

Thuật toán SPFA

Kiến thức cần biết

- Dijkstra (không bắt buộc)
- Duyệt đồ thị cơ bản

Giới thiệu

Cho bài toán như sau:



Cho một đồ thị có hướng có N đỉnh và M cạnh có trọng số. Tìm đường đi ngắn nhất từ một đỉnh bất kỳ S đến tất cả các đỉnh còn lại trong đồ thị.

Đây là một bài toán vô cùng quen thuộc. Cách xử lý thông dụng và hiệu quả cho bài toán này là sử dụng thuật toán Dijkstra. Tuy nhiên, ở một số trường hợp hiếm hoi mà đồ thị bao gồm cạnh âm, ta cần một thuật toán khác để có thể giải quyết được trường hợp này.

Trong bài viết này, chúng ta sẽ cùng tìm hiểu về một số lựa chọn khác để giải quyết bài toán trên.

Thuật toán Bellman-Ford

Thuật toán Bellman-Ford là một trong những thuật toán có thể được dùng để giải bài toán trên. Thuật toán được thực hiện như sau.

1. Tạo một mảng $Dist[1..n]$ để lưu kết quả cuối cùng của bài toán. Ban đầu $Dist[1..n] = +\infty$ và $Dist[s] = 0$.
2. Duyệt qua từng cạnh của đồ thị và cập nhật $Dist$ của hai đỉnh thuộc cạnh nếu có thể.
Giả sử cạnh nối từ u đến v có trọng số là w và $Dist[u] + w < Dist[v]$ thì ta cập nhật $Dist[v] = Dist[u] + w$.
3. Lặp lại bước hai $n - 1$ lần.

Code

```
1 struct Edge{
2     int u, v, w;
3 };
4 int N, M, S;
5
6 const int MaxN = 1e5 + 5;
7 const int Inf = 1e9;
```

```

8
9
10 int Dist[MaxN];
11 vector<Edge> E;
12
13 void solve() {
14     for(int i = 1 ; i <= N ; i++) {
15         Dist[i] = Inf;
16     }
17     Dist[S] = 0;
18     for(int i = 1 ; i < N ; i++) {
19         for(Edge e: E) {
20             if(Dist[e.v] > Dist[e.u] + e.w) {
21                 Dist[e.v] = Dist[e.u] + e.w;
22             }
23         }
24     }

```

Chứng minh và độ phức tạp

Dễ thấy độ phức tạp của thuật toán trên là $O(N \times M)$ với N là số đỉnh và M là số cạnh của đồ thị.

► Chứng minh

Ưu điểm của Bellman-Ford

Dù độ phức tạp lý thuyết tệ hơn khá nhiều so với Dijkstra, nhưng Bellman-Ford cũng có những ưu điểm và những điểm có thể cải thiện để thuật toán trở nên hiệu quả và có tính ứng dụng cao hơn.

1) Dễ hiểu và dễ cài đặt

Thuật toán Bellman-Ford có ý tưởng rất cơ bản và tự nhiên. Nếu gặp những bài toán hoặc subtask có giới hạn thấp thì Bellman-Ford có thể tiết kiệm thời gian cài đặt.

2) Có thể chạy trên đồ thị với cạnh có trọng số âm

Vấn đề lớn nhất mà các thuật toán tìm đường đi ngắn nhất như Dijkstra gặp phải khi chạy trên đồ thị có cạnh có trọng số âm là có thể tồn tại một chu trình mà tổng trọng số là âm. Khi đó, việc chạy lại liên tục trong chu trình này sẽ tạo ra vòng lặp vô tận do khoảng cách từ S đến các đỉnh trên đồ thị liên tục giảm.

Do thuật toán Bellman-Ford dựa trên ý tưởng không một đường đi dài nhất nào có quá $N - 1$ cạnh, nên nếu có chu trình có tổng trọng số âm thì sẽ tồn tại một đường đi có trên $N - 1$ và Dist vẫn có thể được cập nhật sau $N - 1$ bước. Chính vì vậy, kết hợp với ưu điểm thứ 2, thuật toán Bellman-Ford có thể tìm được chu trình với tổng trọng số âm và giải bài toán như bình thường trong trường hợp không có chu trình nào như thế.

