ĐẠI HỌC CÔNG NGHỆ - ĐẠI HỌC QUỐC GIA HÀ NỘI
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO PROJECT CS-145: ROUTING

Tên môn học: Mạng máy tính
Mã lớp học: INT2213 6
Giảng viên giảng dạy: Hoàng Xuân Tùng
Giáo viên trợ giảng: Vũ Đức Trung
Nhóm: Nguyễn Đăng Đạo - 23021516
Đoàn Khánh Nhật - 23021652
Nguyễn Văn Thanh Tùng - 23021716
GitHub của nhóm: https://github.com/daovh123/CNP

HÀ NỘI - 2025

Mục lục

Phần 1

Giới thiệu bài toán

Trong một hệ thống mạng máy tính hiện đại, việc định tuyến các gói tin một cách hiệu quả và nhanh chóng là yếu tố then chốt đảm bảo chất lượng dịch vụ và độ tin cậy của toàn bộ mạng. Mỗi mạng nội bộ (Autonomous System – AS) hoạt động độc lập, bao gồm nhiều router cùng quản lý bởi một tổ chức, cần một thuật toán định tuyến nội bộ (intra-domain routing) để xác định con đường ngắn nhất (hoặc có chi phí thấp nhất) cho các gói tin đi từ nguồn đến đích.

Dự án này tập trung vào việc hiện thực hóa hai thuật toán định tuyến phân tán được sử dụng phổ biến trong thực tế: **thuật toán định tuyến vector khoảng cách (Distance-Vector)** và **thuật toán định tuyến trạng thái liên kết (Link-State)**. Mỗi thuật toán đều có cách tiếp cận riêng để xây dựng bảng định tuyến và phản ứng trước các sự kiện như sự cố đường truyền, thay đổi chi phí liên kết hoặc bổ sung router mới.

Cụ thể:

- Với thuật toán **Distance-Vector**, mỗi router chỉ biết chi phí tới các điểm đích thông qua các router láng giềng, và định kỳ trao đổi vector khoảng cách của mình với các láng giềng để cập nhật bảng định tuyến.

- Trong khi đó, thuật toán **Link-State** yêu cầu mỗi router nắm được sơ đồ toàn mạng thông qua việc thu thập trạng thái liên kết của tất cả các router khác. Sau đó, thuật toán Dijkstra sẽ được sử dụng để tính toán đường đi ngắn nhất.

Thông qua trình mô phỏng mạng được cung cấp, dự án cho phép sinh viên thực hiện mô phỏng hoạt động của các router trong một AS và kiểm tra hiệu quả của thuật toán được triển khai trong các kịch bản có thay đổi động như thêm hoặc gỡ bỏ liên kết.

Mục tiêu cuối cùng của dự án là đảm bảo rằng các thuật toán định tuyến được triển khai có thể tìm được đường đi tối ưu cho các gói tin trong nhiều tình huống khác nhau, phản ánh chính xác tính chất phân tán và thích nghi của các giao thức định

tuyến nội bộ trong thực tế.

Phần 2

Thuật toán

1. Thuật toán sử dụng:

Trong dự án này, mục tiêu của chúng ta là triển khai và kiểm tra hai thuật toán định tuyến trong miền nội bộ (intra-domain routing), cụ thể là **Distance-vector** và **Link-state**. Các thuật toán này được sử dụng để đảm bảo việc truyền tải gói tin trong một mạng cục bộ (LAN) hoặc mạng nội bộ (WAN), nơi các router phải hợp tác với nhau để chuyển tiếp gói tin một cách tối ưu.

Mặc dù cả hai thuật toán này đều có chung mục tiêu là tìm ra con đường ngắn nhất hoặc chi phí thấp nhất giữa các router, nhưng cách thức hoạt động và cách cập nhật thông tin giữa các router lại khác nhau. Dự án yêu cầu triển khai một trong hai thuật toán, nhưng nếu bạn triển khai cả hai, bạn sẽ nhận được điểm thưởng.

Trong báo cáo này, chúng ta sẽ đi sâu vào cách hoạt động của cả hai thuật toán và các điểm mạnh, yếu của từng thuật toán trong việc xử lý các thay đổi trong mạng như thêm hoặc xóa các liên kết, cũng như khả năng duy trì bảng định tuyến chính xác trong trường hợp có sự cố mạng.

2. Thuật toán Link-state

Link-state routing là một thuật toán định tuyến trong đó mỗi router duy trì thông tin về trạng thái liên kết của chính nó và của các router lân cận trong mạng. Thông tin này bao gồm các liên kết (links) mà router đó có với các router khác và chi phí (hoặc độ dài) của các liên kết đó.

2.1 Cách thức hoạt động:

Mỗi router xây dựng cơ sở dữ liệu trạng thái liên kết (LSDB - Link-State Database):

- Mỗi router sẽ thu thập thông tin về các liên kết của nó và thông báo

trạng thái liên kết này đến tất cả các router khác trong mạng. Thông báo này được gọi là **Link-State Advertisement (LSA)** và chứa thông tin về các liên kết (và chi phí của chúng) mà router đó có với các router lân cận.

- **Sử dụng Flooding:** Mỗi router sẽ phát tán LSA của mình cho tất cả các router khác trong mạng (thường là qua kỹ thuật flooding). Khi một router nhận được một LSA từ một router khác, nó sẽ cập nhật cơ sở dữ liệu trạng thái liên kết của mình và tiếp tục phát tán LSA này đến các router khác.

- **Tính toán đường đi ngắn nhất:** Sau khi nhận được tất cả thông tin trạng thái liên kết từ các router khác, mỗi router sẽ sử dụng một thuật toán tối ưu hóa, chẳng hạn như **Thuật toán Dijkstra**, để tính toán đường đi ngắn nhất (shortest path) đến tất cả các đích trong mạng dựa trên cơ sở dữ liệu trạng thái liên kết mà nó có.

