

Các thuật toán về tìm đường đi ngắn nhất

Các thuật toán về tìm đường đi ngắn nhất

Tác giả:

- Trần Hoài An - THPT Hoàng Lê Kha, Tây Ninh

Reviewer:


- Nguyễn Xuân Tùng - Đại học Quốc Tế, Đại học Quốc gia Thành phố Hồ Chí Minh

Giới thiệu

Bài toán tìm đường đi ngắn nhất trên đồ thị là một trong những bài toán đa dạng, có nhiều ứng dụng thực tế (như trong Google Maps, hay các bài toán networking, ...). Các dạng bài về tìm đường đi ngắn nhất cũng thường xuyên có mặt trong các kì thi lập trình.

Bài viết này sẽ giới thiệu ba thuật toán cơ bản của dạng bài tìm đường đi ngắn nhất:

- Thuật toán Bellman - Ford.
- Thuật toán Dijkstra.
- Thuật toán Floyd-Warshall (còn gọi là thuật toán Floyd).

Cần lưu ý rằng, [có một thuật toán thông dụng khác](#)  cũng có tên thường gọi là thuật toán Floyd, dùng để tìm chu trình trong đồ thị có hướng. Bài viết này sẽ chỉ đề cập đến thuật toán tìm đường đi ngắn nhất.

1. Thuật toán Bellman - Ford

Thuật toán Bellman-Ford dùng để giải quyết bài toán **đường đi ngắn nhất một nguồn** (Single-source shortest path), đồ thị **có thể có trọng số âm**.

Bài toán.

Cho đồ thị có hướng N đỉnh và M cạnh, và một đỉnh nguồn là đỉnh S . Mỗi cạnh có trọng số nguyên. **Trọng số này có thể âm hoặc dương hoặc bằng 0**. Với mỗi đỉnh u từ 1 đến N . Yêu cầu xuất kết quả tại mỗi đỉnh u như sau:

1. Nếu không tồn tại đường đi từ S đến u thì in ra: **Impossible**
2. Nếu tồn tại đường đi từ S đến u nhưng không có đường đi ngắn nhất in ra: **-Infinity**

3. Trường hợp còn lại in ra đường đi ngắn nhất từ S đến u .

Input: Dòng đầu tiên gồm 3 số nguyên N, M, S . M dòng tiếp theo, mỗi dòng gồm ba số nguyên $u, v, W_{u,v}$ biểu diễn một cạnh một chiều từ u đến v với trọng số là $W_{u,v}$.

Output: gồm N dòng cho biết đường đi ngắn nhất từ đỉnh S đến các đỉnh từ 0 đến $N - 1$

Giới hạn bài toán : $1 \leq N \leq 1000, 1 \leq M \leq 5000$

Sample Input

```

1 | 7 6 4
2 | 0 1 7
3 | 2 0 1
4 | 1 2 -9
5 | 2 5 1000
6 | 3 0 7
7 | 4 3 3

```

Sample Output

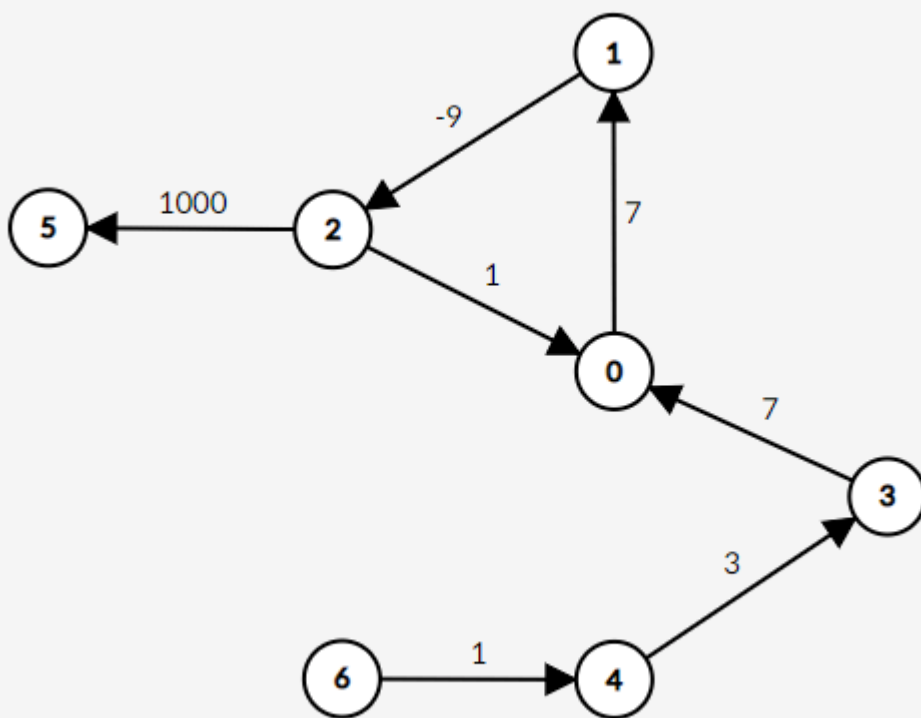
```

1 | -Infinity
2 | -Infinity
3 | -Infinity
4 | 3
5 | 0
6 | -Infinity
7 | Impossible

