Networking Project Technical Report

Problems with Poisoned Response

Within computer networking, one of the main goals is to be able to send data from one computer to another computer. This would be easy if both computers had a direct connection to one another, but alas in the real-world it is often that these two computers do not have a direct connection to each other. However, it is still possible for these two computers to communicate via the use of routers, or otherwise intermediary computers, that connect to one another. So, while there might not necessarily be a direct connection from the source to destination, there might exist a path between multiple computers where eventually data sent from the source computer will reach the destination computer after going through multiple other intermediary computers or routers. All of these inter-connected computers together is known as a network.

There are, however, three key issues about this way of communication within computer networks that need to be addressed. First is how exactly a computer is able to determine if there even is a valid route from itself to whatever computer it wishes to send information to. Second is figuring out what the shortest or fastest route is from itself to its destination, assuming more than one route exists and some are faster than others. And lastly, how exactly each computer in the network should handle changes, such as if a connection between two computers fails and how that might affect the routes taken by each computer.

These issues are handled by algorithms or processes known as routing algorithms. There are many different types of routing algorithms that have different functions, different pros and cons, and different requirements of the network in order to work. However this report will be focusing on just one of these algorithms called the distance vector algorithm. Specifically, this report will focus on one way a distance vector algorithm may handle changes in a network, that being a method called poisoned response, and the issues caused by this approach.

Distance Vector Algorithm

The distance vector algorithm is an algorithm that utilizes the Bellman-Ford algorithm as part of itself. The Bellman-Ford algorithm is an algorithm to determine the minimum cost to get from one node to another node on a graph where connections between nodes has an associated cost to traverse. The algorithm is defined as follows:

$$d_x(y) = \min\{c(x,v) + d_v(y)\}$$

$d_x(y)$ denotes the minimum cost from source x to destination y. $c(x,v)$ represents the minimum cost from source x to some neighbor of x, notated as v. $d_v(y)$ denotes the cost to get to destination y from v.

Because x can have more than one neighbor, v can refer to multiple different nodes, but we only want to save the result of the neighbor that gives us the least cost, hence the $\min\{\}$. This equation is also recursive in nature, as v can have its own set of neighbors, and their neighbors can have neighbors, and so on. This algorithm properly results in the minimum cost to get from one node to another, given that at least one route exists.

The distance vector algorithm acts as a kind of extension of the Bellman-Ford algorithm specific to computer networks. The distance vector algorithm specifies that each node will store its own set of information for its estimated minimum distance from itself to all the other nodes in the network, as well as what node is next along the route. Each node will periodically send its estimated distance vector information to all of its neighbors, in which the Bellman-Ford algorithm will be used by each neighbor to update its own estimates. Eventually, over time, all nodes in the network will eventually settle down on the actual minimum distance from itself to all other nodes in the network.

Simulating the Distance Vector Algorithm

To begin with, this simulation model assumes that in the initial state of the network, each node network already knows its other neighbors and the direct distance to said neighbor. Figure 1 below shows a simple network as an example that we

will be examining:

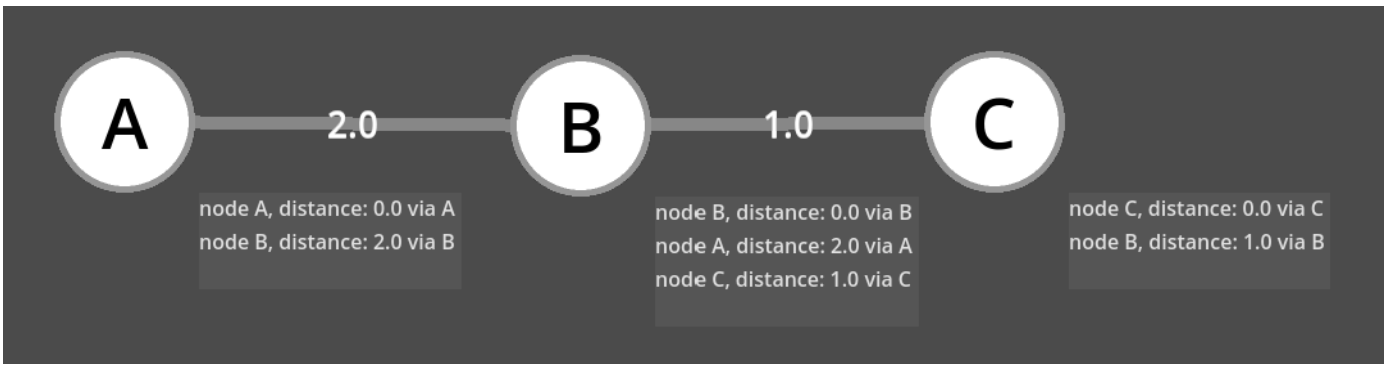

Figure 1. Example of a simple network just initialized

Here we have 3 nodes labeled A, B, and C. There exists a link between A and B with a cost of 2.0 as well as a link between B and C with a cost of 1.0. Note that this 'cost' is somewhat arbitrary, as it can represent time, distance, or something else, but the goal is to remain the same in minimizing this cost.

In the initial state of the network, each node knows only its neighbors. Because of this, A does not know about the existence of C, and C doesn't know about A. This is eventually resolved when B decides to randomly send out its distance vector information to its neighbors A and C.

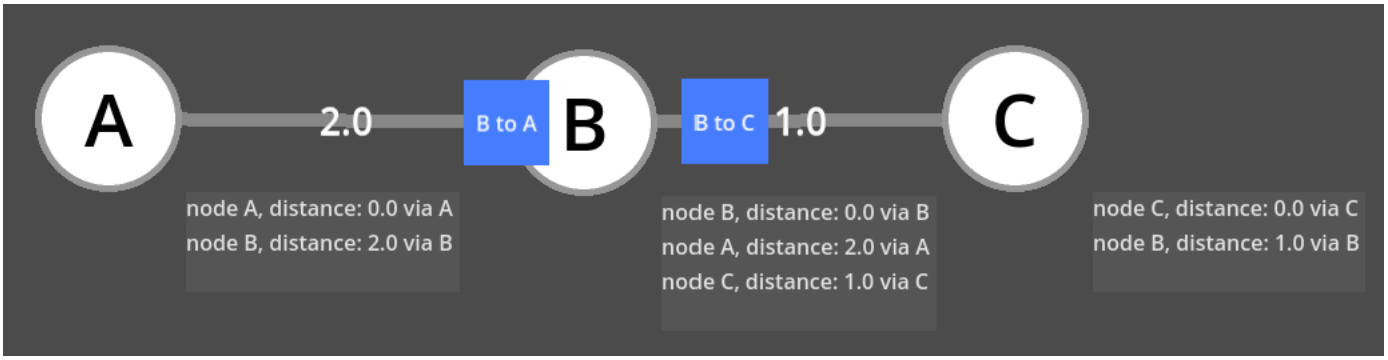

Figure 2. A node in the process of forwarding its distance vector information

Here in Figure 2 we see that B is in the process of sending its information to both A and C through the use of packets, represented by the blue squares. Packets are just blocks of information that are shared between computers, and it will be assumed that all packets contain the distance vector information from their sender.

Eventually when the packets reach their respective destinations, nodes A and C will be updated to know about each others' existence and the need to go through node B in order to reach each other.

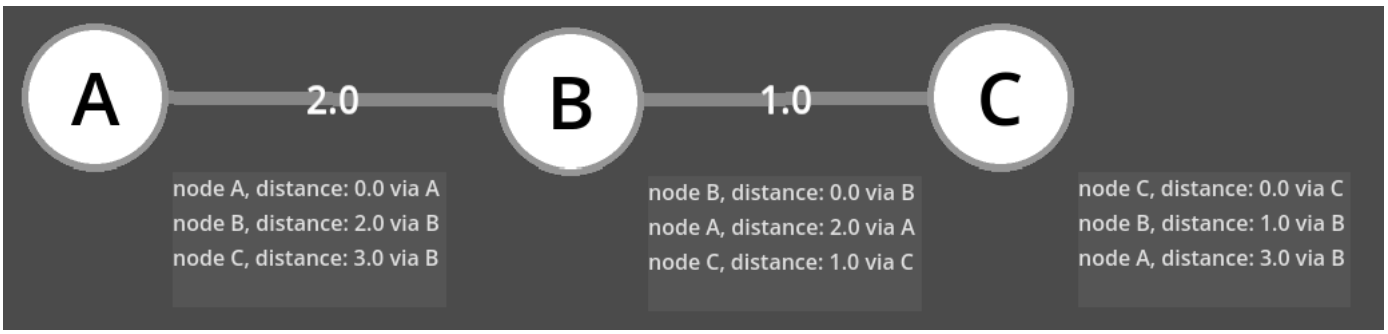

Figure 3. A stabilized network

This is the ideal case for the distance vector algorithm. However, things become complicated when the state of the network changes, such as if a link between two nodes fails.