- **Cập nhật định kỳ:** Các thông tin về trạng thái liên kết được cập nhật định kỳ và mỗi khi có sự thay đổi, như khi có một liên kết bị hỏng hoặc được thêm vào, router sẽ phát tán thông báo trạng thái liên kết mới đến các router còn lại.

2.2 Thuật toán dijkstra

Thuật toán Dijkstra là một thuật toán nổi tiếng dùng để tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trong đồ thị có trọng số không âm. Thuật toán này được phát minh bởi Edsger Dijkstra vào năm 1956 và được sử dụng rộng rãi trong các bài toán định tuyến và đồ thị.

Thuật toán Dijkstra là một trong những thuật toán cơ bản và quan trọng trong lý thuyết đồ thị và mạng máy tính. Nó giúp xác định tuyến đường ngắn nhất trong các mạng không có chu trình âm và rất hiệu quả trong các bài toán mạng máy tính, đặc biệt trong các giao thức định tuyến như **OSPF (Open Shortest Path First)**.

Cách thức hoạt động của Dijkstra: Thuật toán Dijkstra sử dụng phương pháp tham lam (greedy approach) để tìm kiếm đường đi ngắn nhất trong đồ thị. Thuật toán này bắt đầu từ đỉnh nguồn và tìm dần các đỉnh có khoảng cách ngắn nhất đến đỉnh nguồn. Mỗi lần một đỉnh được chọn, các khoảng cách đến các đỉnh lân cận của nó sẽ được cập nhật nếu có tuyến đường ngắn hơn.

Cụ thể, các bước thực hiện của thuật toán Dijkstra như sau:

- **Khởi tạo:**
    - Gán giá trị khoảng cách từ đỉnh nguồn đến chính nó là 0 và khoảng cách đến tất cả các đỉnh còn lại là vô cùng lớn (∞).

    - Đặt tất cả các đỉnh chưa được duyệt vào trong một tập hợp (hoặc hàng đợi ưu tiên), thường là **min-heap** hoặc **priority queue**, để đảm bảo chọn được đỉnh có khoảng cách ngắn nhất mỗi lần.

- **Lặp qua các đỉnh:**

    - Chọn đỉnh có khoảng cách ngắn nhất từ tập hợp các đỉnh chưa được duyệt (tức là đỉnh có giá trị nhỏ nhất trong hàng đợi).

    - Cập nhật khoảng cách cho các đỉnh lân cận của đỉnh đã chọn. Cụ thể, nếu khoảng cách từ đỉnh nguồn đến đỉnh lân cận qua đỉnh đã chọn nhỏ hơn khoảng cách hiện tại của đỉnh lân cận, thì cập nhật lại giá trị khoảng cách cho đỉnh lân cận đó.

    - Đánh dấu đỉnh đã được duyệt (hoặc xóa đỉnh khỏi hàng đợi ưu tiên).

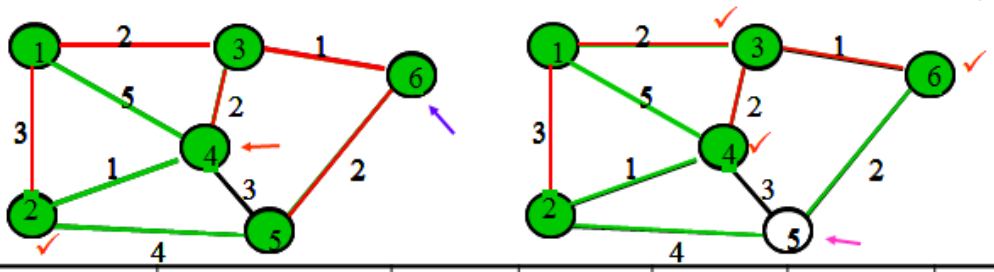
- **Lặp lại:**

    - Tiếp tục quá trình này cho đến khi tất cả các đỉnh đã được duyệt.

- **Kết thúc:**

    - Sau khi thuật toán hoàn thành, ta có được khoảng cách ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh còn lại trong đồ thị.

Execution of Dijkstra's algorithm



Iteration	N	D_2	D_3	D_4	D_5	D_6
Initial	{1}	3	2	5	∞	∞
1	{1,3}	3	2	4	∞	3
2	{1,2,3}	3	2	4	7	3
3	{1,2,3,6}	3	2	4	5	3
4	{1,2,3,4,6}	3	2	4	5	3
5	{1,2,3,4,5,6}	3	2	4	5	3

Trần Xuân Nam, Học viện KTQS

Chi tiết các bước thuật toán:

- **Bước 1:** Khởi tạo bảng khoảng cách, gán khoảng cách của đỉnh nguồn là 0 và khoảng cách của các đỉnh còn lại là vô cùng lớn (∞).
- **Bước 2:** Sử dụng một cấu trúc dữ liệu như **min-heap** (hoặc priority queue) để duyệt qua các đỉnh và tìm ra đỉnh có khoảng cách ngắn nhất. Sau đó, cập nhật khoảng cách cho các đỉnh lân cận.
- **Bước 3:** Lặp lại bước 2 cho đến khi tất cả các đỉnh đã được duyệt.
- **Bước 4:** Sau khi duyệt xong, bảng khoảng cách sẽ chứa khoảng cách ngắn nhất từ đỉnh nguồn đến tất cả các đỉnh còn lại.