Shortest Path Faster Algorithm (SPFA)

Mặc dù thời gian chạy lý thuyết tệ hơn nhiều so với Dijkstra nhưng trên một số đồ thị, Bellman-Ford có thời gian chạy rất ngắn. Giả sử trên một đồ thị mà đường đi ngắn nhất đến một đỉnh bất kỳ từ S chứa không quá 5 cạnh thì thực tế độ phức tạp chỉ là $O(5 \cdot M)$. Chúng ta có thể dừng việc duyệt qua các cạnh ngay khi không có đỉnh

nào có khoảng cách ngắn hơn từ S sau khi duyệt. (Tuy nhiên, cũng cần lưu ý rằng rất dễ để sinh test khiến Bellman-Ford có thời gian chạy tệ nhất).

Từ nhận xét trên, chúng ta đến với phần chính của bài viết này - **Shortest Path Faster Algorithm (SPFA)**.

Đúng như tên gọi, SPFA là phiên bản tối ưu hơn về thời gian của Bellman-Ford. Ở đoạn code phía trên, chúng ta lưu một danh sách các cạnh và duyệt qua tất cả các cạnh nếu như vẫn còn đỉnh chưa tối ưu. Có thể dễ dàng nhận ra chúng ta đã tốn khá nhiều lần duyệt với những đỉnh đã tối ưu sau những lần duyệt đầu tiên.

Để dễ hiểu hơn, giả sử chúng ta có một đồ thị dạng đường thẳng có độ dài N (đỉnh i nối với đỉnh $i + 1$) và cần tìm đường đi ngắn nhất từ đỉnh 1. Tuy nhiên, danh sách cạnh lại chạy ngược từ các cạnh từ N về 1. Như vậy chúng ta sẽ duyệt qua $N - 1$ cạnh đúng $N - 1$ lần vì mỗi lần duyệt qua $N - 1$ cạnh chỉ cải thiện được đúng một đỉnh duy nhất.

Ý tưởng chính là thay vì duyệt qua các cạnh, chúng ta sử dụng một hàng đợi ([queue](#)) để chứa các đỉnh mà đỉnh đó còn có thể cải thiện những đỉnh kề nó.

SPFA cũng có thể phát hiện được cycle với tổng trọng số âm vì về bản chất, SPFA chính là Bellman-Ford.

Code

```

1  #include<bits/stdc++.h>
2  typedef pair<int, int> ii;
3  const int MaxN = 1e5 + 5;
4  const int Inf = 1e9;
5  vector<vector<ii>> AdjList;
6  int Dist[MaxN];
7  int Cnt[MaxN];
8  bool inqueue[MaxN];
9  int S;
10 int N;
11 queue<int> q;
12
13 bool spfa() {
14     for(int i = 1 ; i <= N ; i++) {
15         Dist[i] = Inf;
16         Cnt[i] = 0;
17         inqueue[i] = false;
18     }
19     Dist[S] = 0;
20     q.push(S);
21     inqueue[S] = true;
22     while(!q.empty()) {
23         int u = q.front();
24         q.pop();
25         inqueue[u] = false;
26
27         for (ii tmp: AdjList[u]) {
28             int v = tmp.first;
29             int w = tmp.second;
30
31             if (Dist[u] + w < Dist[v]) {

```

```

32         if (Dist[u] + w < Dist[v]) {
33             Dist[v] = Dist[u] + w;
34             if (!inqueue[v]) {
35                 q.push(v);
36                 inqueue[to] = true;
37                 cnt[v]++;
38                 if (cnt[v] > n)
39                     return false;
40             }
41         }
42     }
43 }
44 return true;
}

```

Trong đó:

- Mảng *Dist* được dùng để lưu đáp án.
- Mảng *Cnt* để lưu lại số lần một đỉnh được cải thiện đáp án (phục vụ việc tìm cycle có tổng trọng số âm). Cũng như Bellman-Ford, nếu đáp án của một đỉnh được cải thiện quá N lần thì tồn tại cycle có tổng trọng số âm.
- Mảng *inqueue* dùng để kiểm tra xem đỉnh đã ở trong queue hay chưa vì chúng ta không muốn một đỉnh ở trong queue nhiều lần cùng một lúc. Đỉnh sẽ được đẩy vào queue ngay khi đáp án của đỉnh cải thiện.