Count-to-Infinity

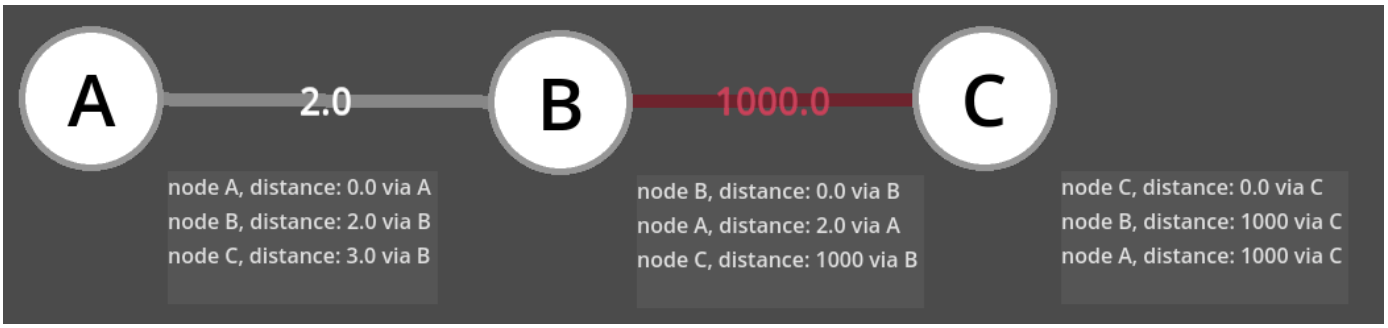

Figure 4. A network with a broken link

In Figure 4, we have the same stabilized network as Figure 3, but this time the link between B and C has become faulty, essentially meaning that node B can no longer communicate with node C, and vice versa. In the traditional distance vector algorithm, this faulty link would be represented as having a 'cost' of infinity.

However for our simulation 'infinity' will just be the extremely high cost of 1000.0. Additionally, it is assumed that both nodes B and C can measure the failure of the link immediately and can adjust their distance vector information accordingly.

So, how exactly should the distance vector algorithm handle this? While node B knows that it does not have a connection to C, node A still thinks that it does. Let's see what happens if, and large emphasis on if, node B were to do nothing at this point. Eventually, node A will send its own distance vector information to B, as seen in Figure 5 below.

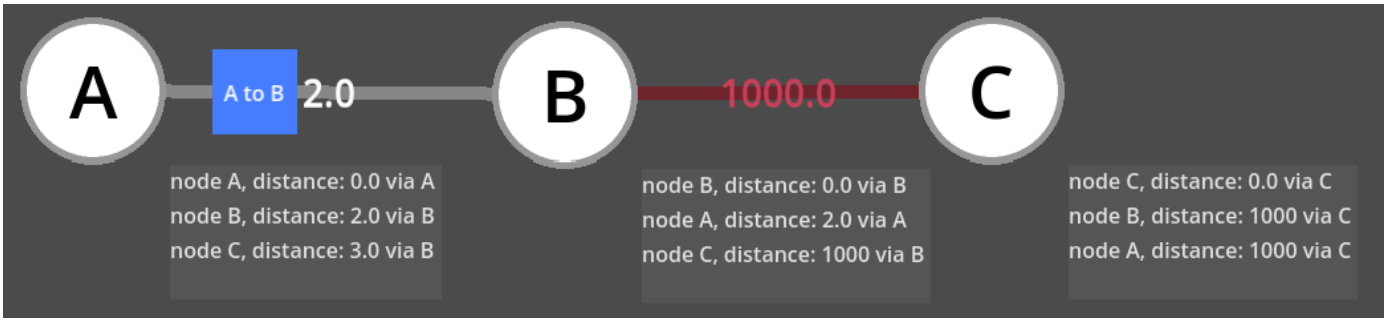

Figure 5. Node A sending outdated information

Upon receiving this information, node B would see that node A 'has' a route to node C, not taking into account that this route is via B itself! The result is that B updates its cost to become the cost to get to C from A, plus the cost for B to get to C because of the Bellman-Ford equation, which comes out to a cost of 5.0. Figure 6 demonstrates this change below:

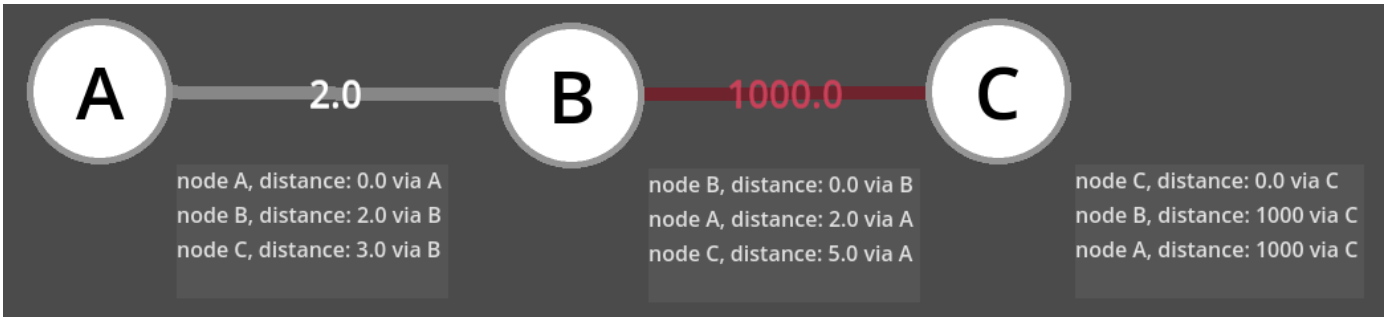

Figure 6. Result of B receiving outdated information

At this point node B now thinks that it can still reach C through node A, even though there is no longer a valid route from either A or B to C! However, things still change. If node B were to send out its distance vector information, node A would receive it and see that B's distance to node C has changed to 5.0, and because A uses B to get to C it must change its own cost to be the cost to get to C from B plus the cost for A to get to B in the first place, which works out to $5.0 + 2.0 = 7.0$.

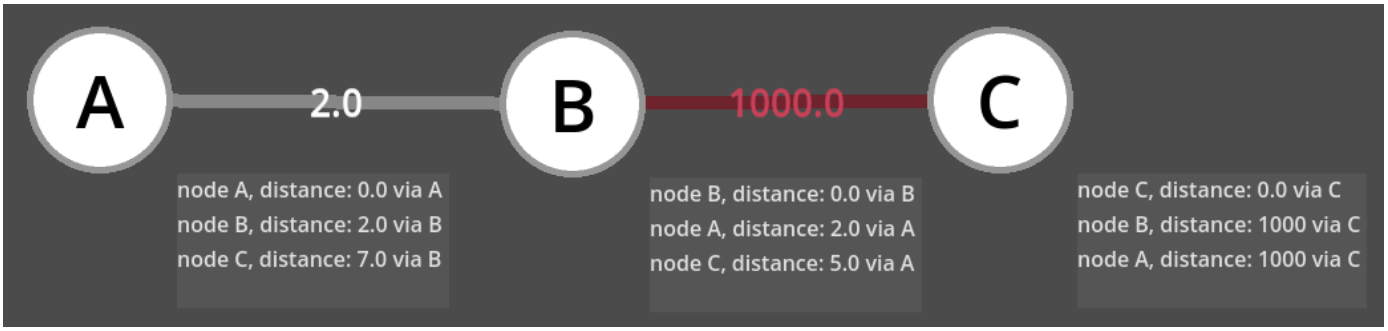

Figure 7. A network in the state of counting to infinity

This change in A's distance vector information is shown in Figure 7. However, now that A has a different cost to get to C, B will eventually have to update its own distance to C because it uses A to get there. This behavior effectively repeats forever, thus the name of the count-to-infinity problem. In this situation, both nodes A and B think that they have a connection to node C through each other, when in reality the route goes in a loop between A and B in an ever-increasing cost.