```

Khái niệm về chu trình âm

- Chu trình âm là một chu trình trong đó tổng trọng số các cạnh là số âm. Ví dụ trong hình dưới, ta có một chu trình âm $0 \rightarrow 1 \rightarrow 2$ có tổng trọng số là $7 - 9 + 1 = -1$



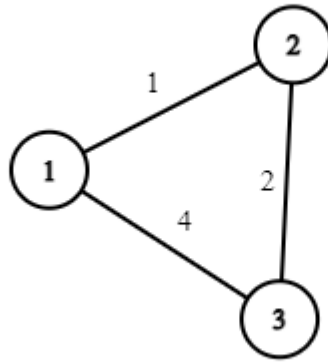
- Nếu trên đường đi từ u đến v chứa chu trình âm thì độ dài đường đi ngắn nhất từ u đến v sẽ là $-\infty$. Vì vậy nên sự xuất hiện của chu trình âm trong đồ thị sẽ khiến một số cặp đỉnh không tồn tại đường đi ngắn nhất (chỉ tồn tại đường đi có độ dài âm vô cực).
- Ví dụ: Ở đồ thị trên, đường đi ngắn nhất từ 4 đến 5 sẽ có cách đi là vô hạn lần qua chu trình âm đã nhắc đến, sau đó mới đi đến 5. Như vậy không có đường đi ngắn nhất.

Ý tưởng của thuật toán.

Xét trường hợp đơn giản hơn, khi đồ thị không có trọng số âm (tức đường đi ngắn nhất luôn tồn tại).

Thuật toán Bellman-Ford sẽ lặp nhiều lần. Ở mỗi vòng lặp, ta sẽ đi qua **tất cả** các cạnh (u, v) trên đồ thị, so sánh đường đi $S \rightarrow v$ đã tìm được với đường đi $S \rightarrow u \rightarrow v$

- Ví dụ đồ thị sau:



- Giả sử ta tìm được đường đi từ $1 \rightarrow 3$ có độ dài là 4, và đường đi từ $1 \rightarrow 2$ có độ dài là 2. Như vậy ta có thể sử dụng cạnh $(2, 3)$ để nối dài đường đi $1 \rightarrow 2$ thành $1 \rightarrow 2 \rightarrow 3$ có độ dài bằng 3, tốt hơn đường đi trực tiếp $1 \rightarrow 3$ ta đã tìm được.

Có thể chứng minh được rằng, vòng lặp trên cần thực hiện $N - 1$ lần, mỗi lần đi qua toàn bộ M cạnh, là sẽ đủ để tìm đường đi ngắn nhất.

- Chứng minh:** Nhận xét rằng một đường đi ngắn nhất bất kì sẽ không có đỉnh nào được đi lại quá một lần. Như vậy một đường đi ngắn nhất sẽ không có quá $N - 1$ cạnh. Việc thực hiện phép tính $D_v = D_u + W_{u,v}$ cũng đồng nghĩa với thêm một cạnh $u \rightarrow v$ vào hành trình đi từ s đến v . Vậy một D_u chỉ có thể được tối ưu tối đa $N - 1$ lần, và từ lần thứ N trở đi sẽ không thể tối ưu hơn được nữa.

Cài Đặt

Ở thuật toán này, đồ thị thường được lưu ở dạng **danh sách cạnh**.

- Định nghĩa $W[u, v]$ là trọng số cạnh nối từ đỉnh u đến đỉnh v (nếu có).
- Định nghĩa mảng $D[u]$ là đường đi ngắn nhất từ $s \rightarrow u$. Ban đầu, chưa có đường đi nào, $D[u] = \infty$ với mọi u , ngoại trừ $D[s] = 0$.
 - Nếu cần tìm lại chính đường đi ngắn nhất, ta có thể định nghĩa thêm một mảng truy vết $trace[u]$. Ban đầu mọi $trace[u]$ bằng -1 nghĩa là chưa có đường đi.
- Thực hiện $N - 1$ lần: Xét lần lượt các cạnh (u, v) trong đồ thị. Nếu $D[u] + W[u, v] < D[v]$, cập nhật $D[v] = D[u] + W[u, v]$, đồng thời cập nhật $trace[v] = u$.
- Độ phức tạp:** Ta có một vòng lặp được thực hiện $N - 1$ lần, mỗi lần lặp ta sẽ xử lí tất cả các cạnh trong đồ thị, như vậy độ phức tạp của thuật toán là $O(N * M)$.

Code:

```

const long long INF = 2000000000000000LL;
struct Edge {
    int u, v;
    long long w; // cạnh từ u đến v, trọng số w

```

```

5   };
6
7   void bellmanFord(int n, int S, vector<Edge> &e, vector<long long> &D, vector<int> &trace) {
8       // e: danh sách cạnh
9       // n: số đỉnh
10      // S: đỉnh bắt đầu
11      // D: độ dài đường đi ngắn nhất
12      // trace: mảng truy vết đường đi
13      // INF nếu không có đường đi
14      // -INF nếu có đường đi âm vô tận
15      D.resize(n, INF);
16      trace.resize(n, -1);
17
18      D[S] = 0;
19      for(int T = 1; T < n; T++) {
20          for (auto E : e) {
21              int u = E.u;
22              int v = E.v;
23              long long w = E.w;
24              if (D[u] != INF && D[v] > D[u] + w) {
25                  D[v] = D[u] + w;
26                  trace[v] = u;
27              }
28          }
29      }

