Handout Buổi 11: Thuật toán Dijkstra và ứng dụng

🎯 Mục tiêu đạt được buổi 1:

- Nhận diện và mô hình hóa bài toán đường đi ngắn nhất trên đồ thị có trọng số.
- Hiểu và thực hiện thuật toán Dijkstra theo từng bước truy vết.
- Biết cài đặt Dijkstra bằng Python với danh sách kề và hàng đợi ưu tiên.
- Phân biệt trường hợp áp dụng và không áp dụng được Dijkstra (như cạnh âm).
- Liên hệ ứng dụng Dijkstra vào định tuyến bản đồ và tình huống thực tế.

PHẦN 1: LÝ THUYẾT

1.1. Giới thiệu bài toán tối ưu trên đồ thị có trọng số

Trong lĩnh vực khoa học máy tính và công nghệ thông tin, đồ thị là một mô hình trừu tượng được sử dụng để biểu diễn các mối quan hệ và kết nối giữa các thực thể. Khi các cạnh trong đồ thị được gán một giá trị đại diện cho chi phí, khoảng cách, thời gian hoặc năng lượng, ta gọi đó là đồ thị có trọng số (*weighted graph*).

Một **đồ thị có trọng số** là một cấu trúc bao gồm tập các đỉnh (hoặc nút) và tập các cạnh có hướng hoặc vô hướng, trong đó mỗi cạnh được gắn với một giá trị số thực gọi là trọng số. Trong ngữ cảnh của các bài toán tối ưu, trọng số thường biểu diễn "chi phí" để đi từ một đỉnh này đến đỉnh khác, chẳng hạn như độ dài quãng đường, thời gian truyền tin, chi phí vận chuyển, hoặc các đại lượng tương tự.

Việc nghiên cứu và giải quyết các bài toán tối ưu trên đồ thị có trọng số đóng vai trò quan trọng trong nhiều ứng dụng thực tế và công nghiệp. Dưới đây là một số ví dụ điển hình:

- **Tìm đường đi ngắn nhất trong bản đồ giao thông**: Đây là ứng dụng trực tiếp của bài toán tìm đường đi ngắn nhất từ một vị trí đến vị trí khác, dựa trên khoảng cách hoặc thời gian di chuyển.
- **Định tuyến gói tin trong mạng máy tính**: Trong các hệ thống mạng, cần xác định con đường tối ưu để truyền dữ liệu từ máy phát đến máy thu với độ trễ hoặc chi phí thấp nhất.
- **Quản lý vận tải và hậu cần**: Các công ty vận chuyển cần tối ưu hóa lộ trình giao hàng nhằm tiết kiệm nhiên liệu và thời gian.
- **Thiết kế mạch tích hợp**: Trong thiết kế vi mạch, việc tối ưu hóa đường đi giữa các thành phần cũng có thể được mô hình hóa như một bài toán trên đồ thị có trọng số.

Từ các ứng dụng đó, nhiều bài toán tối ưu đã được xác định và nghiên cứu sâu rộng. Một số bài toán điển hình bao gồm:

1. **Tìm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh còn lại**: Đây là bài toán cơ bản nhất, thường được giải bằng thuật toán Dijkstra nếu trọng số không âm, hoặc Bellman-Ford nếu tồn tại trọng số âm.

2. **Tìm đường đi ngắn nhất giữa hai đỉnh bất kỳ**: Là trường hợp đặc biệt của bài toán trên, trong đó chỉ quan tâm đến một cặp đỉnh cụ thể.

3. **Tìm đường đi ngắn nhất giữa mọi cặp đỉnh**: Bài toán này thường được giải bằng thuật toán Floyd-Warshall hoặc cải tiến Dijkstra áp dụng cho từng đỉnh nguồn.

4. **Bài toán luồng cực đại**: Xác định luồng tối đa có thể truyền từ một đỉnh nguồn đến đỉnh đích trong một mạng có giới hạn dung lượng trên các cạnh. Bài toán này được giải bằng các thuật toán như Ford-Fulkerson hoặc Edmonds-Karp.

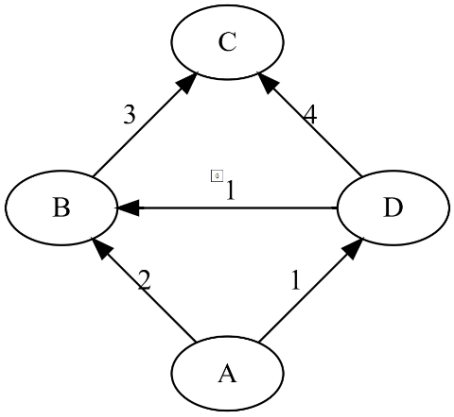
Các bài toán nêu trên không chỉ mang ý nghĩa học thuật mà còn là nền tảng cho nhiều hệ thống phần mềm thực tế. Việc hiểu rõ bản chất, cách mô hình hóa và các phương pháp giải các bài toán tối ưu trên đồ thị có trọng số là mục tiêu then chốt của nội dung học phần này.

1.2. Biểu diễn đồ thị có trọng số

Để giải quyết các bài toán tối ưu trên đồ thị có trọng số, việc lựa chọn cách biểu diễn đồ thị phù hợp là yếu tố quan trọng, ảnh hưởng trực tiếp đến độ phức tạp và hiệu quả của thuật toán được áp dụng. Trong thực tế, hai phương pháp phổ biến nhất được sử dụng là **ma trận kề có trọng số** và **danh sách kề có trọng số**.

a) Ma trận kề có trọng số

Ma trận kề là một ma trận vuông có kích thước $n \times n$, trong đó n là số lượng đỉnh của đồ thị. Mỗi phần tử tại vị trí (i,j) trong ma trận đại diện cho trọng số của cạnh nối từ đỉnh i đến đỉnh j. Nếu không tồn tại cạnh nối giữa hai đỉnh này, giá trị tại ô đó sẽ được gán bằng một giá trị vô cực (ký hiệu là ∞) để biểu diễn "không có cạnh".



Ví dụ: Giả sử ta có đồ thị có 4 đỉnh (A, B, C, D) và các cạnh như sau:

- A → B trọng số 2
- A → D trọng số 1
- B → C trọng số 3
- D → B trọng số 1
- D → C trọng số 4

Ta có thể biểu diễn đồ thị này bằng ma trận kề như sau:

	A	B	C	D
A	0	2	∞	1
B	∞	0	3	∞
C	∞	∞	0	∞
D	∞	1	4	0

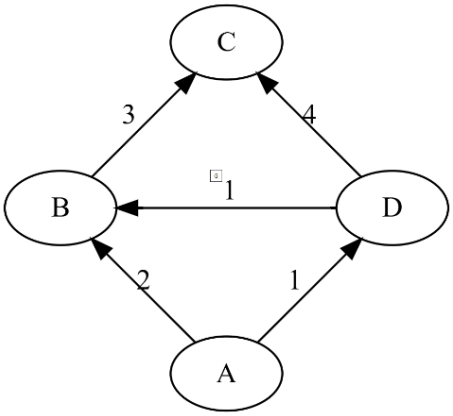
Giá trị tại đường chéo chính thường gán là 0, vì khoảng cách từ một đỉnh đến chính nó là 0.

b) Danh sách kề có trọng số

Danh sách kề là cấu trúc dữ liệu trong đó mỗi đỉnh được ánh xạ đến một danh sách các đỉnh kề của nó. Đối với đồ thị có trọng số, mỗi phần tử trong danh sách kề còn kèm theo trọng số của cạnh tương ứng.

Ví dụ: Với đồ thị như trên, danh sách kề có trọng số được viết như sau:

A: (B, 2), (D, 1)
B: (C, 3)
C:
D: (B, 1), (C, 4)



Danh sách này cho thấy từ đỉnh A có thể đi đến B với trọng số 2 và đến D với trọng số 1; từ D có thể đi đến B và C với các trọng số tương ứng.

c) So sánh hai phương pháp

Tiêu chí	Ma trận kề	Danh sách kề
Dễ cài đặt	Cao	Trung bình
Không gian lưu trữ	$O(n^2)$	$O(n+m)$ với m là số cạnh
Thích hợp cho	Đồ thị dày (dense)	Đồ thị thưa (sparse)
Truy cập cạnh (i, j)	$O(1)$	$O(k)$ với k là số đỉnh kề

Việc lựa chọn biểu diễn nào sẽ phụ thuộc vào đặc trưng của đồ thị và yêu cầu về thời gian xử lý của thuật toán áp dụng.

1.3. Ma trận khoảng cách

Trong các thuật toán tìm đường đi ngắn nhất trên đồ thị có trọng số, một cấu trúc dữ liệu trung tâm thường được sử dụng là **ma trận khoảng cách** (*distance matrix*). Đây là một công cụ quan trọng để lưu trữ và cập nhật liên tục khoảng cách ngắn nhất từ một đỉnh nguồn đến các đỉnh còn lại, hoặc giữa mọi cặp đỉnh trong đồ thị.

a) Khái niệm

Ma trận khoảng cách là một ma trận hai chiều, trong đó mỗi phần tử tại vị trí hàng i, cột j, ký hiệu là $d[i][j]$, biểu diễn độ dài đường đi ngắn nhất từ đỉnh i đến đỉnh j trên đồ thị.