Ưu điểm của thuật toán Dijkstra:

- **Hiệu quả trong đồ thị có trọng số không âm:** Dijkstra là thuật toán lý tưởng để tìm đường đi ngắn nhất trong đồ thị có trọng số không âm. Thuật toán hoạt động rất tốt trong các bài toán như định tuyến trong mạng máy tính, tìm kiếm đường đi trong các hệ thống giao thông, v.v.
- **Tính tham lam và hiệu quả:** Thuật toán Dijkstra hoạt động dựa trên nguyên lý tham lam, đảm bảo tìm được đường đi ngắn nhất ở mỗi bước. Điều này giúp thuật toán hoạt động rất nhanh khi đồ thị có nhiều đỉnh

và cạnh.

- **Có thể sử dụng với hàng đợi ưu tiên:** Việc sử dụng **min-heap** hoặc **priority queue** giúp giảm độ phức tạp của thuật toán Dijkstra xuống O((V + E) log V), trong đó V là số đỉnh và E là số cạnh. Điều này giúp thuật toán chạy hiệu quả trên các đồ thị lớn.

Nhược điểm của thuật toán Dijkstra:

- **Chỉ áp dụng cho đồ thị có trọng số không âm:** Dijkstra không thể xử lý đồ thị có trọng số âm. Nếu có cạnh nào trong đồ thị có trọng số âm, thuật toán sẽ không hoạt động chính xác.

- **Không phát hiện chu trình âm:** Nếu đồ thị có chu trình âm, thuật toán Dijkstra không thể phát hiện ra và sẽ dẫn đến kết quả sai. Để xử lý chu trình âm, cần phải sử dụng thuật toán khác như **Bellman-Ford**.

- **Yêu cầu bộ nhớ cao:** Đối với các đồ thị rất lớn, thuật toán Dijkstra có thể yêu cầu nhiều bộ nhớ và tài nguyên tính toán, đặc biệt nếu không sử dụng tối ưu các cấu trúc dữ liệu như **min-heap**.

Ứng dụng của thuật toán Dijkstra:

- **Định tuyến trong mạng máy tính (Routing):**
  Thuật toán Dijkstra là nền tảng cho các giao thức định tuyến trong mạng máy tính, chẳng hạn như **OSPF (Open Shortest Path First)** và **IS-IS (Intermediate System to Intermediate System)**. Các giao thức này sử dụng Dijkstra để tính toán đường đi ngắn nhất trong mạng, từ đó xác định được tuyến đường tối ưu để chuyển tiếp dữ liệu giữa các router.

- **Ứng dụng trong giao thông và bản đồ:**
  Thuật toán Dijkstra có thể được sử dụng để tìm đường đi ngắn nhất trên bản đồ, ví dụ như tính toán lộ trình lái xe tối ưu giữa hai địa điểm trong các hệ thống GPS. Các hệ thống này sử dụng Dijkstra để đưa ra các tuyến đường ngắn nhất, giúp tiết kiệm thời gian và nhiên liệu.

- **Tìm đường đi ngắn nhất trong các trò chơi:**
  Trong các trò chơi máy tính, đặc biệt là các trò chơi chiến lược hoặc game nhập vai (RPG), thuật toán Dijkstra có thể được sử dụng để tính toán đường đi ngắn nhất cho nhân vật trong bản đồ game.

- **Mạng điện thoại và viễn thông:**
  Thuật toán Dijkstra được sử dụng trong các hệ thống mạng viễn thông

để tìm đường đi ngắn nhất giữa các điểm trong mạng, từ đó tối ưu hóa việc chuyển tiếp các cuộc gọi hoặc truyền tải dữ liệu.

2.3. Ưu điểm

Khả năng phát hiện sự thay đổi nhanh chóng: Một trong những điểm mạnh của Link-state là khả năng phát hiện sự thay đổi trong mạng nhanh chóng và phản ứng lại kịp thời. Khi có sự thay đổi, mọi router trong mạng sẽ cập nhật ngay lập tức bảng định tuyến của mình.

Độ chính xác cao: Vì mỗi router có thông tin đầy đủ về toàn bộ mạng, thuật toán này giúp tính toán chính xác các tuyến đường ngắn nhất.

2.4. Nhược điểm

Chi phí tài nguyên cao: Mỗi router cần lưu trữ toàn bộ thông tin về trạng thái liên kết của tất cả các router trong mạng, điều này có thể tốn nhiều bộ nhớ và băng thông, đặc biệt là trong mạng lớn.

Chi phí tính toán lớn: Thuật toán Dijkstra cần tính toán các tuyến đường ngắn nhất, và điều này có thể tiêu tốn nhiều tài nguyên CPU trong các mạng có kích thước lớn.

3. Thuật toán Distance-vector

Distance-vector routing là một thuật toán định tuyến trong đó mỗi router duy trì một bảng định tuyến chứa các thông tin về khoảng cách đến các đích khác trong mạng. Các router định kỳ trao đổi thông tin này với các router lân cận để cập nhật bảng định tuyến của mình.

3.1 Cách thức hoạt động:

- **Bảng định tuyến:** Mỗi router duy trì một bảng định tuyến, trong đó ghi lại thông tin về các đích trong mạng và khoảng cách đến các đích đó. Mỗi router gửi bảng định tuyến của mình đến các router lân cận một cách định kỳ hoặc khi có sự thay đổi trong bảng định tuyến của nó.

- **Cập nhật bảng định tuyến:** Khi một router nhận được bảng định tuyến từ một router lân cận, nó sẽ so sánh các giá trị khoảng cách của mình với các giá trị nhận được từ router lân cận. Nếu giá trị khoảng cách mới cho

một đích là nhỏ hơn, router sẽ cập nhật bảng định tuyến của mình.