```

Tìm lại đường đi ngắn nhất

Theo tác tìm đường đi ngắn nhất từ S đến u khá đơn giản, ta sẽ bắt đầu từ đỉnh u , sau đó truy vết theo mảng

```

1   vector<int> trace_path(vector<int> &trace, int S, int u) {
2       if (u != S && trace[u] == -1) return vector<int>(0); // không có đường đi
3
4       vector<int> path;
5       while (u != -1) { // truy vết ngược từ u về S
6           path.push_back(u);
7           u = trace[u];
8       }
9       reverse(path.begin(), path.end()); // cần reverse vì đường đi lúc này là từ u về S
10
11      return path;
12  }

```

Các trường hợp có chu trình âm

Thuật toán Bellman-Ford có thể xử lý được thêm trường hợp nhận biết chu trình âm, cũng như nhận biết nếu không tồn tại đường đi ngắn nhất đến một đỉnh.

Nhận biết đường đi âm vô cực

- ▶ Nhận xét tiếp rằng, ta có thể chạy vòng quanh chu trình âm liên tục để được đường đi ngắn hơn. Như vậy thuật toán Bellman-Ford ở vòng lặp thứ N trở đi vẫn sẽ liên tục tối ưu được đường đi, thay vì dừng lại ở lần thứ $N - 1$.
- ▶ Ta chỉ cần chạy thuật toán Bellman-Ford thêm một lần nữa với N vòng lặp, những đỉnh nào vẫn còn tối ưu được ở lần chạy thứ hai sẽ tối ưu được mãi mãi, và đó là các đỉnh không tồn tại đường đi ngắn nhất.

Code:

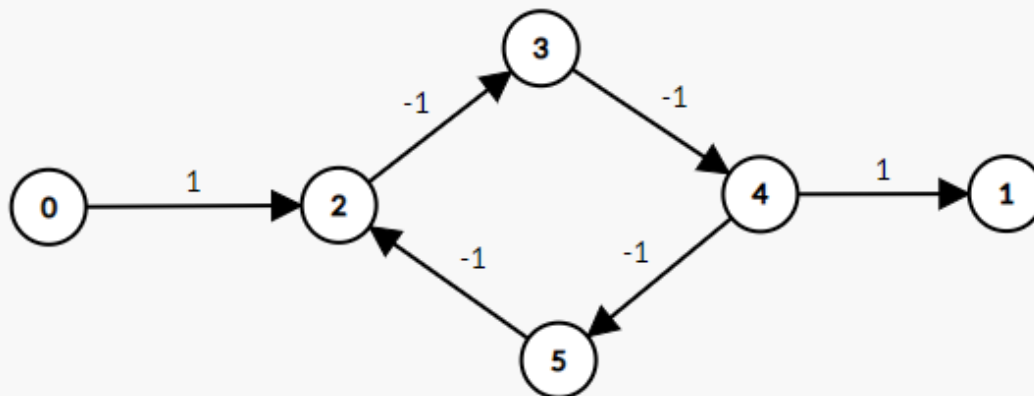
```

1 // sau khi chạy xong N-1 vòng lặp Bellman-Ford
2 for(int T = 0; T < n; T++){
3     for (auto E : e) {
4         int u = E.u;
5         int v = E.v;
6         long long w = E.w;
7         if (D[u] != INF && D[v] > D[u] + w) {
8             // vẫn còn tối ưu được --> âm vô cực
9             D[v] = -INF;
10            trace[v] = u;
11        }
12    }
13 }
```

Tìm chu trình âm

Một số bài toán có thể yêu cầu ta tìm một chu trình âm bất kì trong đồ thị. Ta có thể chỉnh sửa thuật toán Bellman-Ford lại như sau:

- ▶ Thay vì chạy N vòng lặp Bellman-Ford như trường hợp trên, ta chỉ cần chạy một vòng lặp. Như vậy là đủ để phát hiện ít nhất một đỉnh có đường đi bằng $-\infty$ (nếu có).
- ▶ Tiến hành truy vết: Bắt đầu từ đỉnh u bất kì có đường đi bằng $-\infty$, ta sẽ truy vết theo mảng *trace*:
 - ▶ Trước hết gán $u = \text{trace}[u]$ đủ N lần.
 - ▶ Mục đích của bước này là để u chắc chắn thuộc chu trình âm. Ban đầu có thể đỉnh u có đường đi bằng $-\infty$ nhưng chưa chắc thuộc chu trình âm. Ví dụ trường hợp sau:



Ở đây, từ 0 đến 1 có độ dài đường đi ngắn nhất bằng $-\infty$, tuy nhiên đỉnh 1 lại không thuộc chu trình âm nào.