Poisoned Response

Poisoned response is a method that the distance vector algorithm can use to mitigate this count-to-infinity problem. Previously, it was emphasized that B did nothing when the link initially became faulty. This time however, B will instead react and send information to A first, before A gets the time to send its outdated information to B and causing the count-to-infinity problem.

This information that B sends to A is that its connection to C is infinite, once again represented by the very large cost of 1000.0. This way, A will see that B's cost to C has dramatically increased, and therefore A should instead look for a more minimal cost route to get to C if possible since it currently uses B to route to C. Figure 8 below shows this in action.

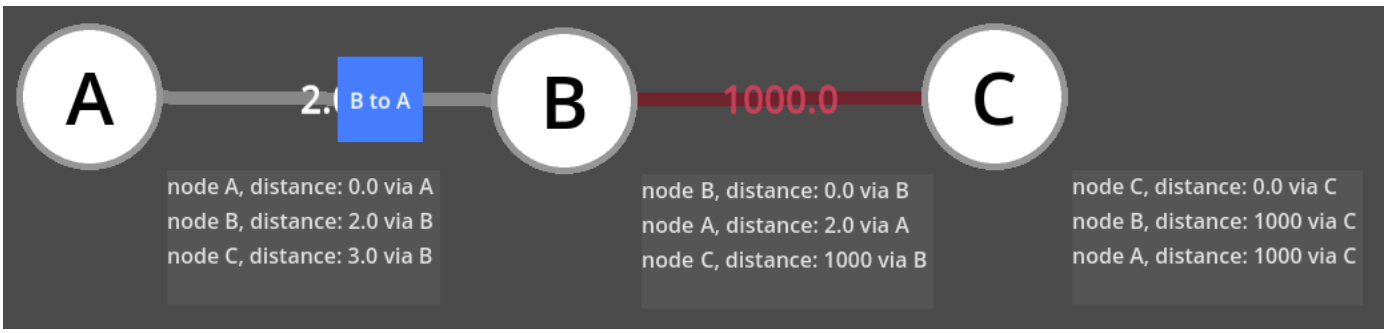

Figure 8. Node B first sending a poisoned response to A

The information that B sends to A where it intentionally includes a very large cost is called a poisoned response. The resulting state of the network can be seen in Figure 9 below.

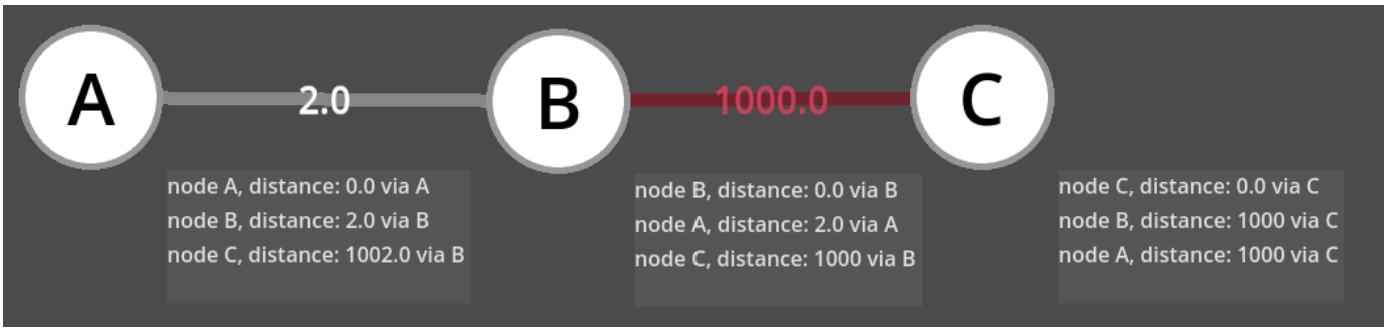

Figure 9. A stabilized network with a broken link

In the end we result in a proper network in which A and B both know that they can not reach C.

The problem

The issue with poisoned response is that it is assumed that B is successful in first sending the poisoned response to A so that A does not send outdated information back to B. However, there is nothing from stopping node A from sending its own packet of information out at the same time B's packet of new information is arriving to A.