- **Lặp lại quá trình:** Các router tiếp tục chia sẻ bảng định tuyến của mình cho các router lân cận, và quá trình này tiếp tục cho đến khi tất cả các router trong mạng có bảng định tuyến chính xác. Tuy nhiên, quá trình này có thể mất thời gian và có thể gặp phải vấn đề **đếm tới vô hạn** (count-to-infinity), trong đó các router không thể điều chỉnh đúng bảng định tuyến khi một liên kết bị hỏng.
  Thuật toán Bellman-Ford

3.2 Thuật toán Bellman-Ford:

Thuật toán **Bellman-Ford** là một thuật toán tìm kiếm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trong đồ thị có trọng số, có thể chứa các trọng số âm. Thuật toán này có thể phát hiện ra chu trình âm trong đồ thị, điều mà một số thuật toán khác như **Dijkstra** không thể làm được.

Thuật toán Bellman-Ford có thể được sử dụng để giải quyết các vấn đề định tuyến trong các mạng máy tính, đặc biệt là khi có các liên kết với chi phí âm hoặc khi mạng có chu trình âm, như trong các bài toán về tính toán tuyến đường trong mạng tài chính hoặc các mạng phân phối chi phí.

Cách thức hoạt động của Bellman-Ford:

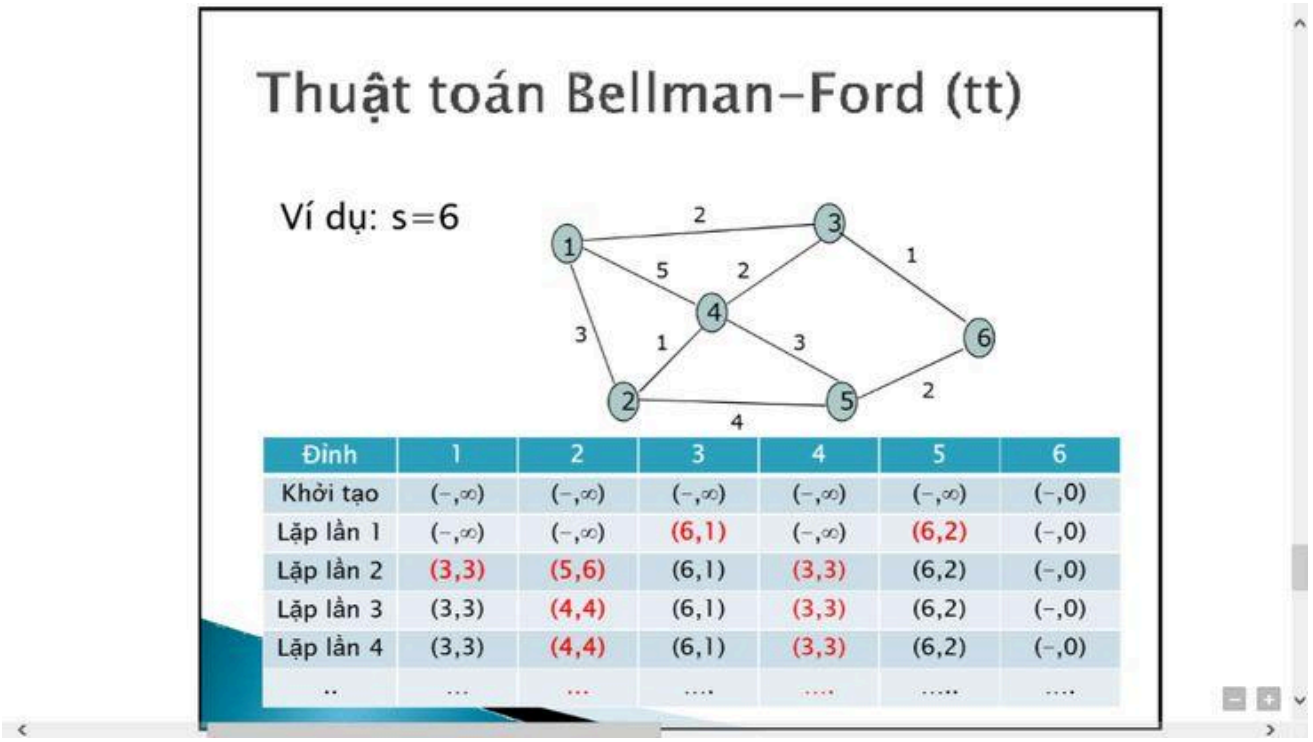
Thuật toán Bellman-Ford hoạt động bằng cách cập nhật các khoảng cách từ đỉnh nguồn đến các đỉnh còn lại trong đồ thị theo từng bước. Mỗi bước cập nhật các khoảng cách dựa trên việc xem xét các cạnh của đồ thị và kiểm tra xem liệu khoảng cách đến các đỉnh có thể được cải thiện hay không.

Cụ thể, thuật toán thực hiện như sau:

- **Khởi tạo:** Bắt đầu với một bảng khoảng cách, trong đó khoảng cách từ đỉnh nguồn đến chính nó là 0 và khoảng cách đến tất cả các đỉnh còn lại là vô cùng lớn (∞).

- **Lặp qua tất cả các cạnh:** Lặp lại bước sau đối với tất cả các cạnh của đồ thị (|V| - 1) lần, trong đó |V| là số lượng đỉnh trong đồ thị. Mỗi lần lặp, thuật toán kiểm tra tất cả các cạnh trong đồ thị và cập nhật khoảng cách đến các đỉnh đích của các cạnh nếu có con đường ngắn hơn.

- **Kiểm tra chu trình âm:** Sau khi lặp qua tất cả các cạnh (|V| - 1) lần, thuật toán tiến hành kiểm tra lại tất cả các cạnh một lần nữa. Nếu có cạnh nào mà khoảng cách đến đỉnh đích có thể được giảm xuống, điều này có nghĩa là đồ thị chứa chu trình âm. Nếu không có sự thay đổi nào trong vòng kiểm tra này, thuật toán sẽ dừng lại.