- Sau đó, u sẽ thuộc một chu trình âm. Ta chỉ cần truy vết đỉnh u theo mảng *trace* cho đến khi gặp lại chính nó, sẽ được một chu trình.
- Chu trình vừa truy vết chính là một chu trình âm của đồ thị. Lưu ý ta vẫn phải đảo ngược kết quả truy vết, vì ta đang truy vết ngược so với đồ thị gốc.

```

bool findNegativeCycle(int n, vector<long long> &D, vector<int> &trace, vector<int> &E) {
    // mảng D và trace đã được chạy qua thuật toán Bellman-Ford
    int negStart = -1; // đỉnh bắt đầu
    for (auto E : e) {
        int u = E.u;
        int v = E.v;
        long long w = E.w;
        if (D[u] != INF && D[v] > D[u] + w) {
            D[v] = -INF;
            trace[v] = u;
            negStart = v; // đã tìm thấy -INF
        }
    }

    if (negStart == -1) return false; // không có chu trình âm

    int u = negStart;
    for (int i = 0; i < n; i++) {
        u = trace[u]; // đưa u về chu trình âm
    }

    negCycle = vector<int>(1, u);
    for (int v = trace[u]; v != u; v = trace[v]) {
        negCycle.push_back(v); // truy vết một vòng
    }
    reverse(negCycle.begin(), negCycle.end());
}

```

```

    return true;
}

```

2. Thuật toán Dijkstra

Thuật toán Dijkstra dùng để giải quyết bài toán **đường đi ngắn nhất một nguồn** (Single-source shortest path), đồ thị **trọng số không âm**.

Bài toán.

Cho một đồ thị có hướng với N đỉnh (được đánh số từ 0 đến $N - 1$), M cạnh có hướng, có trọng số, và một đỉnh nguồn S . **Trọng số của tất cả các cạnh đều không âm**. Yêu cầu tìm ra đường đi ngắn nhất từ đỉnh S tới tất cả các đỉnh còn lại (hoặc cho biết nếu không có đường đi).

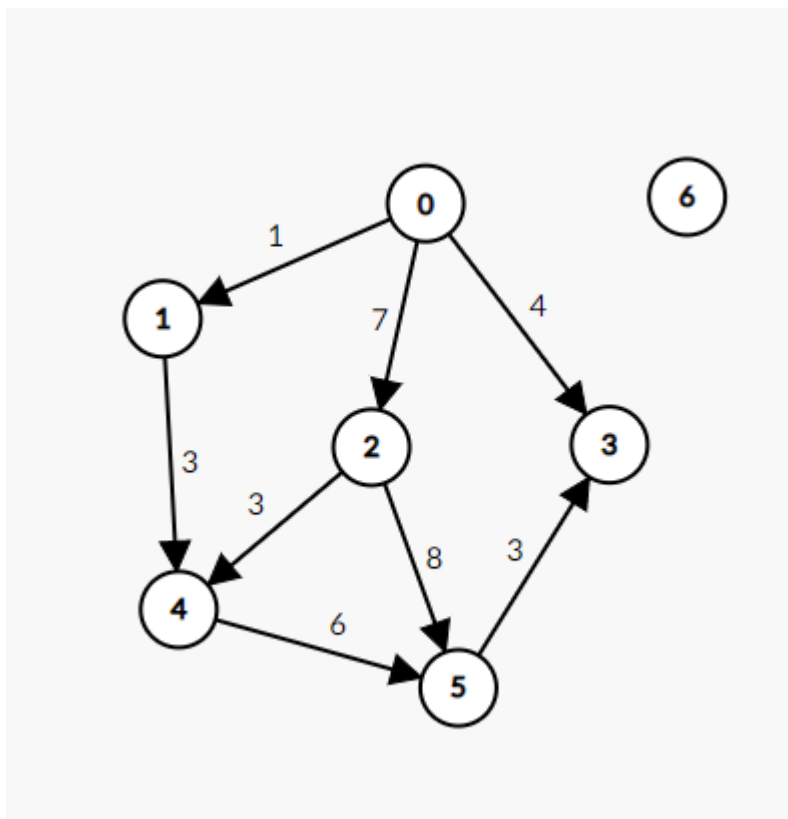
Sample Input

1	7	8	0
2	0	2	7
3	0	1	1
4	0	3	4
5	2	5	8
6	5	3	3
7	4	5	6
8	1	4	3
9	2	4	3

Sample Output

1	0
2	1
3	7
4	4
5	4
6	10
7	-1

Hình ảnh của Test ví dụ. Ở đồ thị này, đỉnh nguồn là đỉnh 0, đường đi ngắn nhất từ 0 đến các đỉnh 0 đến 5 là $[0, 1, 7, 4, 4, 10]$. Riêng đỉnh 6 không có đường đi đến.



Ý tưởng của thuật toán.

Giống như thuật toán Bellman-Ford, thuật toán Dijkstra cũng tối ưu hóa đường đi bằng cách xét các cạnh (u, v) , so sánh hai đường đi $S \rightarrow v$ sẵn có với đường đi $S \rightarrow u \rightarrow v$.

Thuật toán hoạt động bằng cách duy trì một tập hợp các đỉnh trong đó ta đã biết **chắc chắn** đường đi ngắn nhất. Mỗi bước, thuật toán sẽ chọn ra một đỉnh u mà chắc chắn sẽ không thể tối ưu hơn nữa, sau đó tiến hành tối ưu các đỉnh v khác dựa trên các cạnh (u, v) đi ra từ đỉnh u . Sau N bước, tất cả các đỉnh đều sẽ được chọn, và mọi đường đi tìm được sẽ là ngắn nhất.