Tùy thuộc vào loại bài toán, cách khởi tạo và cập nhật ma trận có thể khác nhau:

- **Với thuật toán Dijkstra**: Ma trận (hoặc mảng) khoảng cách thường được khởi tạo để tính khoảng cách ngắn nhất từ một đỉnh nguồn duy nhất đến tất cả các đỉnh còn lại.
- **Với thuật toán Floyd-Warshall**: Ma trận được sử dụng để lưu trữ khoảng cách giữa mọi cặp đỉnh, với mục tiêu tính toán toàn bộ tập các đường đi ngắn nhất trong đồ thị.

b) Vai trò của ma trận khoảng cách

Việc sử dụng ma trận khoảng cách giúp các thuật toán tìm đường đi ngắn nhất có thể:

- Lưu trữ tạm thời kết quả của các bước tính toán trung gian.
- Cập nhật linh hoạt các khoảng cách trong quá trình "nới lỏng cạnh" (relaxation).
- Truy xuất nhanh chóng khoảng cách giữa hai đỉnh bất kỳ.
- Phục vụ cho các thuật toán có tính chất lặp như Floyd-Warshall, nơi mỗi vòng lặp cập nhật toàn bộ ma trận dựa trên đỉnh trung gian.

Cụ thể, trong thuật toán Dijkstra, mảng khoảng cách giúp đảm bảo rằng tại mỗi bước, ta luôn chọn được đỉnh chưa xét có khoảng cách nhỏ nhất để tiếp tục mở rộng.

c) Khởi tạo ban đầu

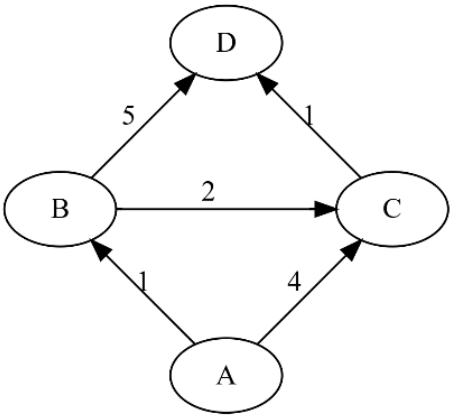
Việc khởi tạo chính xác ma trận khoảng cách là bước khởi đầu quan trọng trong mọi thuật toán tìm đường đi ngắn nhất. Thông thường, ta thực hiện khởi tạo như sau:

- Với thuật toán Dijkstra:
  - Gán distance$[u] = 0$, trong đó u là đỉnh nguồn.
  - Với mọi đỉnh khác $v \neq u$, gán distance$[v] = \infty$ để biểu thị rằng chưa biết khoảng cách từ u đến v.
- Với thuật toán Floyd-Warshall:
  - Nếu $i = j$, gán $d[i][j]$=0 vì khoảng cách từ một đỉnh đến chính nó bằng 0.
  - Nếu tồn tại cạnh từ i đến j, gán $d[i][j]$=trọng số cạnh(i,j).
  - Nếu không tồn tại cạnh, gán $d[i][j]$=∞.

d) Ví dụ minh họa

Xét đồ thị sau đây với 4 đỉnh A, B, C, D và các cạnh có trọng số như sau:

- A → B (trọng số 1)
- A → C (trọng số 4)
- B → C (trọng số 2)
- C → D (trọng số 1)
- B → D (trọng số 5)



Giả sử ta khởi tạo mảng khoảng cách từ đỉnh A (với Dijkstra):

distance[A] = 0
distance[B] = ∞
distance[C] = ∞
distance[D] = ∞

Sau các bước cập nhật trong thuật toán, giá trị distance sẽ dần dần được cập nhật đến giá trị khoảng cách ngắn nhất thực tế.

1.4. Thuật toán Dijkstra

Thuật toán Dijkstra, do Edsger W. Dijkstra đề xuất vào năm 1959, là một trong những thuật toán cơ bản và hiệu quả nhất để giải bài toán tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh còn lại trong một đồ thị có trọng số **không âm**. Thuật toán sử dụng kỹ thuật tham lam (*greedy algorithm*) để đảm bảo rằng tại mỗi bước, nó luôn chọn đỉnh gần nhất chưa được xét đến.

a) Mục tiêu của thuật toán

Mục tiêu của thuật toán Dijkstra là tìm khoảng cách ngắn nhất từ một đỉnh cho trước (gọi là đỉnh nguồn) đến tất cả các đỉnh còn lại trong đồ thị có trọng số không âm. Kết quả của thuật toán là một bảng (hoặc mảng) lưu trữ khoảng cách tối thiểu từ đỉnh nguồn đến từng đỉnh khác, đồng thời có thể lưu lại đường đi để tái dựng lộ trình cụ thể.

b) Nguyên lý hoạt động

Thuật toán Dijkstra hoạt động theo ba bước chính:

1. **Khởi tạo**: Gán khoảng cách từ đỉnh nguồn đến chính nó là 0 và đến tất cả các đỉnh còn lại là ∞ (vô cực). Tập đỉnh đã duyệt ban đầu là rỗng.

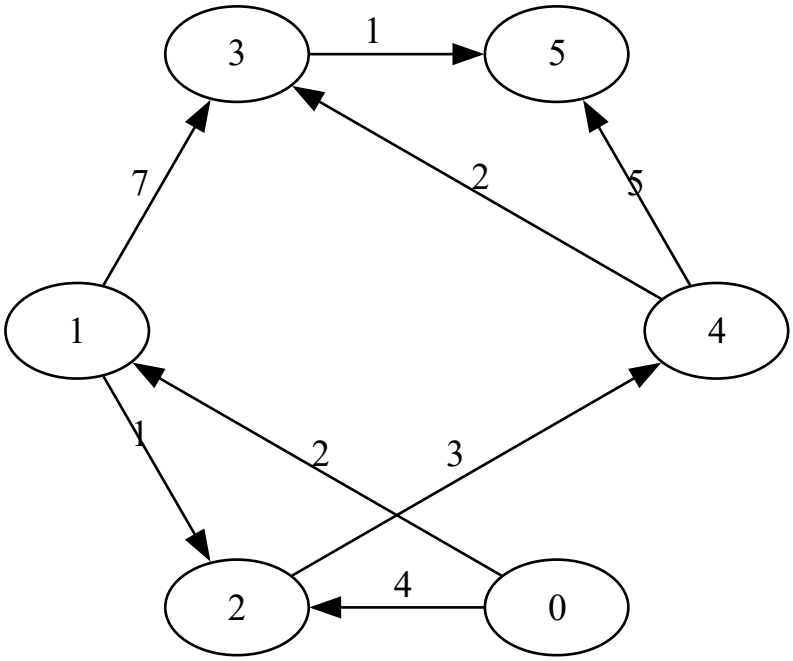
2. **Chọn đỉnh có khoảng cách ngắn nhất chưa được duyệt**: Ở mỗi vòng lặp, chọn đỉnh chưa xét có giá trị khoảng cách hiện tại nhỏ nhất để mở rộng.

3. **Cập nhật khoảng cách các đỉnh kề (relaxation)**: Với mỗi đỉnh kề của đỉnh đang xét, nếu khoảng cách mới thông qua đỉnh này nhỏ hơn khoảng cách đã biết trước đó, thì cập nhật lại giá trị khoảng cách.

Thuật toán tiếp tục lặp lại cho đến khi tất cả các đỉnh đã được duyệt, hoặc khoảng cách ngắn nhất đến các đỉnh còn lại không thể cải thiện thêm (ví dụ, nếu đồ thị không liên thông).

c) Cấu trúc thuật toán

Để tối ưu hóa hiệu quả của việc lựa chọn đỉnh có khoảng cách nhỏ nhất, thuật toán Dijkstra thường sử dụng **hàng đợi ưu tiên** (*priority queue*) với cấu trúc **heap**. Việc này giúp giảm thời gian tìm đỉnh có khoảng cách nhỏ nhất từ $O(V)$ xuống $O(\log V)$, trong đó V là số đỉnh.

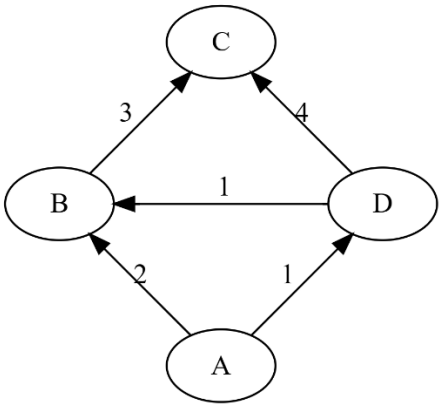


Cấu trúc dữ liệu chính:

- Mảng distance[] lưu khoảng cách ngắn nhất tạm thời từ nguồn đến mỗi đỉnh.
- Mảng visited[] đánh dấu đỉnh đã được xét.
- Cấu trúc heap để ưu tiên đỉnh có khoảng cách nhỏ nhất chưa được duyệt.

d) Điều kiện áp dụng

Thuật toán Dijkstra **chỉ áp dụng được cho đồ thị mà tất cả các trọng số cạnh đều không âm**. Lý do là vì bản chất của thuật toán là luôn chọn đỉnh gần nhất để mở rộng, giả định rằng các đường đi sau không thể làm giảm khoảng cách đã tìm được. Khi có cạnh âm, giả định này không còn đúng và thuật toán có thể đưa ra kết quả sai. Trong các trường hợp có cạnh âm, thuật toán **Bellman-Ford** là phương án thay thế phù hợp hơn.



e) Ví dụ minh họa

Giả sử ta có đồ thị sau với các cạnh có trọng số không âm:

- A → B (trọng số 2)
- A → D (trọng số 1)
- B → C (trọng số 3)
- D → B (trọng số 1)
- D → C (trọng số 4)

Khởi tạo ban đầu khi chạy thuật toán Dijkstra từ đỉnh A:

distance[A] = 0
distance[B] = ∞
distance[C] = ∞
distance[D] = ∞

Qua từng bước cập nhật (relaxation), ta sẽ dần thu được khoảng cách ngắn nhất đến từng đỉnh, đồng thời có thể xây dựng lại lộ trình đi từ A đến các đỉnh khác thông qua mảng truy vết.

