

Describe how split horizon affects convergence →

Split horizon is a technique in which to prevent infinite loops from occuring. It works on the basic principle that information about the routing for a particular packet is never sent back in the direction in which it was received from.
For example:

If Node A routes to Node C via Route B then Node A should not tell Node B that it routes to Node C via Node B. To understand why this may be a problem, assume that the link from Node B to Node C fails. In that case if Node B has in its routing table, that Node A can reach Node C it will try to route packets to Node C via Node A without considering the fact that Node A actually routes to Node C via Node B. This creates a infinite loop and greatly increases the time of convergence.