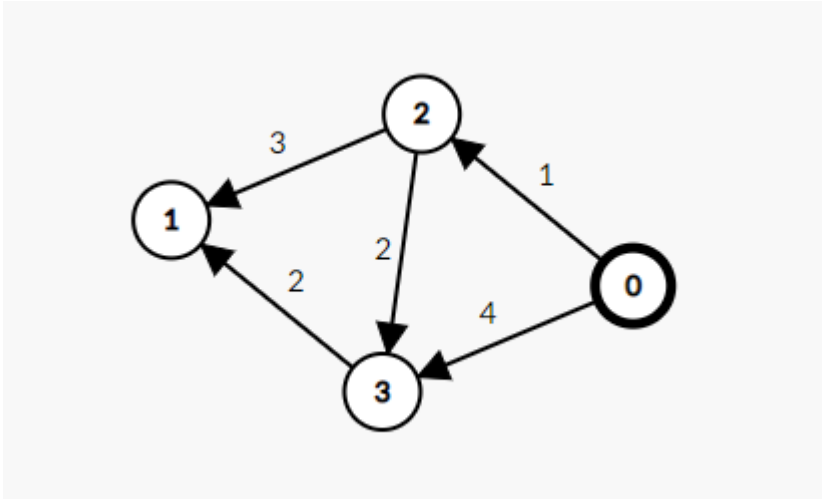
Cụ thể hơn, thuật toán sẽ duy trì đường đi ngắn nhất đến tất cả các đỉnh. Ở mỗi bước, chọn đường đi $S \rightarrow u$ có tổng trọng số nhỏ nhất trong tất cả các đường đi đang được duy trì. Sau đó tiến hành tối ưu các đường đi $S \rightarrow v$ bằng cách thử kéo dài thành $S \rightarrow u \rightarrow v$ như đã mô tả trên.

Minh họa thuật toán

Ta sẽ minh họa thuật toán bằng một đồ thị như hình. Định nghĩa:

- D_u là đường đi ngắn nhất từ đỉnh nguồn đến đỉnh u đã tìm được.
- P_u nhận hai giá trị *true*, *false* cho biết đỉnh P_u đã được chọn để tối ưu chưa.

Đỉnh được tô đen (đỉnh 0) sẽ là đỉnh nguồn.



Ban đầu, $D = [0, \infty, \infty, \infty]$, $P = [false, false, false, false]$

- Bước 1: Thuật toán sẽ chọn đỉnh 0, vì $D_0 = 0$ là nhỏ nhất thỏa mãn $P_0 = false$. Tiến hành tối ưu các cạnh đi ra:
 - Cạnh (0, 2): cập nhật $D_2 = \min(D_2, D_0 + W_{0,2}) = \min(\infty, 0 + 1) = 1$
 - Cạnh (0, 3): cập nhật $D_3 = \min(D_3, D_0 + W_{0,3}) = \min(\infty, 0 + 4) = 4$

Sau bước này, $D = [0, \infty, 1, 4]$, $P = [true, false, false, false]$

- Bước 2: thuật toán sẽ chọn ra đỉnh 2, có $D_2 = 1$ là nhỏ nhất thỏa mãn $P_2 = false$. Tiến hành tối ưu các cạnh đi ra:
 - Cạnh (2, 1): cập nhật $D_1 = \min(D_1, D_2 + W_{2,1}) = \min(\infty, 1 + 3) = 4$
 - Cạnh (2, 3): cập nhật $D_3 = \min(D_3, D_2 + W_{2,3}) = \min(4, 1 + 2) = 3$

Sau bước này, $D = [0, 4, 1, 3]$, $P = [true, false, true, false]$

- Bước 3: thuật toán sẽ chọn ra đỉnh 3, có $D_3 = 1$ là nhỏ nhất thỏa mãn $P_3 = false$. Tiến hành tối ưu các cạnh đi ra:
 - Cạnh (3, 1): cập nhật $D_1 = \min(D_1, D_3 + W_{3,1}) = \min(4, 3 + 2) = 4$

Sau bước này, $D = [0, 4, 1, 3]$, $P = [true, false, true, true]$

- Bước 4: thuật toán sẽ chọn đỉnh 1. Không có cạnh nào đi ra.

Đến đây, tất cả các đỉnh đều đã được đánh dấu. Thuật toán kết thúc. Đường đi ngắn nhất tìm được từ đỉnh 0 là $D = [0, 4, 1, 3]$.

Cài đặt

Ở thuật toán này, ta sẽ lưu đồ thị dưới dạng **danh sách kề**. Ta định nghĩa như sau:

- $D[u]$ là đường đi ngắn nhất từ $s \rightarrow u$. Ban đầu khởi tạo $D[u] = \infty$ với mọi u , riêng $D[s] = 0$.
 - Cũng như thuật toán Bellman-Ford, ta có thể định nghĩa thêm mảng *trace* để truy vết đường đi nếu cần.
- $W[u, v]$ là trọng số cạnh trên đường đi từ $u \rightarrow v$.
- $P[u]$ là mảng đánh dấu các đỉnh u đã được xử lý chưa. Ban đầu tất cả các giá trị đều là **false**.

Ta sẽ lặp N lần quá trình sau:

- Tìm đỉnh u có $D[u]$ nhỏ nhất và $P[u] = false$.
- Sau khi tìm được đỉnh u , ta xét các đỉnh v kề với đỉnh u và tiến hành tối ưu hóa $D[v]$: nếu $D[v] > D[u] + W[u, v]$ thì $D[v] = D[u] + W[u, v]$.
 - Nếu việc tối ưu hóa diễn ra, ta sẽ cập nhật $trace[v] = u$.
- Đánh dấu $P[u] = true$, nghĩa là đỉnh u đã được xử lý xong

Độ phức tạp thuật toán: Ta có N lần lặp:

- Bước đầu tiên có độ phức tạp $O(N)$ **mỗi lần lặp**.
- Bước thứ hai có **tổng độ phức tạp $O(M)$ qua tất cả các lần lặp**

Như vậy độ phức tạp của cách cài đặt cơ bản sẽ là $O(N^2 + M)$.