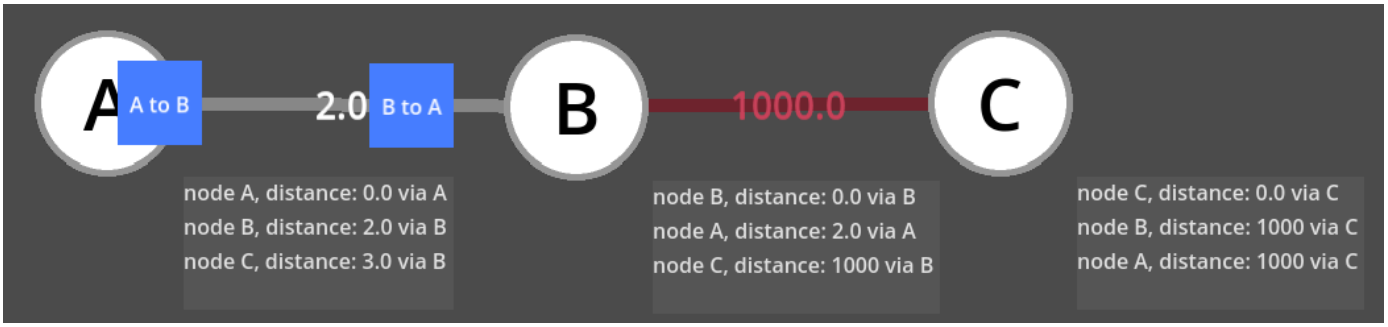

Figure 10. Node A sending a packet at the same time as B's poisoned response

Here in Figure 10 we can see this in action. Node B has just sent its own packet of information to A, but before it reaches its destination node A sends its own packet of now outdated information.

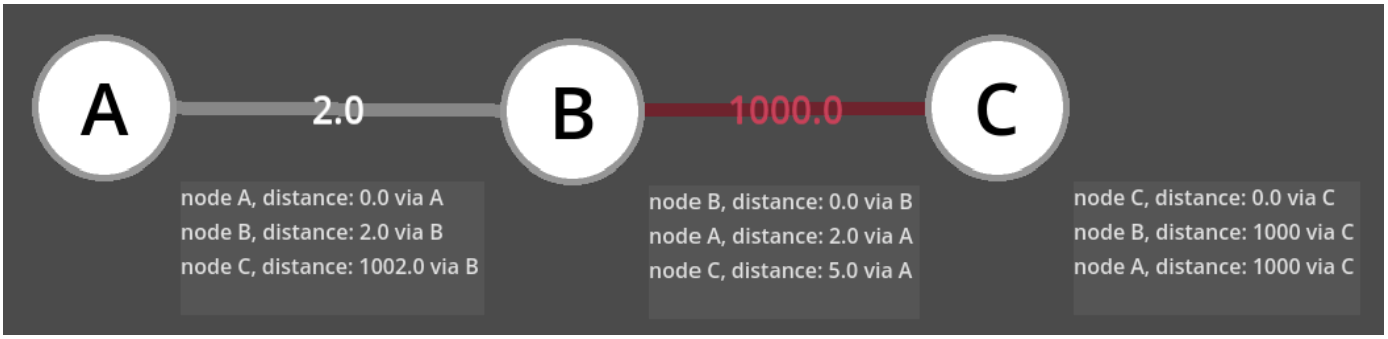

Figure 11. Node B still believing there is a connection to C despite having sent a poisoned response

The result can be seen in Figure 11. Now node B thinks it has a connection to C through A despite A not having so. If B were to now send a packet immediately after receiving this information, A will now think there is a route to C through B, and accept the new cost. Eventually B will receive this new information from A, and update its own cost. This has essentially returned us back to the count-to-infinity problem! The occurrence of both nodes sending their information at roughly the same time causes the poisoned response method to be ineffective.

Results and analysis

This scenario of poisoned response not working in properly communicating a link failure to nodes is primarily caused by a race condition due to multiple packets being in-transit. Because the issue is caused by multiple packets being in-transit at the same time, we can determine some factors that might make this scenario more or less common.

For one, nodes frequently sending out their distance vector information to their neighbors increases the chance of this issue happening, as there are simply more opportunities for it to happen. The size of the network and the speed of links also affects the chances of this issue occurring. The larger the network, the longer it takes for information to propagate and for nodes to settle down to the correct result, thus giving a longer time window for the problem to occur. Links being slow also makes it more likely for this problem to occur, as packets would exist in-transit for a longer period of time.

