Final Project: Routing Algorithms

CS513: Computer Networks

Automne Petitjean
mpetitjean@wpi.edu

Javier Vela Tambo
jvela@wpi.edu

December 5, 2025

1 Introduction

For the final project, we selected the standard "Network Layer Routing" project. The goal of the project is to develop an application to create different network topologies (represented as graphs), execute the link-state and distance-vector routing algorithms, and query the routing tables for each network node. We implement the basic requirements of the standard project and extend it in the following ways:

- A Textual User Interface (TUI) to interact with the application.
- Additional features:
  - Visualization of the network topology.
  - Save and load network topologies from files.
  - Load scripted commands from a file to automate tests.
- An analysis of the performance of both algorithms in terms of convergence time and number of messages exchanged.
- A variation of the distance vector algorithm that includes a "via" field in the routing table to prevent "counting to infinity" issues.

The rest of the report is organized as follows: Section 2 describes the implementation of the project, including the algorithms used. Section 3 presents the experimentation and results obtained. Finally, Section 4 concludes the report with reflections on the project.

2 Implementation

2.1 Algorithms used

In this project, we compare two different algorithms: the distance vector algorithm and the link state algorithm. The link state algorithm is a simple graph traversal that looks at all the possible routes from one specific node.

The distance vector algorithm is run step by step. During one step, each node sends its current routing tables to all its neighbours. To simulate this sending process, we save a copy of all the routing tables at time t. Then this copy is given to each neighbour and used to build the new routing table at time $t + 1$.

The new table for A at $t+1$ is built by creating a table with only one route: the route to A with a distance of 0. It is then updated by the received tables from all neighbours. In our experiment, all tables are sent and received simultaneously. In a context where tables arrive irregularly, we would need to remove only the routes going through B when receiving a table from B. We would also need to remove routes going through neighbours when we detect that we're not connected to them anymore.

When receiving a table from a neighbour, a node uses it to update its own table. Let's call A the receiving node and B the sender. A compares each route in the B table (to which the distance from A to B has been added) with the routes it already knows and keep the shortest option.

2.1.1 Legacy distance vector vs our distance vector

In the classical distance vector algorithm, that we call legacy in this report, tables sent only contains a destination and a distance for each routes. This can create "counting to infinity" issues. In this work we will compare this legacy version with our own variation that includes destination, distance and via for each route. This allows us to ignore that goes through ourselves. If B thinks it can reach C through A, A necessarily knows better and don't need this information. Taking it into account runs the risk of propagating wrong information.

We implemented both methods and compared them to get an idea of the usefulness of such a failsafe.

2.2 Measuring errors

In this report we measure the errors in routing tables when using the dv algorithm compared to ls. We want to know in how many iterations of dv the routing table reaches an accurate result.

There are several ways of measuring the difference between the ideal routing table from ls with the routing table from dv at step t. Let us describe the various metrics used and their pros and cons. But first, here are some useful notations.

Let us note:

- G the graph
- A, B, C... nodes in graph G
- $d_{ls}(A,B)$ the distance between A and B as given by the link state algorithm
- $d_t(A,B)$ with $t \in \mathbb{N}$ the distance between A and B in the routing table of A after t iterations.

A naïve way to measure the error would be to do this :

$$naiveError_t = \sum_{A \in G} \sum_{B \in G} |d_{ls}(A,B) - d_t(A,B)|$$

This, however, presents some problems. Firstly, it is ill defined for cases where A and B are not connected or when the dv routing table doesn't know how to reach B even though a route exists. We can imagine another version:

$$numberErrors_t = \sum_{A \in G} \sum_{B \in G} 1 - \delta_{d_{ls}(A,B),d_t(A,B)} \text{ with } \delta_{a,b} = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$$

This would count exactly the number of routes for which the dv table has a different distance from the ideal route.

While this is an appropriate measure, it doesn't help in cases where the routing table knows of the correct distance but the route itself doesn't match. For this we have to introduce a new notation:
$msg_t(A, B)$ is the time it takes a message to go from A to B following the routes given by the dv algorithm at step t.

Due to the nature of this measure, it can only be computed when all nodes in the graph have a routing table computed by dv. Some messages may never reach their destination, either because they reach a node that doesn't know how to reach said destination or because it gets stuck in an infinite loop between two nodes (for this specific case we implemented a hop limit). It allows us imagine this error metric:

$$msgError_t = \sum_{A \in G} \sum_{B \in G} 1 - \delta_{d_{ls}(A,B), msg_t(A,B)}$$

This metric is still not perfect. It doesn't take into account that some routing tables can have a correct route stored with a wrong distance. We argue that it is a good metric as it is close real world expectations of routing tables. Each error means that a message took longer to reach the destination than the ideal route. We can also measure only the message that failed to reach their destination. Sending a message through a non-efficient route is less of an issue than being unable to reach the destination at all even though a route exists.