Code:

```
const long long INF = 2000000000000000000LL;
struct Edge{
    int v;
    long long w;
};
void dijkstra(int n, int S, vector<vector<Edge>> E, vector<long long> &D, vector<bool> &P, vector<int> &trace){
    D.resize(n, INF);
    trace.resize(n, -1);

    vector<bool> P(n, 0);
    D[S] = 0;

    for (int i = 0; i < n; i++) {
        int uBest; // tìm đỉnh u chưa dùng, có khoảng cách nhỏ nhất
        long long Max = INF;
        for (int u = 0; u < n; u++) {
            if(D[u] < Max && P[u] == false) {
                uBest = u;
                Max = D[u];
            }
        }

        // cải tiến các đường đi qua u
        int u = uBest;
        P[u] = true;
        for(auto x : E[u]) {
            int v = x.v;
            long long w = x.w;
            if(D[v] > D[u] + w) {
                D[v] = D[u] + w;
                trace[v] = u;
            }
        }
    }
}
```

```

33 |         }
34 |     }
35 | }

```

Cải tiến đối với đồ thị thưa

- ▶ Nhận xét rằng bước đầu tiên: "Tìm đỉnh u có D_u nhỏ nhất và $P_u = false$ ", có thể được cải tiến. Ta có thể sử dụng cấu trúc dữ liệu [Heap](#) \square (cụ thể là Min Heap) hoặc cây nhị phân tìm kiếm để cải tiến bước này.
- ▶ Mỗi lần tối ưu hóa D_v , ta đẩy cặp $\{D_v, v\}$ vào trong Heap.
- ▶ Để tìm đỉnh có D_u nhỏ nhất, ta chỉ cần liên tục lấy phần tử trên cùng trong Heap ra, cho đến khi gặp đỉnh u thỏa mãn $P_u = false$.
- ▶ Mỗi lần tối ưu D_v , ta phải push vào Heap một lần. Với mỗi lần push vào trong Heap, ta đều phải pop ra lại một lần. Do có tối đa $O(M)$ lần tối ưu, độ phức tạp của thuật toán là $O(M \log N)$.

Độ phức tạp sau khi cải tiến: $O(M \log N)$. **Lưu ý rằng với M lớn thì độ phức tạp này không tốt hơn cài đặt cơ bản.** Chứng minh như sau:

- ▶ Ta có độ phức tạp của hai cách cài đặt:
 - ▶ Cách cài đặt cơ bản: $O(N^2 + M)$.
 - ▶ Cách cài đặt cải tiến: $O(M \log N)$.
- ▶ Số lượng cạnh M trong đơn đồ thị không vượt quá N^2 , như vậy độ phức tạp của cách cơ bản có thể được viết đơn giản thành $O(N^2)$.
- ▶ Để cách cài đặt cải tiến tốt hơn, ta cần $M \log N < N^2$ suy ra $M < N^2 / \log N$.
 - ▶ Ví dụ: đối với $N = 10^5$, ta cần $M > 6 \cdot 10^8$ để cách cài đặt cơ bản tốt hơn cách cài đặt cải tiến. Thực tế trong lập trình thi đấu khó có đồ thị nào có số cạnh lớn như vậy. Vì thế nhìn chung khi N lớn thì thuật $O(M \log N)$ luôn tốt hơn.

Code:

```

const long long INF = 2000000000000000000LL;
struct Edge{// kiểu dữ liệu tự tạo để lưu thông số của một cạnh.
    int v;
    long long w;
};
struct Node{// kiểu dữ liệu để lưu đỉnh u và độ dài của đường đi ngắn nhất từ :
    int u;
    long long Dist_u;
};
struct cmp{
    bool operator() (Node a, Node b) {
        return a.Dist_u > b.Dist_u;
    }
};
void dijkstraSparse(int n, int s, vector<vector<Edge>> &E, vector<long long> &D, int INF);

```

```

17     trace.resize(n, -1);
18     vector<bool> P(n, 0);
19
20     D[s] = 0;
21     priority_queue<Node, vector<Node>, cmp> h; // hàng đợi ưu tiên, sắp xếp th
22     h.push({s, D[s]});
23
24     while(!h.empty()) {
25         Node x = h.top();
26         h.pop();
27
28         int u = x.u;
29         if(P[u] == true) // Đỉnh u đã được chọn trước đó, bỏ qua
30             continue;
31
32         P[u] = true; // Đánh dấu đỉnh u đã được chọn
33         for(auto e : E[u]) {
34             int v = e.v;
35             long long w = e.w;
36
37             if(D[v] > D[u] + w) {
38                 D[v] = D[u] + w;
39                 h.push({v, D[v]});
40                 trace[v] = u;
41             }
42         }
43     }
44 }

```

Tìm lại đường đi ngắn nhất

Cũng giống như thuật toán Bellman-Ford, để tìm lại đường đi ngắn nhất từ S về u , ta sẽ bắt đầu từ đỉnh u , sau đó truy vết theo mảng $trace$ ngược về S .

```

vector<int> trace_path(vector<int> &trace, int S, int u) {
    if (u != S && trace[u] == -1) return vector<int>(0); // không có đường đi

    vector<int> path;
    while (u != -1) { // truy vết ngược từ u về S
        path.push_back(u);
        u = trace[u];
    }
    reverse(path.begin(), path.end()); // cần reverse vì đường đi lúc này là t

```

```
    return path;
}
```

3. Thuật toán Floyd-Warshall

Thuật toán Floyd-Warshall dùng để giải quyết bài toán **đường đi ngắn nhất mọi cặp đỉnh** (All-pairs shortest path), đồ thị **có thể có trọng số âm**.