Đỉnh	1	2	3	4	5	6
Khởi tạo	(-,∞)	(-,∞)	(-,∞)	(-,∞)	(-,∞)	(-,0)
Lặp lần 1	(-,∞)	(-,∞)	(6,1)	(-,∞)	(6,2)	(-,0)
Lặp lần 2	(3,3)	(5,6)	(6,1)	(3,3)	(6,2)	(-,0)
Lặp lần 3	(3,3)	(4,4)	(6,1)	(3,3)	(6,2)	(-,0)
Lặp lần 4	(3,3)	(4,4)	(6,1)	(3,3)	(6,2)	(-,0)
..	...	...	....	....	.....	....

Chi tiết các bước thuật toán:

- Bước 1: Khởi tạo khoảng cách từ đỉnh nguồn đến tất cả các đỉnh khác là vô cùng lớn, ngoại trừ đỉnh nguồn có khoảng cách là 0.

- Bước 2: Lặp qua tất cả các cạnh trong đồ thị và kiểm tra xem liệu có thể giảm khoảng cách đến đích của các cạnh đó không. Nếu có, cập nhật khoảng cách mới.

- Bước 3: Lặp lại bước 2 tổng cộng |V| - 1 lần (với |V| là số đỉnh trong đồ thị).

- Bước 4: Kiểm tra một lần nữa tất cả các cạnh. Nếu có cạnh nào có thể giảm khoảng cách, điều này cho thấy đồ thị có chu trình âm.

Ưu điểm của thuật toán Bellman-Ford:

- **Có thể xử lý trọng số âm:** Đây là một trong những ưu điểm nổi bật của thuật toán Bellman-Ford. Thuật toán có thể tìm ra đường đi ngắn nhất trong đồ thị có trọng số âm, điều mà thuật toán **Dijkstra** không thể làm được.

- **Phát hiện chu trình âm:** Bellman-Ford có khả năng phát hiện chu trình âm trong đồ thị, điều này rất hữu ích trong các bài toán cần đảm bảo rằng không có chu trình âm trong đồ thị (ví dụ, trong các bài toán về tiền tệ hay mạng tài chính).

- **Đơn giản và dễ triển khai:** Thuật toán này có cấu trúc đơn giản và dễ triển khai, đặc biệt là trong các mạng nhỏ hoặc khi không yêu cầu hiệu suất tính toán quá cao.

Nhược điểm của thuật toán Bellman-Ford:

- **Hiệu suất thấp:** Thuật toán Bellman-Ford có độ phức tạp thời gian là O(|V| * |E|), trong đó |V| là số đỉnh và |E| là số cạnh trong đồ thị. Điều này có thể rất chậm khi làm việc với các đồ thị lớn, đặc biệt là khi so với thuật toán Dijkstra (O(|E| + |V| log |V|)) trong các đồ thị có trọng số không âm.

- **Không hiệu quả đối với đồ thị lớn:** Do độ phức tạp thời gian cao, Bellman-Ford không phải là lựa chọn tốt cho các bài toán có đồ thị lớn hoặc trong các mạng yêu cầu thời gian phản hồi nhanh.

Ứng dụng của Bellman-Ford:

Thuật toán Bellman-Ford được ứng dụng rộng rãi trong các lĩnh vực cần tính toán đường đi ngắn nhất trong đồ thị có trọng số âm, hoặc trong các bài toán đòi hỏi phải kiểm tra chu trình âm. Dưới đây là một số ứng dụng tiêu biểu của Bellman-Ford:

- **Định tuyến trong mạng máy tính:** Bellman-Ford có thể được sử dụng trong các giao thức định tuyến như **RIP (Routing Information Protocol)**. RIP sử dụng Bellman-Ford để tính toán các tuyến đường ngắn nhất trong mạng. Mặc dù có sự hạn chế về khả năng mở rộng trong các mạng lớn, nhưng Bellman-Ford vẫn là lựa chọn chính trong các mạng nhỏ và trung bình.

- **Mạng tài chính:** Trong các mạng tài chính, thuật toán Bellman-Ford có thể được dùng để phát hiện chu trình âm, điều này rất quan trọng trong các bài toán về sự ổn định của hệ thống tài chính, chẳng hạn như việc xác định xem một chu kỳ tiền tệ có thể sinh ra lợi nhuận vô hạn hay không.

- **Ứng dụng trong đồ thị có trọng số âm:** Các bài toán liên quan đến đồ thị có trọng số âm, chẳng hạn như tính toán chi phí thấp nhất trong các bài toán vận chuyển hoặc bài toán đồ thị có chu trình âm, có thể sử dụng Bellman-Ford để tìm ra các giải pháp chính xác.

3.3. Ưu điểm

- **Đơn giản và dễ triển khai:** Distance-vector có cấu trúc đơn giản và dễ triển khai hơn so với Link-state, vì mỗi router chỉ cần duy trì một bảng định tuyến và trao đổi với các router lân cận.

- **Ít yêu cầu tài nguyên:** Các router chỉ cần lưu trữ thông tin về các tuyến đường và khoảng cách đến các đích, không cần thông tin đầy đủ về toàn bộ mạng như trong Link-state.

3.4. Nhược điểm

- **Vấn đề đếm tới vô hạn (Count-to-infinity):** Một trong những vấn đề lớn của Distance-vector là khi có sự thay đổi lớn trong mạng (như liên kết bị hỏng), quá trình cập nhật định tuyến có thể mất rất nhiều thời gian để tất cả các router cập nhật đúng bảng định tuyến, dẫn đến việc sử dụng các tuyến đường không hợp lệ trong một khoảng thời gian dài.