Độ phức tạp và một vài cách tối ưu

Độ phức tạp


Về cơ bản, cách implement của SPFA gần giống với BFS hay Dijkstra. Tuy nhiên khác biệt lớn nhất là việc queue trong BFS hay **priority_queue** trong Dijkstra được sử dụng để đảm bảo các đỉnh tuân theo một thứ tự nhất định (đáp án tốt nhất có thể). Queue trong SPFA chỉ đơn giản là để lưu lại các đỉnh còn có khả năng cải thiện, do vậy độ phức tạp về mặt lý thuyết của SPFA trong trường hợp tệ nhất vẫn là $O(MN)$.

Tuy nhiên, thời gian chạy trung bình của SPFA là rất nhanh. Một số thử nghiệm cho thấy thời gian chạy trên đồ thị trung bình của SPFA chỉ là $O(M)$. Tuy nhiên, việc sinh test để chống lại SPFA vẫn là rất dễ tuy có khó hơn Bellman-Ford thông thường.

Khi nào nên sử dụng SPFA

Trên thực tế, có thể thấy rằng SPFA hoạt động tốt hơn nhiều khi đồ thị thưa, vì thế bạn đọc có thể cân nhắc sử dụng SPFA cho những bài toán với đồ thị có giới hạn số đỉnh và số cạnh gần bằng nhau (ví dụ $N \leq 10^5, M \leq 2 \times 10^5$).

Đặc biệt, trong những cuộc thi chấm điểm theo test (thay vì subtask) hoặc trong những bài toán đồ thị được sinh random, SPFA có thể sẽ rất hiệu quả trong nhiều bài toán trong trường hợp cần tối ưu thời gian.

Ngoài ra, SPFA còn nên được sử dụng trong những bài toán có cạnh âm. Ví dụ như một số cách cài đặt thuật toán min-cost max flow phổ biến sử dụng SPFA thay vì dijkstra trong việc tìm đường tăng luồng ngắn nhất **MCMF** 

Một số cách tối ưu

Thời gian chạy của SPFA phụ thuộc khá nhiều vào thứ tự của các đỉnh trong queue. Nếu thay queue bằng `priority_queue` thì cơ bản thuật toán gần như biến thành Dijkstra.

Nếu không muốn sử dụng `priority_queue` nhưng cũng không muốn thứ tự các đỉnh trong hàng đợi quá tệ, có thể sử dụng phương pháp tối ưu cơ bản như sau.

1) Small Label First

Giả sử chúng ta có một đỉnh v đang chuẩn bị được cho vào queue. Nếu $Dist[v] < Dist[front(Q)]$ với $front(Q)$ là đỉnh đầu tiên trong queue thì chúng ta sẽ đẩy v vào đầu thay vì cuối hàng đợi.









2) Large Label Last

Giả sử đỉnh chuẩn bị xét có $Dist$ tệ hơn trung bình của các đỉnh trong hàng đợi, chúng ta đẩy đỉnh này xuống cuối và xét các đỉnh khác trước.

Bạn đọc có thể kết hợp cả hai cách tối ưu và nghĩ ra những cách tối ưu tương tự (đẩy những đỉnh có $Dist$ nhỏ hơn lên đầu, hạn chế dùng những đỉnh $Dist$ to) mà không tốn nhiều thời gian.

Do thời gian có thể phụ thuộc nhiều vào thứ tự các đỉnh, việc đảo lộn thứ tự theo một cách nhất định hoặc ngẫu nhiên đôi khi có thể tránh được những test được sinh để chống lại SPFA.

Bài tập luyện tập

- [E-OLYMP #1453 "Ford-Bellman"](#) 
- [UVA #423 "MPI Maelstrom"](#) 
- [UVA #534 "Frogger"](#) 
- [UVA #10099 "The Tourist Guide"](#) 
- [UVA #515 "King"](#) 
- [UVA 12519 - The Farnsworth Parabox](#) 
- [CSES - High Score](#) 
- [CSES - Cycle Finding](#) 

Được cung cấp bởi [Wiki.js](#)