1.5. Mã giả thuật toán Dijkstra

a) Mã giả thuật toán

Dưới đây là dạng mã giả tiêu chuẩn của thuật toán Dijkstra áp dụng cho đồ thị có trọng số không âm, được biểu diễn dưới dạng danh sách kề:

```
Input:

   G = (V, E)    // đồ thị có trọng số không âm
   w(u, v)       // trọng số của cạnh (u, v)
   s             // đỉnh nguồn
Output:
   - distance[v]: khoảng cách ngắn nhất từ s đến v, với mọi v ∈ V
   - mảng previous

1.  for each vertex v in V:
2.      distance[v] ← ∞
3.      previous[v] ← null
4.  distance[s] ← 0
5.  Q ← priority queue chứa tất cả các đỉnh v ∈ V, ưu tiên theo distance[v]

6.  while Q is not empty:
7.      u ← đỉnh trong Q có distance[u] nhỏ nhất
8.      remove u khỏi Q

9.      for each neighbor v of u:
10.         alt ← distance[u] + w(u, v)
11.         if alt < distance[v]:
12.             distance[v] ← alt
```

```
13.         previous[v] ← u
14.         cập nhật thứ tự ưu tiên của v trong Q
```

b) Giải thích từng bước

- **Bước 1–4**: Khởi tạo mảng distance với giá trị vô cực (∞) cho tất cả các đỉnh, trừ đỉnh nguồn s có khoảng cách bằng 0. Mảng previous được dùng để lưu lại đường đi ngắn nhất.
- **Bước 5**: Tạo hàng đợi ưu tiên Q chứa tất cả các đỉnh, được sắp xếp theo giá trị distance. Đỉnh có khoảng cách nhỏ nhất sẽ được ưu tiên lấy ra trước.
- **Bước 6–8**: Lặp cho đến khi hàng đợi Q rỗng. Tại mỗi vòng lặp, chọn đỉnh u có distance[u] nhỏ nhất để xử lý, sau đó loại u khỏi hàng đợi.
- **Bước 9–13**: Duyệt tất cả các đỉnh kề v của đỉnh u. Tính toán khoảng cách mới alt = distance[u] + w(u, v). Nếu alt nhỏ hơn distance[v] hiện tại, thì cập nhật distance[v] bằng alt và lưu lại đỉnh trước đó u để phục vụ việc tái dựng đường đi.
- **Bước 14**: Cập nhật lại vị trí của đỉnh v trong hàng đợi ưu tiên Q do giá trị ưu tiên đã thay đổi (giảm xuống).

c) Cơ chế cập nhật (relaxation)

Quá trình cập nhật khoảng cách (relaxation) là trung tâm của thuật toán. Tại mỗi lần xét đỉnh u, ta duyệt tất cả các đỉnh kề v và kiểm tra xem liệu đường đi qua u có ngắn hơn đường đi hiện tại đến v hay không. Nếu có, ta cập nhật giá trị mới và tiếp tục duyệt.

Công thức cập nhật:

```
Nếu distance[u] + w(u, v) < distance[v]
⇒ cập nhật distance[v] ← distance[u] + w(u, v)
  cập nhật previous[v] ← u
```

Cơ chế này đảm bảo rằng tại thời điểm một đỉnh được duyệt, giá trị distance của nó là tối ưu (ngắn nhất có thể).

d) Điều kiện dừng

Thuật toán dừng khi tất cả các đỉnh trong hàng đợi Q đã được duyệt, nghĩa là:

- Toàn bộ các khoảng cách ngắn nhất từ đỉnh nguồn đến các đỉnh còn lại đã được xác định.
- Hoặc, nếu tồn tại đỉnh không liên thông với đỉnh nguồn, thì khoảng cách của nó vẫn giữ nguyên giá trị ∞.

Trường hợp chỉ cần tìm đường đi đến một đỉnh đích cụ thể, ta có thể dừng thuật toán ngay khi đỉnh đó được đưa ra khỏi hàng đợi (tức là khi tìm được đường đi ngắn nhất đến đích).

1.6. Độ phức tạp thuật toán

Thuật toán Dijkstra có thể được hiện thực với nhiều cách khác nhau, và độ phức tạp thời gian của thuật toán phụ thuộc đáng kể vào cách chọn cấu trúc dữ liệu dùng để quản lý hàng đợi ưu tiên (*priority queue*). Trong mục này, chúng ta sẽ phân tích hai trường hợp phổ biến nhất và so sánh với thuật toán Bellman-Ford.

a) Trường hợp dùng mảng thường (array-based implementation)

Trong cách cài đặt đơn giản nhất, ta sử dụng một mảng để lưu trữ khoảng cách và lựa chọn thủ công đỉnh có khoảng cách nhỏ nhất trong tập đỉnh chưa được duyệt.

- Việc chọn đỉnh có khoảng cách nhỏ nhất trong mỗi vòng lặp (bước 7) cần duyệt qua toàn bộ mảng: mất $O(V)$.
- Việc cập nhật các đỉnh kề thực hiện tối đa E lần, mỗi lần với chi phí $O(1)$.

⇒ Tổng độ phức tạp thời gian:

$$O(V^2+E) = O(V^2) \text{ (vì } E \leq V \text{ trong đồ thị đơn)}$$

Cách hiện thực này phù hợp cho đồ thị **dày đặc** (dense graph), tức là đồ thị có nhiều cạnh, khi $E \approx V^2$

b) Trường hợp dùng hàng đợi ưu tiên với heap (heap-based implementation)

Với các ngôn ngữ lập trình hiện đại, ta có thể sử dụng **min-heap** hoặc **priority queue** (hàng đợi ưu tiên) để truy xuất nhanh đỉnh có khoảng cách nhỏ nhất, giúp tối ưu hóa quá trình chọn đỉnh.

- Chọn đỉnh gần nhất mất $O(\log V)$ với mỗi đỉnh → $O(V \log V)$.
- Mỗi cạnh có thể được xét một lần và cập nhật trong hàng đợi ưu tiên → $O(E \log V)$.

⇒ Tổng độ phức tạp thời gian:

$$O\big((V+E)\log V\big)$$

Cách hiện thực này đặc biệt hiệu quả với đồ thị **thưa (sparse graph)**, tức là đồ thị có số cạnh $E \ll V^2$, chẳng hạn như mạng giao thông, mạng máy tính thực tế.

c) So sánh với thuật toán Bellman-Ford

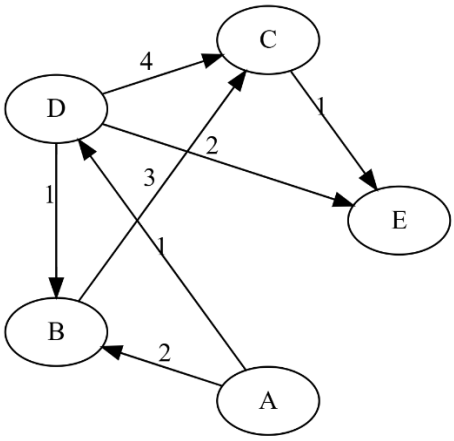
Thuật toán **Bellman-Ford** cũng là một thuật toán tìm đường đi ngắn nhất từ một đỉnh nguồn đến tất cả các đỉnh khác, với ưu điểm là **xử lý được đồ thị có cạnh trọng số âm**. Tuy nhiên, nó có chi phí thời gian cao hơn và không hiệu quả bằng Dijkstra trong hầu hết các trường hợp không có trọng số âm.

Thuật toán	Trọng số âm	Độ phức tạp thời gian	Độ phức tạp không gian
Dijkstra (mảng)	Không	$O(V^2)$	$O(V)$
Dijkstra (heap)	Không	$O\big((V+E)\log V\big)$	$O(V+E)$

Thuật toán	Trọng số âm	Độ phức tạp thời gian	Độ phức tạp không gian
Bellman-Ford	Có	$O(V \cdot E)$	$O(V)$

Do đó:

- Nếu đồ thị không có cạnh âm và cần tốc độ cao: ưu tiên **Dijkstra với heap**.
- Nếu đồ thị có trọng số âm: sử dụng **Bellman-Ford**.
- Nếu cần tìm đường đi ngắn nhất giữa mọi cặp đỉnh: chuyển sang **thuật toán Floyd-Warshall** (sẽ học ở buổi tiếp theo).



PHẦN 2: BÀI TẬP CÓ LỜI GIẢI

🔷 Bài 1. Truy vết thuật toán Dijkstra với đồ thị nhỏ

Cho đồ thị có 5 đỉnh: A, B, C, D, E với trọng số các cạnh như sau:

- A → B (2), A → D (1)
- B → C (3)
- C → E (1)
- D → B (1), D → C (4), D → E (2)

Yêu cầu:

- Trình bày từng bước thực hiện thuật toán Dijkstra để tìm đường đi ngắn nhất từ đỉnh **A** đến tất cả các đỉnh còn lại.
- Ghi lại bảng distance[] và previous[] sau mỗi lần cập nhật.

📌 **Mục tiêu:** Luyện cách truy vết từng bước thuật toán và đọc ra đường đi ngắn nhất.

✅ **Lời giải chi tiết**

Bước 0: Khởi tạo

- Tập đỉnh đã xét: S = ∅
- distance[A] = 0, các đỉnh còn lại là ∞
- previous của các đỉnh là null

Đỉnh	Distance	Previous
A	0	-
B	∞	-

Đỉnh	Distance	Previous
C	∞	-
D	∞	-
E	∞	-

Bước 1: Chọn A (distance nhỏ nhất = 0)

- Cập nhật B: distance[B] = 0 + 2 = 2, previous[B] = A
- Cập nhật D: distance[D] = 0 + 1 = 1, previous[D] = A

Đỉnh	Distance	Previous
A	0	-
B	2	A
C	∞	-
D	1	A
E	∞	-

S = {A}

Bước 2: Chọn D (distance = 1)

- D → B (1): 1 + 1 = 2 = current → giữ nguyên
- D → C (4): 1 + 4 = 5 → cập nhật
- D → E (2): 1 + 2 = 3 → cập nhật

Đỉnh	Distance	Previous
A	0	-
B	2	A
C	5	D
D	1	A
E	3	D

S = {A, D}

Bước 3: Chọn B (distance = 2)

- B → C (3): 2 + 3 = 5 = current → giữ nguyên

Không có cập nhật mới.