- **Tốc độ cập nhật chậm:** Do phải chờ đợi sự trao đổi thông tin giữa các router, việc cập nhật bảng định tuyến có thể chậm chạp, đặc biệt trong mạng lớn.

- **Vấn đề về ổn định:** Việc cập nhật định kỳ và thiếu thông tin đầy đủ có thể dẫn đến sự không ổn định trong mạng, đặc biệt khi có sự cố hoặc thay đổi trong mạng.

Phần 3

Báo cáo bài làm

1. Môi trường và công cụ triển khai

Project được triển khai trên môi trường Python, sử dụng trình mô phỏng mạng do đề bài cung cấp. Các thành phần chính trong hệ thống gồm:

- DVrouter.py: hiện thực thuật toán định tuyến vector khoảng cách.
- LSrouter.py: hiện thực thuật toán định tuyến trạng thái liên kết.
- router.py, packet.py: lớp nền và định nghĩa gói tin.
- Tập tin .json: mô tả topology mạng và các sự kiện như thêm hoặc gỡ bỏ liên kết.

Việc chạy mô phỏng có thể thực hiện bằng hai cách:

- Có giao diện: python visualize_network.py <file>.json {DV,LS}
- Không giao diện: python network.py <file>.json {DV,LS}

2. Quá trình triển khai

2.1. File DVrouter.py

- Cập nhật liên kết mới (handle_new_link): Khi phát hiện hàng xóm mới, router cập nhật chi phí liên kết, cập nhật distance vector, và phát vector đến hàng xóm.

```
def handle_new_link(self, port, endpoint, cost):
    self.ports[endpoint] = port
    self.neighbors[endpoint] = cost

    # Khởi tạo distance vector của chính nó
    self.distance_vector[endpoint] = (cost, endpoint)
    self._recalculate_routes()
    self._broadcast_distance_vector()
```

- Xử lý gói định tuyến (handle_packet): Giải mã gói tin nhận được, lưu lại DV của hàng xóm và tính toán lại bảng định tuyến.

```
def handle_packet(self, port, packet):
    if packet.is_traceroute:
        dst = packet.dst_addr
        if dst in self.routing_table:
            self.send(self.routing_table[dst], packet)
    else:
        # Routing packet
        try:
            received = json.loads(packet.content)
            sender = packet.src_addr
            self.neighbor_vectors[sender] = received

            updated = self._recalculate_routes()
            if updated:
                self._broadcast_distance_vector()
        except Exception:
            return
```

- Xử lý gỡ bỏ liên kết (handle_remove_link): Xóa thông tin hàng xóm khỏi bảng vector và định tuyến. Gán chi phí vô cực cho các đích liên quan để tránh "routing loop".

```python
def handle_remove_link(self, port):
    neighbor = None
    for n, p in self.ports.items():
        if p == port:
            neighbor = n
            break

    if neighbor:
        del self.ports[neighbor]
        del self.neighbors[neighbor]
        self.neighbor_vectors.pop(neighbor, None)

        # Xoá các đường đi liên quan
        to_delete = []
        for dest, (cost, nhop) in self.distance_vector.items():
            if nhop == neighbor:
                to_delete.append(dest)
        for d in to_delete:
            self.distance_vector[d] = (INFINITY, None)

        self._recalculate_routes()
        self._broadcast_distance_vector()
```

- Gửi định kỳ (handle_time): Cứ mỗi heartbeat_time, router phát lại distance vector của mình.

```python
def handle_time(self, time_ms):
    if time_ms - self.last_time >= self.heartbeat_time:
        self.last_time = time_ms
        self._broadcast_distance_vector()
```

- Tính toán lại bảng định tuyến (_recalculate_routes): Tổng hợp tất cả các DV từ hàng xóm, áp dụng công thức Bellman-Ford để chọn đường đi có chi phí thấp nhất.

```python
def _recalculate_routes(self):
    updated = False
    new_dv = {}
    new_rt = {}

    all_dests = set(self.distance_vector.keys())
    for vec in self.neighbor_vectors.values():
        all_dests.update(vec.keys())
    all_dests.update(self.neighbors.keys())

    for dest in all_dests:
        if dest == self.addr:
            continue
        min_cost = INFINITY
        next_hop = None
        for neighbor in self.neighbors:
            link_cost = self.neighbors[neighbor]
            neighbor_vec = self.neighbor_vectors.get(neighbor, {})
            cost_to_dest = neighbor_vec.get(dest, INFINITY)
            total_cost = link_cost + cost_to_dest
            if total_cost < min_cost:
                min_cost = total_cost
                next_hop = neighbor

        if dest in self.neighbors and self.neighbors[dest] < min_cost:
            min_cost = self.neighbors[dest]
            next_hop = dest

        new_dv[dest] = (min_cost, next_hop)
        if next_hop in self.ports:
            new_rt[dest] = self.ports[next_hop]

    if new_dv != self.distance_vector:
        self.distance_vector = new_dv
        self.routing_table = new_rt
        updated = True

    return updated
```

2.2. File LSrouter.py

- Cập nhật liên kết mới (handle_new_link): Khi có liên kết mới, router tăng số thứ tự, cập nhật link-state database và phát bản tin trạng thái liên kết mới.

```python
def handle_new_link(self, port, endpoint, cost):
    """Handle new link."""
    self.ports[endpoint] = port
    self.neighbors[endpoint] = cost

    self.seq += 1
    self.link_state_db[self.addr] = (self.seq, self.neighbors.copy())
    self.seq_nums[self.addr] = self.seq

    self._update_forwarding_table()
    self._broadcast_link_state()
```