3 Experimentation and Results

BLABLABLA

3.1 Evaluation of our dv algorithm

Our proposed dv algorithm aims to fix the problem of the "count to infinity" posed by the legacy dv algorithm. Let us look at a simple example:

We assume that dv has been run enough time so that the routing tables are correct. If the edge between A and B suddenly becomes 300, the legacy algorithm will take a long time before correcting the routing tables.

This is a good illustration of counting to infinity. The bad news took a long time to travel, B and C remain convinced that they can reach A by going through each other, meaning that all messages from B or C to A are lost until the count reached 300. Meanwhile, our own dv algorithm doesn't have any error and after two steps, the routing tables are updated with the correct distances. The absence of errors can be explained by the fact that B refused to take a route from C that would go through itself thus avoiding the infinite loop route that loses messages.

However, our algorithm is not immune to "count to infinity" problems, we just have to find another example:

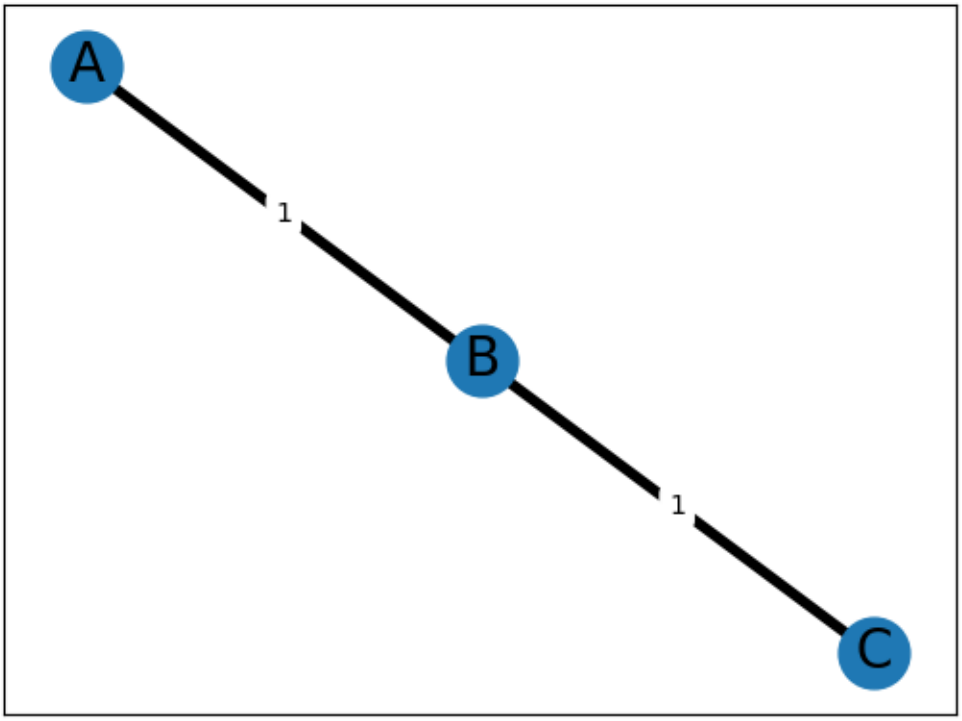

Figure 1: A simple example graph named Armand

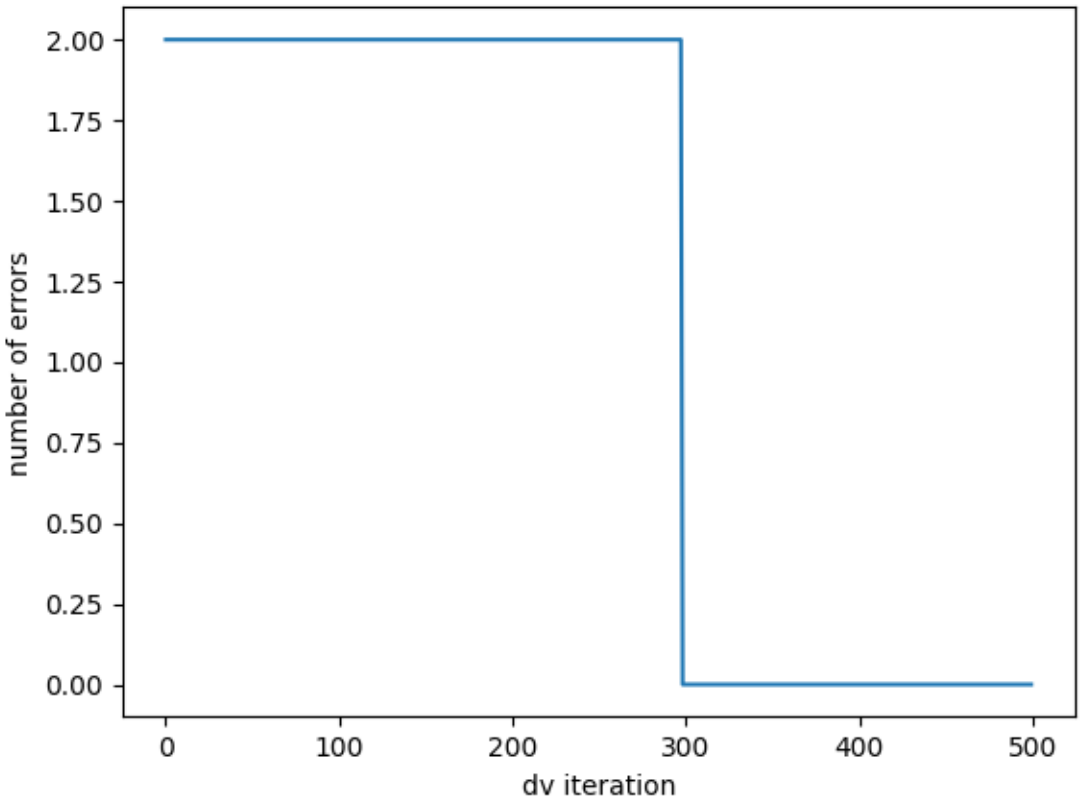

Figure 2: Evolution of the msgError during 500 iterations of the dv legacy algorithm

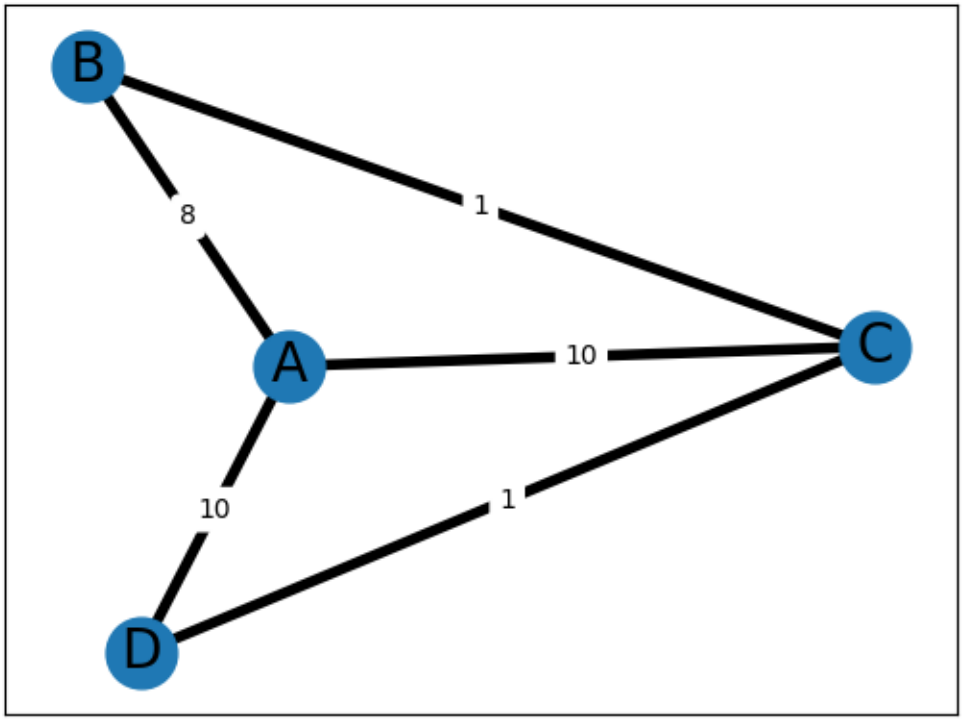

Figure 3: Another example graph named Béatrice

4 Conclusions

Author Contributions

Conceptualization: A.P., J.V.T.; Algorithm Implementation: A.P.; Interface Implementation: J.V.T.; Testing: A.P., J.V.T.; Experimentation and Result Analysis: A.P., J.V.T.; Documentation and Report: A.P., J.V.T.