S = {A, D, B}

Bước 4: Chọn E (distance = 3)

Không có cạnh ra từ E. Không có cập nhật.

S = {A, D, B, E}

Bước 5: Chọn C (distance = 5)

- C → E (1): 5 + 1 = 6 > 3 → không cập nhật

S = {A, D, B, E, C}

✳️ **Kết quả cuối cùng**

Đỉnh	Distance	Previous
A	0	-
B	2	A
C	5	D
D	1	A
E	3	D

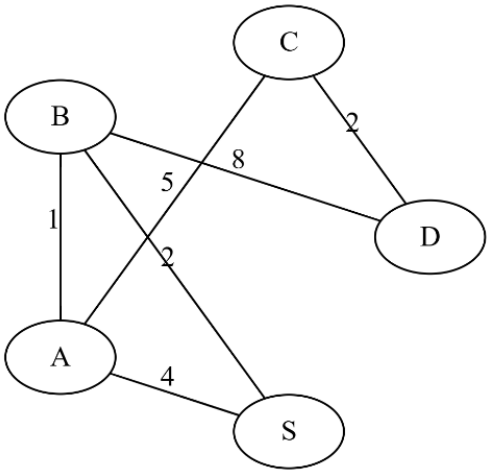
🔁 **Đọc đường đi ngắn nhất từ A**

- A → B: A → B
- A → C: A → D → C
- A → D: A → D
- A → E: A → D → E

Bài 2. Áp dụng Dijkstra cho đồ thị vô hướng

Cho đồ thị vô hướng gồm các đỉnh: {S, A, B, C, D} với các cạnh và trọng số:

- S – A (4)
- S – B (2)
- A – C (5)
- B – A (1)
- B – D (8)
- C – D (2)



Yêu cầu:

- Áp dụng thuật toán Dijkstra để tìm đường đi ngắn nhất từ đỉnh **S** đến **D**.
- Ghi lại đường đi và tổng trọng số.

📌 **Mục tiêu:** Phân biệt đồ thị vô hướng và biết suy luận đường đi ngắn nhất.

Lời giải chi tiết

Lưu ý ban đầu:

Vì đây là **đồ thị vô hướng**, mỗi cạnh **S – A (4)** được hiểu là **S → A và A → S** (trọng số như nhau).

Bước 0: Khởi tạo

- Tập đỉnh đã xét: S = ∅
- distance[S] = 0, các đỉnh còn lại là ∞
- previous[] ban đầu là null

Đỉnh	Distance	Previous
S	0	-
A	∞	-
B	∞	-
C	∞	-
D	∞	-

Bước 1: Chọn S (distance = 0)

- S → A (4): distance[A] = 4, previous[A] = S
- S → B (2): distance[B] = 2, previous[B] = S

Đỉnh	Distance	Previous
S	0	-
A	4	S
B	2	S
C	∞	-
D	∞	-

S = {S}

Bước 2: Chọn B (distance = 2)

- B → A (1): 2 + 1 = 3 < 4 → cập nhật distance[A] = 3, previous[A] = B
- B → D (8): 2 + 8 = 10, distance[D] = 10, previous[D] = B

Đỉnh	Distance	Previous
S	0	-
A	3	B
B	2	S
C	∞	-
D	10	B

S = {S, B}

Bước 3: Chọn A (distance = 3)

- A → C (5): 3 + 5 = 8, distance[C] = 8, previous[C] = A

Đỉnh	Distance	Previous
S	0	-
A	3	B
B	2	S
C	8	A
D	10	B

S = {S, B, A}

Bước 4: Chọn C (distance = 8)

- C → D (2): 8 + 2 = 10 = current[D] → giữ nguyên previous[D] = B

Đỉnh	Distance	Previous
S	0	-
A	3	B
B	2	S
C	8	A
D	10	B

S = {S, B, A, C}

Bước 5: Chọn D (distance = 10)

Không còn cạnh mới. Kết thúc thuật toán.

Kết quả

- Đường đi ngắn nhất từ **S** → **D** là:
  S → B → D
- Tổng trọng số:
  S → B (2) + B → D (8) = 10

Bài 3. Cài đặt thuật toán Dijkstra (mã giả hoặc Python)

Viết mã giả hoặc chương trình Python đơn giản để cài đặt thuật toán Dijkstra trên đồ thị có trọng số không âm (biểu diễn bằng **danh sách kề**).

Yêu cầu:

- Sử dụng **hàng đợi ưu tiên**
- Cho phép **người dùng nhập danh sách cạnh**
- Xuất ra **bảng khoảng cách** từ đỉnh nguồn đến tất cả các đỉnh

📌 **Mục tiêu học tập:**
Biết cách **chuyển hóa lý thuyết thành hiện thực**, rèn luyện kỹ năng lập trình thuật toán.

1. **Mã giả (Pseudocode) – Thuật toán Dijkstra**

```
Hàm Dijkstra(G, source):
    Khởi tạo khoảng cách distance[v] = ∞ cho mọi đỉnh v ∈ G
    distance[source] = 0
    previous[v] = null với mọi v
    Tạo hàng đợi ưu tiên Q chứa tất cả các đỉnh (theo distance)
    Trong khi Q không rỗng:
        u = đỉnh trong Q có distance nhỏ nhất
        Loại bỏ u khỏi Q
        Với mỗi đỉnh v kề với u:
            nếu distance[u] + trọng số(u, v) < distance[v]:
                distance[v] = distance[u] + trọng số(u, v)
                previous[v] = u
                Cập nhật v trong Q (theo distance mới)
    Trả về bảng distance và previous
```

2. **Cài đặt Python minh họa**

```python
import heapq
def dijkstra(n, edges, start):
    from collections import defaultdict
    # Tạo danh sách kề
    graph = defaultdict(list)
    for u, v, w in edges:
        graph[u].append((v, w))

    # Khởi tạo khoảng cách và hàng đợi ưu tiên
    distance = {v: float('inf') for v in range(n)}
    distance[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        dist_u, u = heapq.heappop(priority_queue)
        if dist_u > distance[u]:
            continue

        for v, weight in graph[u]:
            if distance[u] + weight < distance[v]:
                distance[v] = distance[u] + weight
                heapq.heappush(priority_queue, (distance[v], v))

    return distance
```

✅ **Ví dụ sử dụng**

```python
if __name__ == "__main__":
    print("Nhập số đỉnh:")
    n = int(input())  # Số lượng đỉnh (giả sử đánh số từ 0 đến n-1)
```

```
print("Nhập số cạnh:")
m = int(input())

print("Nhập các cạnh dưới dạng: u v w (u → v, trọng số w):")
edges = []
for _ in range(m):
    u, v, w = map(int, input().split())
    edges.append((u, v, w))

print("Nhập đỉnh xuất phát:")
source = int(input())

dist = dijkstra(n, edges, source)

print("\nKhoảng cách từ đỉnh", source)
for v in range(n):
    print(f"{source} → {v}: {dist[v]}")
```

📌 **Gợi ý bài kiểm thử**

Input mẫu:

Số đỉnh: 5
Số cạnh: 6
Danh sách cạnh:
0 1 2
0 3 1
1 2 3
3 1 1
3 2 4
3 4 2
Đỉnh bắt đầu: 0

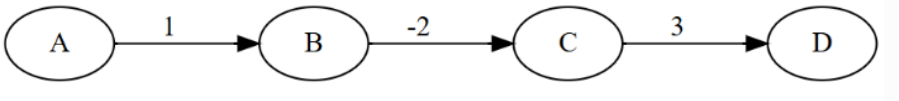
Kết quả mong đợi:

0 → 0: 0
0 → 1: 2
0 → 2: 5
0 → 3: 1
0 → 4: 3

Bài 4. Phản biện thuật toán khi có cạnh âm

Cho đồ thị có 4 đỉnh: {A, B, C, D} với các cạnh và trọng số như sau:

- A → B (1)
- B → C (-2)
- C → D (3)



Yêu cầu:

1. Giải thích tại sao **thuật toán Dijkstra không áp dụng được** cho đồ thị có cạnh âm.
2. Nếu **vẫn cố áp dụng Dijkstra từ đỉnh A**, kết quả sẽ **sai lệch như thế nào**?

📌 **Mục tiêu:** Nhận diện giới hạn áp dụng của Dijkstra và phát triển khả năng tư duy phản biện về tính đúng đắn của thuật toán.

1. Vì sao Dijkstra không áp dụng được khi có cạnh âm?

⛔ **Nguyên lý sai lệch:**

Thuật toán Dijkstra **dựa trên giả định** rằng:

> Khi ta chọn một đỉnh có khoảng cách nhỏ nhất và đánh dấu là "đã xét", thì khoảng cách đó là **tối ưu (final)** và không còn bị cải thiện sau này.

Điều này chỉ đúng **khi tất cả trọng số cạnh đều không âm**.

➤ **Khi tồn tại cạnh âm (ví dụ: B → C có trọng số -2):**

- Khoảng cách đến C có thể được **giảm sau** khi đi qua B → C, nhưng Dijkstra **không quay lại cập nhật** do đỉnh đã được "finalized".
- Do đó, thuật toán **bỏ qua cơ hội cải thiện** đường đi tốt hơn → kết quả **sai**.