Bài toán

Cho đồ thị gồm N đỉnh và một ma trận trọng số W , trong đó ô (i, j) cho biết rằng có một đường đi trực tiếp từ $i \rightarrow j$ với trọng số là $W_{i,j}$. Yêu cầu tìm đường đi ngắn nhất giữa **mọi cặp đỉnh** trên đồ thị.

Giới hạn bài toán: $1 \leq N \leq 100$

trong đó ô (i, j) cho biết đường đi trực tiếp từ $i \rightarrow j$ có trọng số là $W[i, j]$.

Output: Ma trận kết quả trong đó giá trị ô (i, j) là đường đi ngắn nhất từ $i \rightarrow j$.

Sample Input

1	5
2	0 4 2 1 6
3	7 0 1 2 4
4	2 3 0 1 2
5	4 5 5 0 1
6	6 8 7 3 0

Sample Output

1	0 4 2 1 2
2	3 0 1 2 3
3	2 3 0 1 2
4	4 5 5 0 1
5	6 8 7 3 0

Ý tưởng của thuật toán.

Ý tưởng của thuật toán này là: "Liệu chúng ta có thể chèn một đỉnh k vào đường đi ngắn nhất giữa 2 đỉnh u và v ?".

- Ví dụ như có một đường đi ngắn nhất từ $0 \rightarrow 4$ như sau: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Vậy việc tính đường đi ngắn nhất từ $0 \rightarrow 4$ hoàn toàn có thể được chia thành tính đường đi ngắn nhất từ $0 \rightarrow 2$ sau đó cộng với

đường đi ngắn nhất từ $2 \rightarrow 4$. Tương tự thế đường đi ngắn nhất từ $0 \rightarrow 2$ và $2 \rightarrow 4$ lại tiếp tục được phân hoạch thành những đường đi ngắn nhất khác đơn giản hơn và tối ưu hơn.

Ta nhận thấy có một cấu trúc đệ quy, chia nhỏ bài toán ở đây. Ý tưởng này cho phép chúng ta thực hiện một thuật toán mang hương vị quy hoạch động như sau:

- Gọi $D(u, v, k)$ là đường đi ngắn nhất, trong đó ta chỉ được đi qua k đỉnh đầu tiên (có số thứ tự từ 0 đến $k - 1$), ngoại trừ chính u và v . Ta có công thức truy hồi:
 - $D(u, v, 0) = W_{u,v}$ (không được dùng đỉnh nào ngoài chính u, v).
 - $D(u, u, k) = 0$
 - $D(u, v, k) = \min$ của 2 trường hợp:
 - $D(u, v, k - 1)$: ta không dùng đỉnh k làm trung gian, giữ nguyên đường đi cũ.
 - $D(u, k, k - 1) + D(k, v, k - 1)$: ta dùng đỉnh k làm trung gian, từ đường đi $u \rightarrow v$ thành đường đi $u \rightarrow k \rightarrow v$.

Đến đây ta có thể sử dụng trực tiếp công thức quy hoạch động để cài đặt thuật toán. Tuy nhiên, để đảm bảo bộ nhớ, ta có thể tính các $D(u, v, k)$ với k lần lượt từ 1 đến N , và khi cài đặt chỉ cần lưu lại $D(u, v)$.

Cài đặt

- Định nghĩa:
 - $W[u, v]$ là giá trị đường đi trực tiếp từ $u \rightarrow v$.
 - $D[u, v]$ là giá trị đường đi ngắn nhất từ $u \rightarrow v$.
 - $trace[u, v]$ là mảng truy vết đường đi ngắn nhất từ $u \rightarrow v$
- Đồ thị sẽ được lưu dưới dạng **ma trận kề**. Ban đầu sẽ khởi tạo mọi $D[u, v] = W[u, v]$ vì khi chưa tối ưu gì thì đường đi trực tiếp luôn là đường đi ngắn nhất.
 - $trace[u, v]$ sẽ khởi tạo bằng u với mọi cặp u, v .
 - Nếu không có cạnh nối giữa u và v , coi như $W[u, v] = \infty$.
- Thuật toán chỉ cần một vòng lặp xét mọi đỉnh k như một đỉnh trung gian. Tiếp theo đến là 2 vòng lặp u, v , có ý nghĩa thử chèn đỉnh k vào giữa đường đi từ u đến v .
 - Nếu như $D[u, v]$ được tối ưu bằng đỉnh k , ta cập nhật thêm $trace[u, v] = trace[k, v]$
- Độ phức tạp của thuật toán là $O(N^3)$.