Now with this understanding, there are multiple ways to address this issue.

The frequency at which nodes send out their distance vector information to their neighbors can be reduced, increasing the time between packets being sent and thus reducing the chance two conflicting packets are sent at roughly the same time. This does however result in the network taking longer to update and respond to changes to the network.

Another solution could be avoiding 'isolated links', in which there is a singular point of failure and no alternative route to get to the same node. If there was an alternative route the count-to-infinity problem would not persist very long as

eventually the nodes would discover the alternative route, and consider that one better than the infinite cost they will eventually reach up to. However, this still does not prevent the issue altogether if it just so happens that multiple links fail at the same time, and end up segmenting the network into two or more disconnected sub-networks.

One might also just be ok with count-to-infinity problem, maybe even adding a 'cap' from which any distance greater than some number is considered unreachable. This would allow the count-to-infinity problem to potentially stop in a reasonable amount of time. However this would limit the size of the network since an especially large network with lengthy connections might have a valid route be so long nodes start considering other nodes unreachable when in reality they still are.

Lastly, one could simply use a different routing algorithm altogether if need be and if they are able to use a different algorithm, such as Dijkstra's algorithm. However, this too has its own pros and cons depending on the configuration of the network in question and if it is even possible to use a different routing algorithm.

Novel Contributions

The network diagrams used above in Figures 1-11 were made through the use of a custom application designed to attempt to simulate networks and the distance vector algorithm. The application uses the Godot 4 game engine in order to provide a graphical user interface. In order to best simulate the behavior of computer networks, objects are able to exist independently from their creator, such as packets just being packets with a source and destination and their own data.