2. Thử áp dụng Dijkstra từ đỉnh A

Các bước giả định thuật toán Dijkstra:

Khởi tạo:

distance[A] = 0
distance[B, C, D] = ∞

Bước 1: Chọn A → cập nhật B = 0 + 1 = 1

distance = { A: 0, B: 1, C: ∞, D: ∞ }

Bước 2: Chọn B (min = 1)

- B → C = 1 + (-2) = **-1** → cập nhật C

distance = { A: 0, B: 1, C: -1, D: ∞ }

Bước 3: Chọn C → cập nhật D = -1 + 3 = 2

distance = { A: 0, B: 1, C: -1, D: 2 }

Kết thúc thuật toán.

➞ **Kết quả thoạt nhìn có vẻ đúng**, nhưng đây là **may mắn**, do đồ thị nhỏ và đơn giản. Với đồ thị có chu trình âm hoặc nhiều đỉnh hơn, **Dijkstra sẽ cho sai**.

3. Nếu có chu trình âm thì sao?

Giả sử thêm cạnh C → A với trọng số **-5**, tạo thành chu trình âm:

A → B (1) → C (-2) → A (-5)

Khi áp dụng Dijkstra:

- Sau khi xét A, B, C xong, khoảng cách đến A là 0.
- Nhưng nếu quay lại qua chu trình âm: A → B → C → A thì tổng = 1 - 2 - 5 = **-6** → tốt hơn rất nhiều.

➡ Dijkstra sẽ **không phát hiện** đường đi tốt hơn này vì **A đã bị "kết thúc" trước đó**.

✅ Kết luận:

Câu hỏi	Trả lời
Có dùng Dijkstra khi có cạnh âm?	**Không.** Thuật toán chỉ đúng với đồ thị có trọng số **không âm**.
Lý do chính?	Vì Dijkstra **không quay lại** cập nhật khi tìm thấy đường tốt hơn.
Giải pháp thay thế?	**Bellman-Ford** là thuật toán đúng với trọng số âm, phát hiện được chu trình âm.

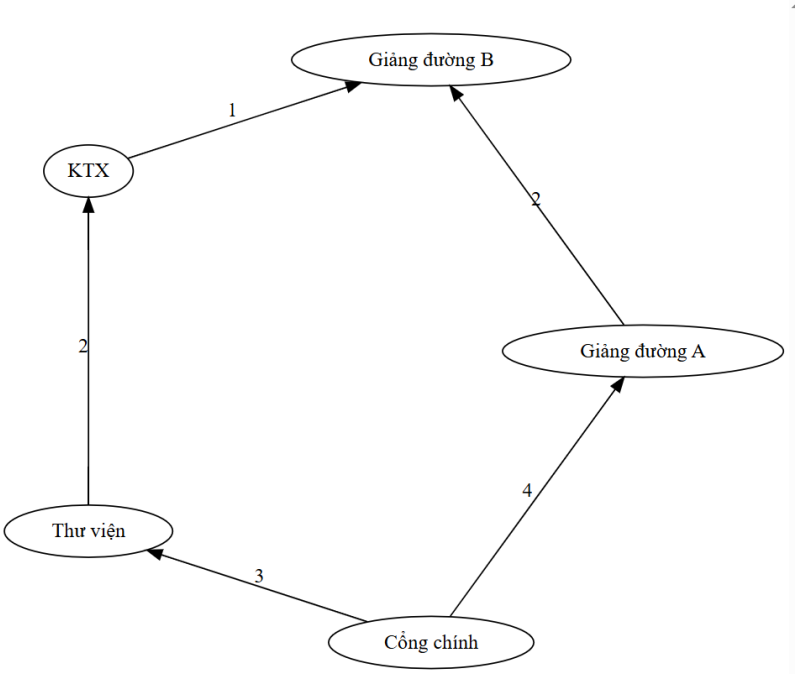
Bài 5. Ứng dụng thực tế – Định tuyến trong bản đồ

Ngữ cảnh thực tế: Một sinh viên mới vào trường muốn tìm đường **ngắn nhất từ Cổng chính đến Giảng đường B** trong khuôn viên đại học. Khuôn viên có các địa điểm:

- Cổng chính
- Thư viện
- Ký túc xá (KTX)
- Giảng đường A
- Giảng đường B

Các tuyến đường giữa các địa điểm:

- Cổng chính → Thư viện (3)
- Thư viện → KTX (2)
- Cổng chính → Giảng đường A (4)
- KTX → Giảng đường B (1)
- Giảng đường A → Giảng đường B (2)



1. Mô hình hóa thành đồ thị có trọng số

Gán tên định danh cho các đỉnh:

- 0: Cổng chính
- 1: Thư viện
- 2: KTX
- 3: Giảng đường A
- 4: Giảng đường B

Danh sách cạnh có hướng với trọng số:

```
edges = [
    (0, 1, 3),  # Cổng chính → Thư viện
    (1, 2, 2),  # Thư viện → KTX
    (0, 3, 4),  # Cổng chính → Giảng đường A
    (2, 4, 1),  # KTX → Giảng đường B
    (3, 4, 2),  # Giảng đường A → Giảng đường B
]
```

2. Áp dụng thuật toán Dijkstra từ "Cổng chính" đến "Giảng đường B"

Sử dụng lại hàm dijkstra() đã xây dựng ở Bài 3:

Chạy thuật toán Dijkstra từ đỉnh 0 (Cổng chính):

```
dist = dijkstra(5, edges, 0)
```

Bảng kết quả khoảng cách:

Vị trí	Khoảng cách từ Cổng chính	Đường đi ngắn nhất
Cổng chính	0	-
Thư viện	3	0 → 1
KTX	5	0 → 1 → 2
Giảng đường A	4	0 → 3
Giảng đường B	**6**	**0 → 1 → 2 → 4**

✅ **Kết luận:** Đường đi ngắn nhất từ **Cổng chính → Giảng đường B** là:
Cổng chính → Thư viện → KTX → Giảng đường B với **tổng trọng số là 6**

✅ **Ghi chú sư phạm**

Khía cạnh	Mục tiêu đạt được
Mô hình hóa	Chuyển tình huống thực tế thành đồ thị có trọng số
Giải thuật	Vận dụng Dijkstra để giải bài toán đường đi ngắn nhất
Phân tích kết quả	So sánh nhiều tuyến → chọn tuyến có tổng trọng số nhỏ nhất
Tư duy thực hành	Áp dụng kỹ năng lập trình và biểu diễn để giải quyết bài toán

PHẦN 3: TRẮC NGHIỆM

📚 Phần 1: Kiến thức cơ bản về đồ thị và bài toán tối ưu (Câu 1–5)

Câu 1. Đồ thị có trọng số là gì?

A. Đồ thị mà mỗi đỉnh có trọng số
B. Đồ thị mà mỗi cạnh có trọng số
C. Đồ thị có chu trình
D. Đồ thị vô hướng

Câu 2. Mô hình thực tế nào sau đây không phù hợp với bài toán tìm đường đi ngắn nhất?

A. Dẫn nước từ hồ đến các khu dân cư
B. Tìm đường đi từ nhà đến trường trên bản đồ
C. Tính chu kỳ học của một môn học
D. Giao hàng nhanh từ kho đến khách hàng

Câu 3. Trong ma trận trọng số, giá trị ∞ thường biểu diễn điều gì?

A. Trọng số lớn nhất
B. Không có đường đi giữa hai đỉnh
C. Có chu trình âm
D. Trọng số bằng 0

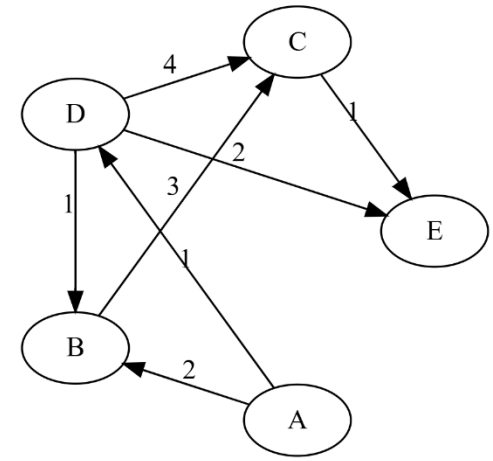
Câu 4. Đâu là yêu cầu bắt buộc để áp dụng thuật toán Dijkstra?

A. Đồ thị phải có chu trình
B. Tất cả trọng số phải dương hoặc bằng 0
C. Phải là đồ thị vô hướng
D. Đồ thị có ít hơn 5 đỉnh

Câu 5. Độ phức tạp thời gian của Dijkstra khi dùng hàng đợi ưu tiên (heap) là:

A. $O(V^2)$
B. $O(E \log V)$

C. O((V + E) log V)
D. O(VE)



📚 Phần 2: Truy vết thuật toán Dijkstra (Câu 6–10)

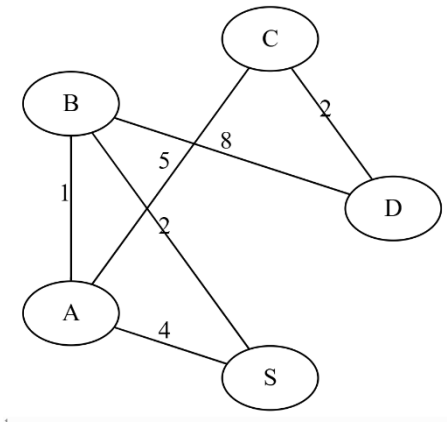
Câu 6. Với đồ thị sau, đường đi ngắn nhất từ A đến E là bao nhiêu?

A. 3
B. 4
C. 5
D. 6

Câu 7. Trong bảng distance sau bước đầu tiên (từ A), giá trị distance của D là:

A. 1
B. 2
C. 0
D. ∞

Câu 8. Với đồ thị vô hướng sau, khoảng cách ngắn nhất từ S đến D là?