Code:

Thuật toán có thể được cài đặt rất dễ dàng chỉ với 3 vòng lặp:

```
void init_trace(vector<vector<int>> &trace) {
    int n = trace.size();
    for (int u = 0; u < n; u++) {
        for (int v = 0; v < n; v++) {
            trace[u][v] = u;
        }
    }
}
```

```

10 void floydWarshall(int n, vector<vector<long long>> &w, vector<vector<long long>> &D, vector<vector<int>> &trace) {
11     D = w;
12     init_trace(trace); // nếu cần dò đường đi
13
14     for (int k = 0; k < n; k++) {
15         for (int u = 0; u < n; u++) {
16             for (int v = 0; v < n; v++) {
17                 if (D[u][v] > D[u][k] + D[k][v]) {
18                     D[u][v] = D[u][k] + D[k][v];
19                     trace[u][v] = trace[k][v];
20                 }
21             }
22         }
23     }
24 }

```

Tìm lại đường đi ngắn nhất

Giống như hai thuật toán Bellman-Ford và Dijkstra, để tìm đường đi từ u đến v , ta sẽ bắt đầu từ v , truy ngược về u theo mảng trace đã tìm được.

```

1 vector<int> trace_path(vector<vector<int>> &trace, int u, int v) {
2     vector<int> path;
3     while (v != u) { // truy vết ngược từ v về u
4         path.push_back(v);
5         v = trace[u][v];
6     }
7     path.push_back(u);
8
9     reverse(path.begin(), path.end()); // cần reverse vì đường đi từ v ngược về u
10
11     return path;
12 }

```

4. Lưu ý

- Bảng so sánh các thuật toán được đề cập:





	Bellman-Ford	Dijkstra (cơ bản)	Dijkstra (trên đồ thị thưa)	Floyd-Warshall
Bài toán giải quyết	Đường đi ngắn nhất một nguồn	Đường đi ngắn nhất một nguồn	Đường đi ngắn nhất một nguồn	Đường đi ngắn nhất mọi cặp

	Bellman-Ford	Dijkstra (cơ bản)	Dijkstra (trên đồ thị thưa)	Floyd-Warshall
				đỉnh
Độ phức tạp	$O(N * M)$	$O(N^2 + M)$	$O(M \log N)$	$O(N^3)$
Sử dụng được cho trọng số âm	Có	Không	Không	Có
Tìm được chu trình âm	Có	Không	Không	Không






- Trong trường hợp có chu trình âm, thuật toán Floyd-Warshall có thể phải tính toán đến những giá trị rất nhỏ (về phía số âm), đủ để gây ra hiện tượng tràn số (thậm chí với N tương đối nhỏ). Cần phải chú ý đặc biệt đến trường hợp này khi cài đặt.
 - Một cách thường dùng để giải quyết trường hợp này là gán $D[u][v] = \max(D[u][v], -INF)$ ngay sau mỗi lần tối ưu, chặn không cho $D[u, v]$ xuống dưới hằng số âm vô tận.
- Thuật toán Floyd-Warshall có thứ tự 3 vòng lặp là $k \rightarrow u \rightarrow v$ thay vì $u \rightarrow v \rightarrow k$ (đỉnh trung gian phải được đặt ở vòng lặp ngoài cùng), đây là một nhầm lẫn tương đối phổ biến khi cài đặt.
- Heap không phải là cấu trúc dữ liệu duy nhất có thể sử dụng khi cài đặt Dijkstra dành cho đồ thị thưa. Ta có thể sử dụng bất cứ cấu trúc dữ liệu nào hỗ trợ các thao tác "xóa khỏi tập hợp", "cập nhật phần tử trong tập hợp", "tìm phần tử nhỏ nhất trong tập hợp". Thực tế cây nhị phân tìm kiếm (`std::set` trong C++) cũng là một lựa chọn phổ biến khi cài đặt thuật toán này.
- Với đồ thị thưa, không có trọng số âm, thay vì sử dụng thuật toán Floyd, ta có thể chạy thuật toán Dijkstra cải tiến N lần với N đỉnh nguồn để tìm đường đi ngắn nhất giữa mọi cặp đỉnh, với độ phức tạp tốt hơn thuật toán Floyd:
 - Ta có độ phức tạp của hai thuật toán:
 - Dijkstra cải tiến, N lần: $O(N * M \log N)$
 - Floyd-Warshall: $O(N^3)$
 - Như vậy, để Dijkstra N lần tốt hơn, ta cần có $N * M \log N < N^3$ suy ra $M < N^2 / \log N$ (tương tự như so sánh giữa hai cách cài đặt thuật toán Dijkstra).

Bài tập vận dụng.





Thuật toán Bellman-Ford:

- [Kattis - shortestpath3](#) 
- [CSES - High Score](#) 
- [CSES - Cycle Finding](#)  (tìm chu trình âm)
- [Kattis - xyzzy](#) 

Thuật toán Dijkstra:

- [Kattis - shortestpath1](#) 
- [Codeforces - Dijkstra?](#)  (truy vết đường đi)
- [CSES - Flight Discount](#) 
- [Codeforces - Reducing Delivery Cost](#) 
- [Các bài theo tag trên VNOJ](#) 

Thuật toán Floyd-Warshall:

- [CSES - Shortest Route II](#)  (trọng số không âm)
- [Kattis - allpairspath](#)  (trọng số có thể âm)
- [Codeforces - Greg and Graph](#) 
- [Các bài theo tag trên VNOJ](#) 

Được cung cấp bởi [Wiki.js](#)