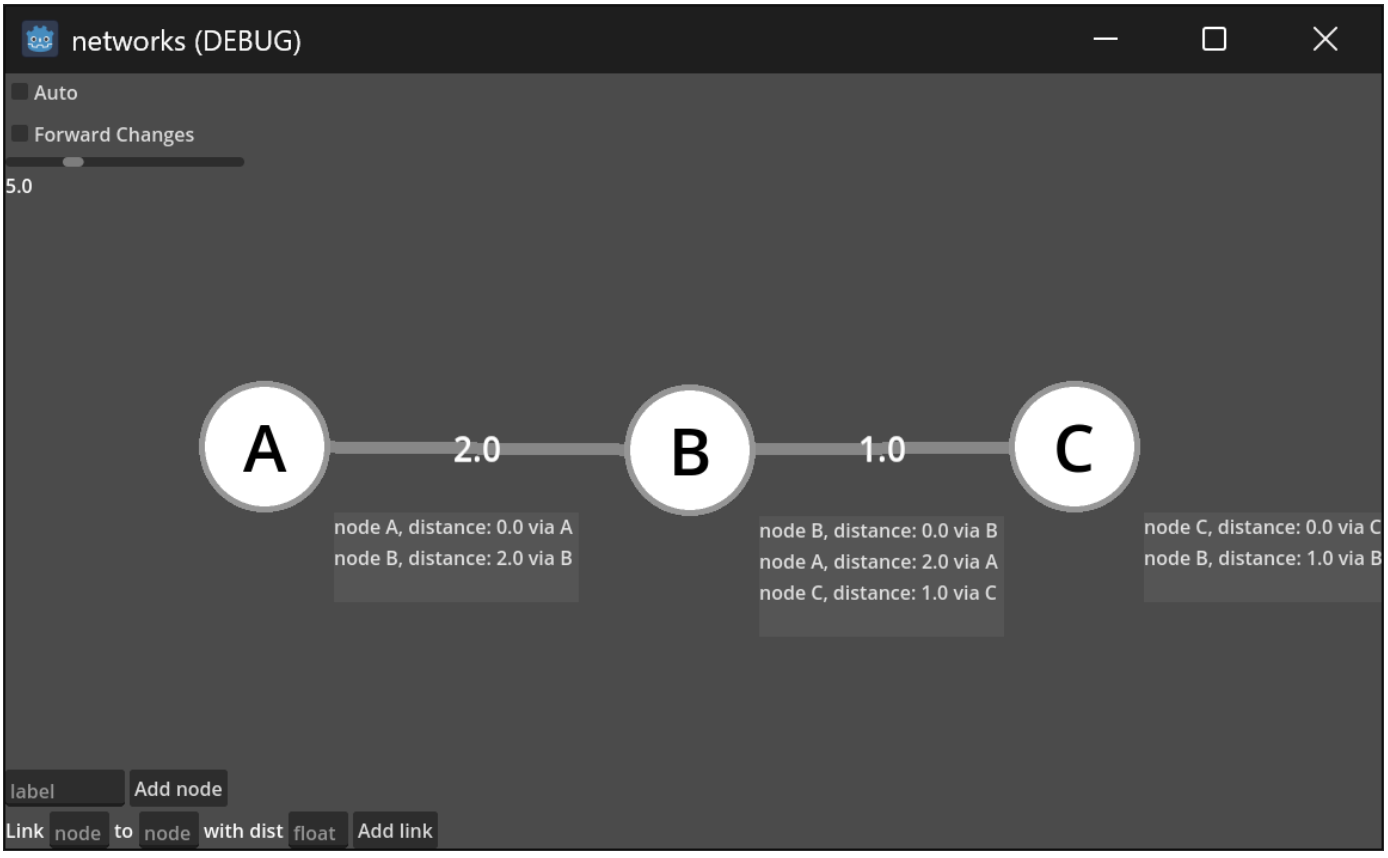

Figure 12. Screenshot of application upon launch

Unfortunately, due to time constraints, the application is extremely barebones and has not been tested against crashing. Only the following features have been implemented:

- Moving the camera with middle mouse drag
- Zooming in and out with scroll wheel
- Dragging nodes around with left mouse drag
- Forcing a node to send out packets with left-click

- Disabling and re-enabling links with right-click
- Deleting links by pressing T and having the mouse over it
- Creating new links between two nodes with some defined cost
- Creating new nodes with their own label
  - Note that deleting nodes has not been implemented
- Pausing and resuming the simulation with spacebar
- Toggling automatic random packet sending for nodes
- Toggling automatic packet forwarding on changes to a node's distance vector information
- Adjusting the timescale with a scrollbar on the top left

Despite the application being incomplete, hopefully it comes as somewhat useful for demonstrating how a computer network might function.

Code Explanation

The code for the application is written in multiple GDScript files, with their file extension ending in `.gd`. GDScript is a programming language that looks very similar to Python.

The majority of the implementation of the distance vector algorithm exists within the file `network_node.gd`. This script contains the logic for just an individual node, with multiple nodes running their own instance of the script.

The script first stores two dictionaries, one for the distance vector table and another for purely direct neighbor information. The methods `add_neighbor()` and `remove_neighbor()` do as they say, and are important in being able to dynamically modify the network at the user's will. The most important method however is the `update_distances()` method. This method takes in the source of the data as well as the distance vector table provided. Below is the code for the `update_distances()` method:

```
func update_distances(from_node: NetworkNode, new_data: Dictionary):
    var prev_distances = distance_vector.duplicate()
    var link_cost: float = self.my_neighbors[from_node]

    for dest in new_data:
        if dest == self:
            continue
        var their_cost: float = new_data[dest][0]
        var new_cost: float = link_cost + their_cost
        var old_cost: float = distance_vector.get(dest, [999999, null])[0]

        if new_cost < old_cost:
            self.distance_vector[dest] = [new_cost, from_node]

        if self.distance_vector[dest][1] == from_node:
            self.distance_vector[dest] = [new_cost, from_node]
```

First, a copy of the node's current distance vector table is created to be compared to. Then the cost to go to the node where we are getting data from is retrieved. We then loop over each entry in the incoming distance vector information. `new_cost` is calculated by taking their cost to go to some destination node plus the cost for us to get to them, same as the Bellman-Ford equation. Then the `old_cost` is retrieved. If we don't have an existing cost to the destination node, that means we have just discovered this node, so we set `old_cost` to be some really high cost so that the `new_cost` is assigned to it. Lastly is a special case for overriding our cost even the new cost is greater than our old cost if our current route to some node is forwarded through the node we just got data from. This essentially handles the case in which the distance for the node we just got data from became longer, and so we need to adjust accordingly.

The rest of the `.gd` scripts handle managing the simulation and is not too relevant on the discussion of computer networks. Regardless, there are numerous comments within the files explaining roughly what they do.