A. 8
B. 9
C. 10
D. 11

Câu 9. Thuật toán Dijkstra có đảm bảo tối ưu trên mọi đồ thị không?

A. Có, nếu không có chu trình
B. Không, nếu có cạnh âm
C. Có, với mọi loại đồ thị
D. Có, nếu số đỉnh ≤ 5

Câu 10. Trong thuật toán Dijkstra, khi nào khoảng cách của một đỉnh được coi là tối ưu?

A. Khi vừa cập nhật xong
B. Khi lần đầu tiên được đưa vào hàng đợi
C. Khi được lấy ra khỏi hàng đợi và xử lý
D. Khi không còn đỉnh nào kề nó

📚 Phần 3: Lập trình và mô phỏng Dijkstra (Câu 11–15)

Câu 11. Cấu trúc dữ liệu nào bắt buộc phải dùng để đảm bảo hiệu năng Dijkstra?

A. Stack
B. Queue
C. Priority Queue (heap)
D. Dictionary

Câu 12. Trong đoạn mã Python, mục đích của dòng if dist_u > distance[u]: continue là gì?

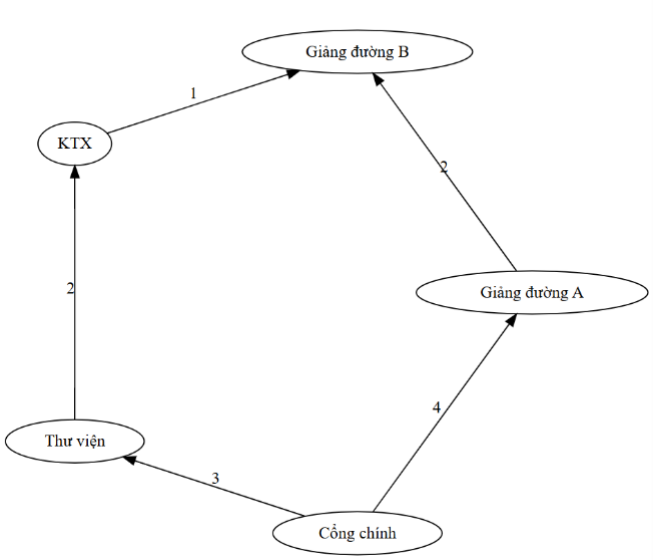
A. Ngăn tràn bộ nhớ
B. Loại bỏ các bản sao không còn tối ưu trong heap
C. So sánh các đỉnh cuối cùng
D. Tăng hiệu suất xử lý chu trình

Câu 13. Trong đoạn mã sau, lỗi logic nếu có sẽ xảy ra khi nào?

```
for v, w in graph[u]:
    if distance[u] + w < distance[v]:
        distance[v] = distance[u] + w
        heapq.heappush(queue, (distance[v], v))
```

A. Nếu w < 0
B. Nếu graph[u] trống
C. Nếu distance[u] = ∞
D. Không bao giờ

Câu 14. Với mô hình bản đồ sau, đường đi ngắn nhất từ "Cổng chính" đến "Giảng đường B" là?



A. Cổng chính → Giảng đường A → Giảng đường B
B. Cổng chính → Thư viện → KTX → Giảng đường B
C. Cổng chính → Thư viện → Giảng đường B
D. Cổng chính → Giảng đường B

Câu 15. Giải pháp đúng để xử lý đồ thị có cạnh âm là:

A. Dùng Dijkstra và bỏ qua cạnh âm
B. Dùng Bellman-Ford
C. Dùng BFS
D. Dùng DFS và đánh dấu cạnh âm

📚 Phần 4: Phản biện và tư duy mở rộng (Câu 16–20)

Câu 16. Nếu áp dụng Dijkstra cho đồ thị có cạnh âm, hậu quả có thể là?

A. Chương trình lỗi
B. Bỏ qua chu trình
C. Kết quả sai do không cập nhật lại khoảng cách
D. Không có đường đi

Câu 17. Với đồ thị có chu trình âm, thuật toán nào sẽ phát hiện được?

A. Dijkstra
B. Bellman-Ford
C. Kruskal
D. DFS

Câu 18. Đâu là đặc điểm nổi bật nhất của thuật toán Dijkstra?

A. Quy hoạch động
B. Tham lam (greedy)

C. Quay lui
D. Brute force

Câu 19. Trong trường hợp có nhiều đường đi có cùng độ dài, Dijkstra sẽ:

A. Tìm tất cả
B. Tìm ngẫu nhiên một đường
C. Tìm một đường đầu tiên được cập nhật
D. Dừng lại

Câu 20. Ứng dụng của Dijkstra phù hợp nhất với bài toán nào?

A. Tối ưu hóa ngân sách
B. Xác định thứ tự học phần
C. Định tuyến GPS
D. Nhận diện ảnh

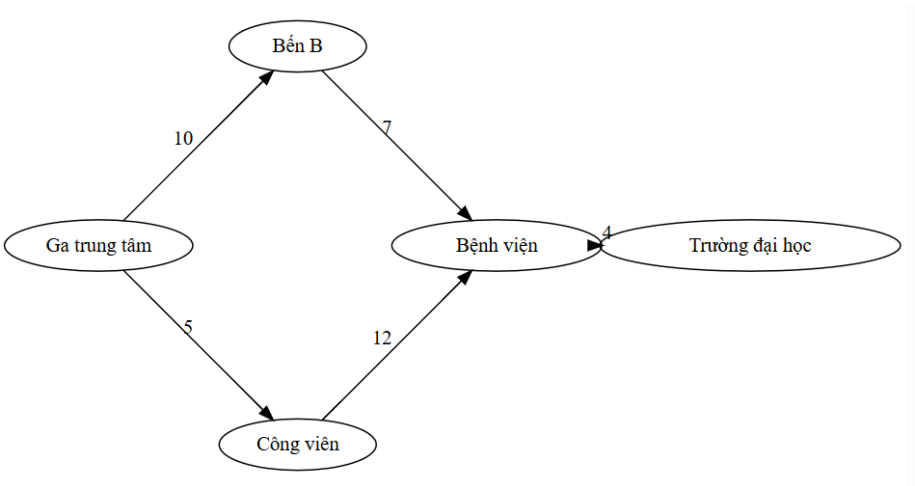
PHẦN 4: BÀI TẬP LUYỆN TẬP

🧩 Phần A – Nhận diện và mô hình hóa đồ thị

Bài 1. Mô hình hóa hệ thống xe buýt

Trong một thành phố, các trạm xe buýt gồm: {Ga trung tâm, Bến B, Công viên, Bệnh viện, Trường đại học}. Các tuyến đường có thời gian đi như sau (phút):

- Ga trung tâm → Bến B (10)
- Ga trung tâm → Công viên (5)
- Bến B → Bệnh viện (7)
- Công viên → Bệnh viện (12)
- Bệnh viện → Trường đại học (4)



Yêu cầu:

- Mô hình hóa hệ thống thành đồ thị có trọng số.
- Viết mã Python để lưu danh sách kề biểu diễn đồ thị.

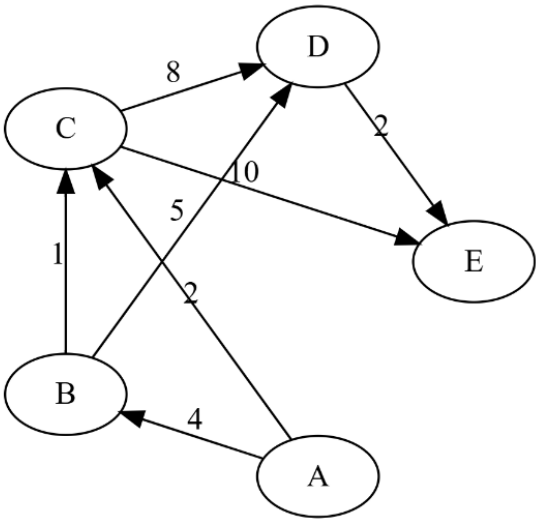
Hướng dẫn:

- Dùng số nguyên để đánh chỉ số đỉnh: 0 đến 4.
- Dùng defaultdict(list) để lưu danh sách kề.

🧩 Phần B – Truy vết thuật toán Dijkstra

Bài 2. Truy vết từ đỉnh nguồn

Cho đồ thị sau:



Yêu cầu:

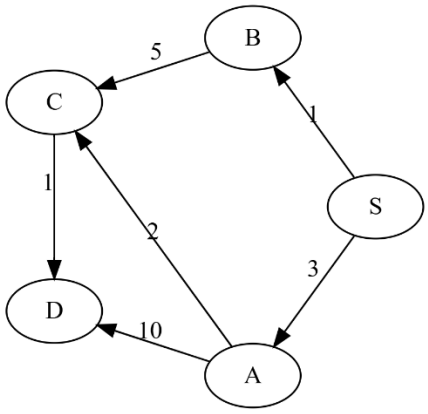
- Viết mã Python cài đặt thuật toán Dijkstra.
- In ra bảng khoảng cách từ đỉnh A đến tất cả các đỉnh.
- Xuất kết quả distance[] và previous[] để người dùng có thể đọc lại đường đi.

Hướng dẫn:

5. Dùng priority queue (heapq)
6. Lưu previous[] để truy vết đường đi cuối cùng.

Bài 3. So sánh hai đường đi

Cho đồ thị sau:



Yêu cầu:

- Áp dụng Dijkstra từ đỉnh S.

- Viết chương trình in đường đi và tổng độ dài đến đỉnh D.
- So sánh hai đường: S → A → D và S → B → C → D.

Hướng dẫn:

- In các đường đi ra màn hình.
- Dùng cấu trúc path[] kết hợp previous[].