- Xử lý gói trạng thái liên kết (handle_packet): Nếu là bản mới (dựa trên sequence number), router lưu lại và lan truyền đến các hàng xóm khác. Sau đó tính lại bảng định tuyến.

```python
def handle_packet(self, port, packet):
    """Process incoming packet."""
    if packet.is_traceroute:
        dst = packet.dst_addr
        if dst in self.forwarding_table:
            out_port = self.forwarding_table[dst]
            self.send(out_port, packet)
    else:
        try:
            data = json.loads(packet.content)
            sender = data["router"]
            seq = data["seq"]
            neighbors = data["neighbors"]
        except (KeyError, json.JSONDecodeError):
            return  # Invalid packet format

        if sender in self.seq_nums and seq <= self.seq_nums[sender]:
            return  # Old update, discard

        self.link_state_db[sender] = (seq, neighbors)
        self.seq_nums[sender] = seq

        self._update_forwarding_table()

        for neighbor, out_port in self.ports.items():
            if out_port != port:
                fwd_pkt = Packet(Packet.ROUTING, self.addr, neighbor, content=packet.content)
                self.send(out_port, fwd_pkt)
```

- Xử lý gỡ bỏ liên kết (handle_remove_link): Cập nhật cơ sở dữ liệu và phát bản tin mới đến các router khác.

```python
def handle_remove_link(self, port):
    """Handle removed link."""
    neighbor = None
    for n, p in self.ports.items():
        if p == port:
            neighbor = n
            break

    if neighbor:
        del self.ports[neighbor]
        del self.neighbors[neighbor]

        self.seq += 1
        self.link_state_db[self.addr] = (self.seq, self.neighbors.copy
        self.seq_nums[self.addr] = self.seq

        self._update_forwarding_table()
        self._broadcast_link_state()
```

- Tính bảng định tuyến (_update_forwarding_table): Duyệt toàn bộ graph hiện tại bằng thuật toán Dijkstra (tự viết, không dùng thư viện) để xây dựng bảng định tuyến mới.

```python
def _update_forwarding_table(self):
    graph = {}
    for router, (_, neighbors) in self.link_state_db.items():
        graph[router] = neighbors.copy()

    dist = {self.addr: 0}
    prev = {}
    visited = set()
    heap = [(0, self.addr)]

    while heap:
        cost_u, u = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)
        for v, weight in graph.get(u, {}).items():
            alt = cost_u + weight
            if v not in dist or alt < dist[v]:
                dist[v] = alt
                prev[v] = u
                heapq.heappush(heap, (alt, v))

    new_table = {}
    for dest in dist:
        if dest == self.addr:
            continue
        next_hop = dest
        while prev[next_hop] != self.addr:
            next_hop = prev[next_hop]
        if next_hop in self.ports:
            new_table[dest] = self.ports[next_hop]

    self.forwarding_table = new_table
```

- Gửi định kỳ (handle_time): Định kỳ phát trạng thái liên kết đến các router khác.

```python
def handle_time(self, time_ms):
    """Handle current time."""
    if time_ms - self.last_time >= self.heartbeat_time:
        self.last_time = time_ms
        self._broadcast_link_state()
```

3. Kiểm thử và kết quả

3.1. Phương pháp kiểm thử

- Sử dụng tập lệnh test_scripts/test_dv_ls.sh để chạy hàng loạt kịch bản mô phỏng có sẵn.
- Kiểm thử cả hai chế độ (có và không có sự kiện mạng).
- Kiểm tra tính đúng đắn của bảng định tuyến cuối cùng bằng đánh giá của trình mô phỏng.

3.2. Kết quả đạt được

Tên mô phỏng	DVrouter	LSrouter	Kết quả
01_small_net.json	✅	✅	All Routes correct!
02_small_net_events.json	✅	✅	All Routes correct!
03_pg244_net.json	✅	✅	All Routes correct!
04_pg244_net_events.json	✅	✅	All Routes correct!
05_pg242_net.json	✅	✅	All Routes correct!
06_pg242_net_events.json	✅	✅	All Routes correct!

Cả hai thuật toán đều xử lý đúng các trường hợp thay đổi liên kết, tính được đường đi có chi phí thấp nhất, không gây lặp hay treo mạng.

4. Đánh giá triển khai

- Tính đúng đắn: Các đường đi được tính toán đều khớp với đường ngắn nhất theo đánh giá của mô phỏng.
- Tính thích nghi: Các router phản ứng kịp thời khi có thêm hoặc mất liên kết.
- Tuân thủ yêu cầu: Không truy cập hàm/biến bị hạn chế, không thay đổi các file ngoài DVrouter.py và LSrouter.py.

Phần 4

Tổng kết

Trong dự án này, chúng tôi đã triển khai hai thuật toán định tuyến trong miền nội bộ (intra-domain routing), đó là **Distance-Vector** và **Link-State**. Các thuật toán này nhằm mục đích tối ưu hóa quá trình chuyển tiếp gói tin trong mạng bằng cách xác định đường đi ngắn nhất hoặc đường đi có chi phí thấp nhất giữa các router trong một mạng tự trị (AS).