🧩 Phần C – Cài đặt và đánh giá thuật toán

Bài 4. Viết hàm Dijkstra hỗ trợ vô hướng

Viết chương trình Python cho phép người dùng nhập đồ thị vô hướng bằng danh sách cạnh.

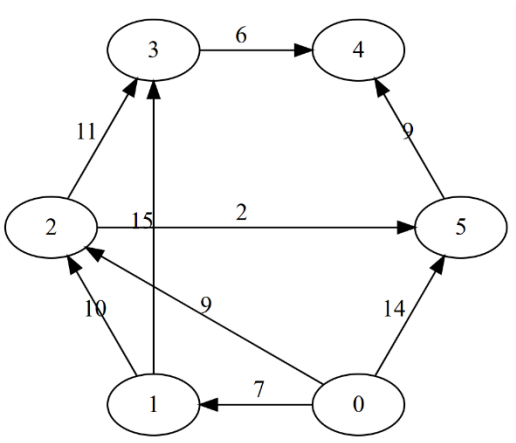
- Dữ liệu đầu vào: (u, v, w)
- Viết hàm tự động thêm cả hai chiều cạnh.
- Tìm đường ngắn nhất giữa hai đỉnh bất kỳ.

Hướng dẫn:

- Khi nhập cạnh (u, v, w) hãy thêm cả (v, u, w) vào danh sách kề.
- Viết hàm find_shortest_path(graph, start, end).

Bài 5. Tìm tất cả đỉnh xa nhất

Cho đồ thị có 6 đỉnh, biểu diễn như sau:



Yêu cầu:

- Viết chương trình tìm khoảng cách xa nhất từ đỉnh 0 đến các đỉnh khác.
- In ra đỉnh xa nhất và khoảng cách tương ứng.

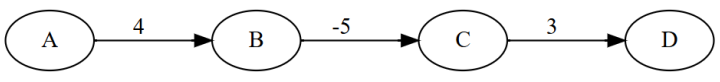
Hướng dẫn:

4. Áp dụng Dijkstra từ đỉnh 0.
5. Duyệt distance[] để tìm max.

🧩 Phần D – Tình huống phản biện và kiểm chứng thuật toán

Bài 6. Thử chạy Dijkstra trên đồ thị có cạnh âm

Cho đồ thị sau:



Yêu cầu:

- Viết code Dijkstra từ A và in kết quả.
- Nhận xét và phản biện kết quả.
- Thử chạy lại với trọng số không âm và so sánh.

Hướng dẫn:

- Ghi chú Dijkstra không hỗ trợ trọng số âm.
- Cho học sinh tự kiểm chứng độ chính xác kết quả.

Bài 7. Viết kiểm thử tự động cho Dijkstra

- Viết một hàm test_dijkstra() để kiểm thử Dijkstra với đồ thị nhỏ.
- Dữ liệu test là một đồ thị cụ thể, đầu vào đỉnh xuất phát, và khoảng cách mong đợi.
- So sánh kết quả thực tế với kỳ vọng.

Hướng dẫn:

- Dùng assert trong Python.
- Thiết lập đồ thị test như bài 2 hoặc 5.

🧩 Phần E – Ứng dụng và mô phỏng thực tế

Bài 8. Mô hình hóa mạng WiFi trong khu học xá
Một trường học có các điểm WiFi tại: Cổng chính, Thư viện, KTX, Phòng thực hành, Canteen.
Kết nối như sau:

- Cổng chính → Thư viện (2)
- Thư viện → KTX (3)
- KTX → Phòng thực hành (1)
- Thư viện → Canteen (5)
- Canteen → Phòng thực hành (4)

Yêu cầu:

- Viết chương trình xác định tuyến phát sóng tối ưu từ Cổng chính đến Phòng thực hành.

Hướng dẫn:

- Gợi ý học sinh mô phỏng đường đi tốt nhất với thuật toán Dijkstra.

Bài 9. Xây dựng công cụ tương tác

- Viết chương trình cho phép người dùng nhập danh sách cạnh từ bàn phím.
- Nhập số đỉnh, đỉnh xuất phát, và in bảng khoảng cách sau khi chạy Dijkstra.

Hướng dẫn:

- Dùng input() và split() để nhập dữ liệu.
- Hiển thị kết quả rõ ràng.

Bài 10. Mở rộng hiển thị đường đi

- Cập nhật chương trình Dijkstra để in không chỉ khoảng cách, mà còn cả **đường đi** chi tiết từ đỉnh xuất phát đến mỗi đỉnh.

Hướng dẫn:

- Dùng previous[] để truy ngược đường đi.
- Dùng đệ quy hoặc vòng lặp để in từ đích → nguồn.

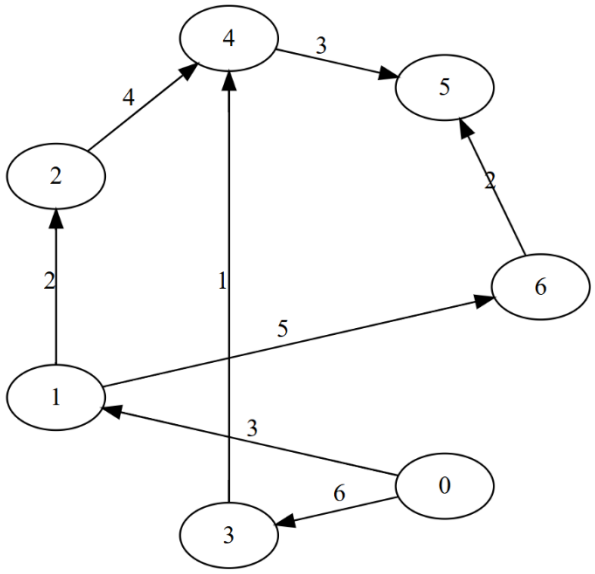
PHẦN 5: BÀI TẬP DỰ ÁN

🎯 Dự án 1: Ứng dụng định tuyến trong khuôn viên – Tìm đường đi tối ưu

Mục tiêu:
Xây dựng ứng dụng tìm đường đi ngắn nhất giữa hai vị trí bất kỳ trong khuôn viên trường đại học.
Mô tả bài toán:
Giả sử khuôn viên có các địa điểm:

- Cổng chính (0)
- Thư viện (1)
- Ký túc xá (2)
- Giảng đường A (3)
- Giảng đường B (4)
- Sân vận động (5)
- Phòng thực hành (6)

Các tuyến đường giữa các địa điểm:



Yêu cầu của dự án:

- Mô hình hóa đồ thị trong chương trình Python bằng danh sách kề.
- Viết hàm nhập điểm xuất phát và điểm đến từ người dùng.
- Áp dụng thuật toán Dijkstra để tìm đường đi ngắn nhất.
- In ra:
    - Tổng trọng số
    - Đường đi cụ thể (ví dụ: Cổng chính → Thư viện → Ký túc xá → Giảng đường B → Sân vận động)

Gợi ý triển khai:

- Viết lớp CampusMap, trong đó có:
    - add_edge(u, v, w)
    - shortest_path(start, end)
- Dùng heapq cho hàng đợi ưu tiên.
- Dùng dictionary previous[] để tái dựng đường đi.

🎯 Dự án 2: Công cụ định tuyến giao hàng trong thành phố nhỏ

Mục tiêu:

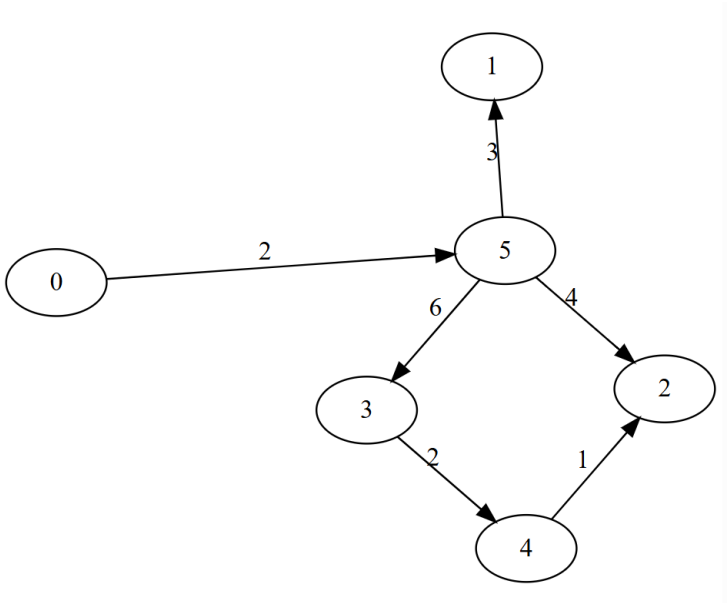
Tạo công cụ hỗ trợ doanh nghiệp giao hàng tìm tuyến đường ngắn nhất từ kho đến nhiều khách hàng trong một thành phố đơn giản (dạng cây hoặc đồ thị nhỏ không có chu trình âm).

Mô tả bài toán:

Thành phố có các vị trí:

- Kho hàng (0)
- Khách hàng A (1)
- Khách hàng B (2)
- Khách hàng C (3)
- Trung tâm phân phối (4)
- Giao lộ X (5)

Các đường phố được biểu diễn như sau:



Yêu cầu của dự án:

- Nhập danh sách cạnh từ người dùng hoặc đọc từ file.
- Hiển thị tất cả khoảng cách từ kho hàng (0) đến các khách hàng (1, 2, 3).
- Hiển thị tuyến đường đi cụ thể đến từng khách hàng.
- Cho phép người dùng chọn khách hàng cần giao và xuất đường đi.
- Mở rộng nâng cao:
    - Cho phép tính tổng quãng đường nếu đi đến tất cả khách hàng một lượt (gợi ý dùng Greedy + Dijkstra từng cặp).