1. Quá trình triển khai

- **Thuật toán Distance-Vector:**
  Thuật toán **Distance-Vector** sử dụng thuật toán **Bellman-Ford** để tìm ra đường đi ngắn nhất trong mạng. Mỗi router trong thuật toán **Distance-Vector** sẽ duy trì một bảng định tuyến, chứa khoảng cách đến các đích trong mạng. Các router sẽ trao đổi bảng định tuyến của mình với các router lân cận định kỳ. Khi một router nhận được thông tin mới từ các router lân cận, nó sẽ cập nhật bảng định tuyến của mình bằng cách sử dụng thuật toán **Bellman-Ford**, từ đó tính toán lại đường đi ngắn nhất. Thuật toán **Bellman-Ford** sẽ kiểm tra và cập nhật khoảng cách từ router đến tất cả các đích, nếu có một con đường ngắn hơn. Thuật toán **Distance-Vector** sẽ lặp lại quá trình này, cập nhật định kỳ bảng định tuyến của mình và gửi thông tin tới các router lân cận. Chúng tôi cũng xử lý vấn đề **count-to-infinity** trong quá trình triển khai thuật toán, sử dụng các kỹ thuật như **split horizon** và **poison reverse** để giảm thiểu tình trạng không ổn định trong mạng.

- **Thuật toán Link-State**:
  Thuật toán **Link-State** yêu cầu mỗi router duy trì cơ sở dữ liệu trạng thái liên kết (LSDB) của chính nó và của các router lân cận. Mỗi khi có thay đổi trong mạng, router sẽ gửi các **Link-State Advertisements (LSA)** để cập nhật thông tin trạng thái liên kết cho các router khác. Thuật toán sử dụng **Thuật toán Dijkstra** để tính toán đường đi ngắn nhất từ router đến tất cả các đích trong mạng. Việc triển khai **Link-State** của chúng tôi đảm bảo rằng mỗi router có thể tính toán và cập nhật bảng định tuyến của mình dựa trên trạng thái liên kết mới nhất. Thuật

toán **Dijkstra** được sử dụng trong thuật toán **Link-State** để tìm kiếm đường đi ngắn nhất từ router đến tất cả các đích trong mạng. Chúng tôi đã sử dụng **min-heap** (priority queue) để tối ưu hóa quá trình tìm kiếm đường đi ngắn nhất và cập nhật bảng định tuyến.

2. Kết quả thử nghiệm

Sau khi triển khai xong các thuật toán, chúng tôi đã tiến hành thử nghiệm với nhiều mô phỏng mạng khác nhau. Các thuật toán đã hoạt động chính xác trong việc tìm kiếm đường đi ngắn nhất và tối ưu hóa định tuyến trong các mạng có thay đổi liên kết và thất bại của router. Thuật toán **Link-State** cho kết quả chính xác hơn và nhanh chóng phản hồi khi có thay đổi trong mạng, trong khi thuật toán **Distance-Vector** dựa trên **Bellman-Ford** gặp phải độ trễ trong việc cập nhật thông tin định tuyến khi mạng có sự thay đổi.

Chúng tôi đã sử dụng bộ mô phỏng mạng cung cấp sẵn để kiểm tra các trường hợp với **link failures** và **link additions**. Kết quả thử nghiệm cho thấy các thuật toán đều có thể tính toán lại bảng định tuyến và xác định đường đi chính xác trong các tình huống này.

3. Những khó khăn và thách thức

Trong quá trình triển khai, chúng tôi đã gặp phải một số khó khăn, bao gồm:

- **Quản lý thay đổi mạng**: Việc cập nhật bảng định tuyến khi có sự thay đổi liên kết hoặc thất bại của router yêu cầu xử lý chính xác và nhanh chóng. Với **Distance-Vector**, việc đảm bảo không có vòng lặp vô hạn (loop) trong bảng định tuyến là một thử thách lớn, và việc sử dụng các kỹ thuật như **split horizon** và **poison reverse** là rất quan trọng.

- **Cập nhật trạng thái liên kết trong Link-State**: Việc gửi và nhận **Link-State Advertisements (LSA)** trong một mạng lớn có thể gây tắc nghẽn, đặc biệt nếu không kiểm soát tốt quá trình **flooding**. Do đó, việc tối ưu hóa và giới hạn các thông báo được gửi trong mạng là một yếu tố quan trọng để tránh tình trạng tắc nghẽn.

- **Sử dụng thuật toán Dijkstra**: Thuật toán **Dijkstra** yêu cầu sử dụng các cấu trúc dữ liệu như **priority queue** (min-heap) để tối ưu hóa quá trình tìm kiếm đường đi ngắn nhất. Tuy nhiên, việc triển khai Dijkstra trong môi trường phân tán đòi hỏi phải xử lý nhiều thách thức về đồng bộ hóa và hiệu suất.

4. Kết luận

Dự án đã giúp chúng tôi hiểu rõ hơn về các thuật toán định tuyến phân tán như **Distance-Vector** và **Link-State**, cũng như cách thức hoạt động của chúng trong các mạng máy tính. Cả hai thuật toán đều có những ưu điểm và nhược điểm riêng. **Distance-Vector** dựa trên thuật toán **Bellman-Ford**, đơn giản và dễ triển khai nhưng gặp phải vấn đề về độ ổn định khi có sự thay đổi trong mạng. **Link-State** cung cấp sự chính xác và phản hồi nhanh chóng hơn trong các mạng lớn nhưng yêu cầu tài nguyên tính toán và bộ nhớ cao hơn.

Qua quá trình thử nghiệm, chúng tôi đã cải thiện khả năng xử lý thay đổi trong mạng và tối ưu hóa việc tính toán đường đi ngắn nhất. Những kỹ thuật và công cụ học được trong dự án này sẽ rất hữu ích trong việc phát triển các hệ thống mạng lớn và nâng cao hiệu quả định tuyến trong các môi trường mạng phân tán.

Phần 5

Trích dẫn

Nguồn tham khảo:

1. Harvard CS145 – Routing on GitHub
2. GeeksforGeeks – Unicast Routing & Link State Routing
3. GeeksforGeeks – Distance Vector Routing (DVR) Protocol
4. GeeksforGeeks – Dijkstra's Shortest Path Algorithm
5. GeeksforGeeks – Bellman-Ford Algorithm