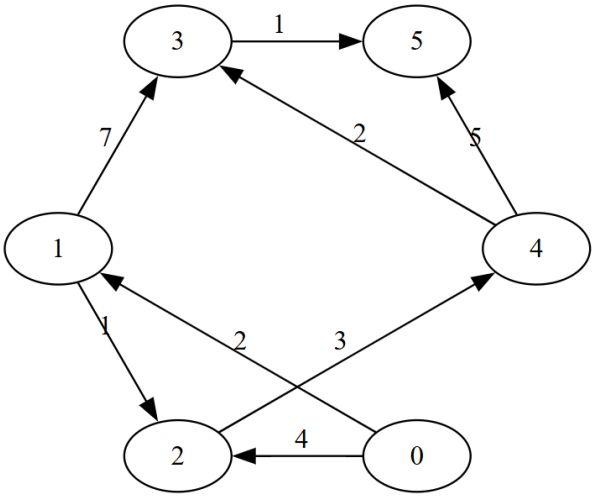
Hướng dẫn thực hiện:

- Viết chương trình tương tác dòng lệnh (main()).
- Tạo menu:
    1. Nhập dữ liệu đồ thị
    2. Chọn đỉnh nguồn và đỉnh đích
    3. Xem tất cả khoảng cách từ kho
    4. Tìm đường đi đến khách hàng cụ thể
- Dùng networkx + matplotlib để hiển thị nếu muốn đồ họa.

Gợi ý kỹ thuật nâng cao (tùy chọn):

- Tích hợp với Tkinter hoặc Streamlit để làm giao diện đơn giản.

MỘT SỐ GHI CHÚ THÊM

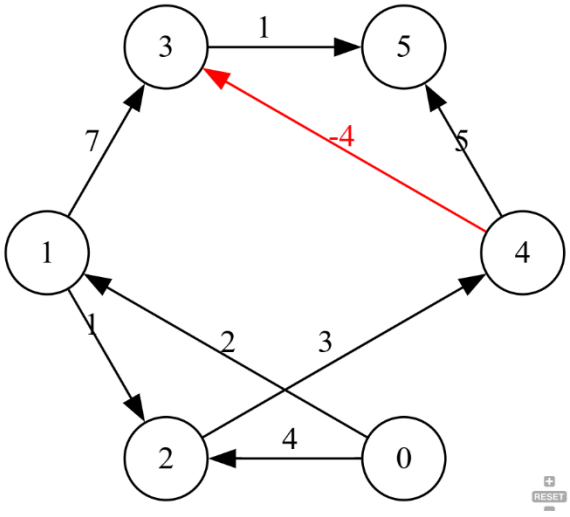


🧮 Bảng các bước Dijkstra

Step	Đỉnh được chọn	dist	previous	Các cập nhật trong bước
0	—	[0, ∞, ∞, ∞, ∞, ∞]	[None, None, None, None, None, None]	Khởi tạo
1	0	[0, 2, 4, ∞, ∞, ∞]	[None, 0, 0, None, None, None]	0→1 (2), 0→2 (4)
2	1	[0, 2, 3, 9, ∞, ∞]	[None, 0, 1, 1, None, None]	1→2 (1), 1→3 (7)
3	2	[0, 2, 3, 9, 6, ∞]	[None, 0, 1, 1, 2, None]	2→4 (3)
4	4	[0, 2, 3, 8, 6, 11]	[None, 0, 1, 4, 2, 4]	4→3 (2), 4→5 (5)
5	3	[0, 2, 3, 8, 6, 9]	[None, 0, 1, 4, 2, 3]	3→5 (1)
6	5	[0, 2, 3, 8, 6, 9]	[None, 0, 1, 4, 2, 3]	Không có cập nhật

✅ Kết quả cuối cùng:

Đỉnh	Khoảng cách từ 0	Đường đi ngắn nhất
0	0	0
1	2	0 → 1
2	3	0 → 1 → 2
3	8	0 → 1 → 2 → 4 → 3
4	6	0 → 1 → 2 → 4
5	9	0 → 1 → 2 → 4 → 3 → 5



📌 **Đồ thị có cạnh âm:**

- Cạnh âm duy nhất:

4 → 3 trọng số = -4

🔍 **Phân tích theo từng bước:**

✅ **Bước 1 → 3:**

- Các cập nhật là hợp lệ, chưa liên quan đến cạnh âm.
- dist[2] = 3 được cập nhật đúng từ đường đi tốt nhất: 0 → 1 → 2.

❗ **Bước 4:**

- Đỉnh 4 được xử lý:
  - dist[4] = 6 (qua 0 → 1 → 2 → 4)
  - Tại đây **xảy ra cạnh âm**: 4 → 3 = -4
    - dist[3] được cập nhật từ 9 → 6 + (−4) = **2** ✅
    - dist[5] = 6 + 5 = 11 ✅

Dù có cạnh âm, Dijkstra vẫn *cập nhật được giá trị đúng lúc này* vì đỉnh 4 chưa bị "khóa".

✅ **Bước 5:**

- Đỉnh 3 được chọn (vì dist[3] = 2)
  - dist[5] được cập nhật: 2 + 1 = 3
    ✅ Tốt hơn giá trị cũ là 11 → OK.

✅ **Bước 6:**

- Đỉnh 5 được chọn với dist[5] = 3
- Không có cạnh xuất phát từ 5, kết thúc thuật toán.

✅ **Kết quả cuối cùng (theo Dijkstra):**

Đỉnh	dist	Đường đi (theo previous)
0	0	—
1	2	0 → 1
2	3	0 → 1 → 2

Đỉnh	dist	Đường đi (theo previous)
3	2	0 → 1 → 2 → 4 → 3
4	6	0 → 1 → 2 → 4
5	3	0 → 1 → 2 → 4 → 3 → 5

⚠️ **Dijkstra vẫn đúng trong trường hợp này?**
Có vẻ như thuật toán đã cho kết quả đúng. Vậy có vấn đề không?

👉 **Câu trả lời: Có.**

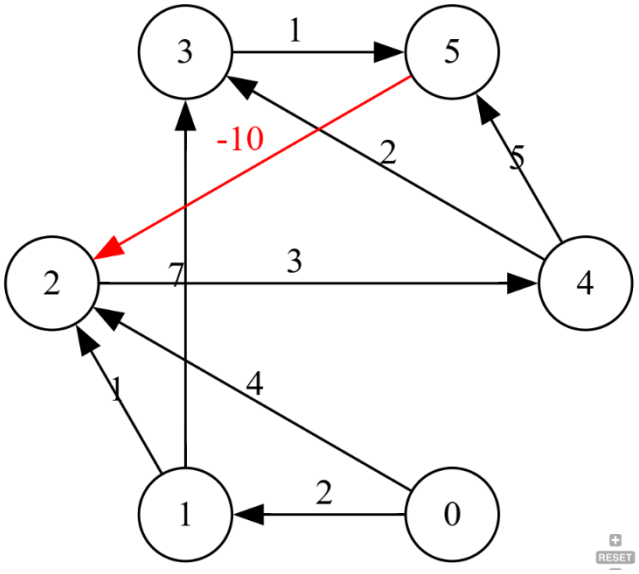
⚠️ **Dijkstra chỉ may mắn đúng ở đây vì:**

- Cạnh âm **không tạo thành chu trình âm**.
- Cạnh âm được xử lý **trước khi đỉnh đích (đỉnh 3)** bị khóa.
- Không có tình huống cần **quay lại** đỉnh đã "visited" để cập nhật dist.

❌ **Nhưng nếu đổi đồ thị hoặc cạnh âm nằm sau — Dijkstra sẽ thất bại.**
Ví dụ:

- Nếu 4 → 3 = -10 hoặc
- Nếu có chu trình âm: 3 → 5 → 4 → 3 chẳng hạn

→ **Dijkstra không cho phép quay lại để cập nhật đường đi tốt hơn**, và sẽ trả kết quả **sai**.



⚠️ **Tóm tắt đồ thị có chu trình âm**

Cạnh âm:
scss
CopyEdit
5 → 2 (−10)
Chu trình âm:
bash
CopyEdit
3 → 5 → 2 → 4 → 3 (Tổng trọng số: -4)

✅ **Phân tích theo các bước bạn đã mô tả:**

✔️ **Bước 1 → 4:**

Thuật toán cập nhật như bình thường. Cho đến đây, Dijkstra hoạt động ổn vì:

- Chưa chạm tới cạnh âm
- Các cập nhật dist và previous hợp lý theo nguyên tắc Dijkstra.

❗ **Bước 5: Xử lý đỉnh 3**

- dist[3] = 8, từ đỉnh 4 cập nhật trước đó.
- Cập nhật dist[5] = 8 + 1 = 9

➡️ Nhưng tại đây **Dijkstra đã "chốt" đỉnh 5 với khoảng cách 9, khóa lại không cho cập nhật thêm**.

❌ **Vấn đề xảy ra ở cạnh âm: 5 → 2 (−10)**

Sau khi đỉnh 5 được chọn và xử lý xong, **Dijkstra sẽ không kiểm tra lại đỉnh 2**, mặc dù:

- Đường đi mới:

CopyEdit

0 → 1 → 2 → 4 → 3 → 5 → 2

tổng trọng số:

0→1(2) + 1→2(1) + 2→4(3) + 4→3(2) + 3→5(1) + 5→2(-10) = -1

- Trong khi Dijkstra **đã chốt** dist[2] = 3, **nó không phát hiện ra rằng có đường đi tốt hơn với giá trị -1.**

  ➡️ **Sai nghiêm trọng**: Dijkstra cho rằng khoảng cách từ 0 → 2 là 3, trong khi nó có thể là -1 nếu cho phép cập nhật lại qua chu trình âm.