2.4 Greedy (Thuật toán tham lam)

Created by TUNG DUC NGUYEN tung2.nguyen, last modified on 2021/05/21

Với bài toán Knapsack (đã gặp ở phần vét cạn và quy hoạch động), giả sử bạn chưa biết cách áp dụng quy hoạch động để giải quyết bài toán này. Bạn sẽ làm cách nào?

> *Bài toán KNAPSACK:*
>
> *Trong siêu thị có n (n ≤ 2000) gói hàng, gói hàng thứ i có trọng lượng là W(i) {1, 2, ..., i-1 2000 và giá trị V(i) ≤ 2000. Một tên trộm đột nhập vào siêu thị, tên trộm mang theo cái túi có thể mang được tối đa trọng lượng là M ≤ 2000. Hỏi tên trộm sẽ lấy đi những gói hàng nào để tổng giá trị là lớn nhất.*

Rõ ràng với số lượng lớn, n = 2000, cách giải quyết liệt kê sẽ không giải quyết được bài toán do thời gian quá lớn (10^576 tỷ năm).

Nếu bạn là tên trộm, thì việc áp dụng một cách nào đó để giải quyết bài toán nhanh nhất có thể, vài cách sau đây sẽ có thể được áp dụng:

1. Ưu tiên lấy vật có giá trị lớn hơn.
2. Ưu tiên lấy vật có trọng lượng nhỏ hơn.
3. Ưu tiên lấy vật có giá trị/trọng lượng hơn.

› Giải thích cách chọn và phản ví dụ

Rõ ràng việc áp dụng một trong ba cách này tốc độ sẽ nhanh hơn rất nhiều O(nlogn) để sắp xếp và O(n) để lựa chọn.

Tuy nhiên, không phải lúc nào cũng là kết quả tốt nhất:

Với cách 1 sẽ dễ dàng giải quyết bài toán V = {1, 2, 2, 4, 10}, W = {1, 3, 2, 12, 4}, M = 15 với kết quả tốt nhất là 15 với việc lựa chọn các gói hàng (5, 3, 2, 1)

Nhưng lại cho kết quả chưa tối ưu là 14 với bài toán V = {1, 2, 2, 10, 11}, W = {1, 3, 2, 4, 12}, M = 15 với việc lựa chọn các gói hàng (5, 3, 1) (kết quả tốt nhất là 15).

Với cách 2 sẽ dễ dàng giải quyết bài toán V = {1, 2, 2, 10, 11}, W = {1, 3, 2, 4, 12}, M = 15 với kết quả tốt nhất là 15 với việc lựa chọn các gói hàng (1, 2, 3, 4)

Nhưng lại cho kết quả chưa tối ưu là 10 với bài toán V = {1, 2, 2, 5, 11}, W = {1, 3, 2, 8, 12}, M = 15 với việc lựa chọn các gói hàng (1, 2, 3, 4) (kết quả tốt nhất là 14).

Với cách 3 sẽ dễ dàng giải quyết bài toán V = {1, 2, 2, 5, 11}, W = {1, 3, 2, 8, 12}, M = 15 với kết quả tốt nhất là 14 với việc lựa chọn các gói hàng (1, 2, 5) (tỷ lệ giá trị/trọng lượng tương ứng là : (1, 0.67, 1, 0.63, 0.92))

Nhưng lại cho kết quả chưa tối ưu là 10 với bài toán V = {1, 2, 2, 5, 11}, W = {1, 3, 2, 8, 13}, M = 15 với việc lựa chọn các gói hàng (5, 3, 1) (kết quả tốt nhất là 13 với tỷ lệ giá trị/trọng lượng tương ứng là : (1, 0.67, 1, 0.63, 0.85)).

Ta có thể áp dụng cả ba cách rồi chọn kết quả tốt nhất → vẫn có rất nhiều trường hợp chưa đưa ra kết quả tối ưu, bạn hãy lấy ví dụ?

1. Thuật toán tham lam

Thuật toán tham lam là **bất kỳ thuật toán** nào tuân theo **kinh nghiệm giải quyết vấn đề** để đưa ra **tối ưu cục bộ** ở **mỗi giai đoạn**.

Trong hầu hết các vấn đề, việc áp dụng tham lam không tạo ra một giải pháp tối ưu nhưng đem lại các giải pháp tối ưu cục bộ gần đúng với một giải pháp tối ưu toàn cục trong thời gian hợp lý.

Nói chung thuật toán tham lam có năm phần:

1. Một tập hợp các cấu hình để dựa vào đó đưa ra một giải pháp - *A candidate set*
2. Một hàm lựa chọn để chọn cấu hình tốt nhất để thêm vào giải pháp - *A selection function*
3. Một hàm khả thi để xác định xem một cấu hình có thể được đưa vào giải pháp không? - *A feasibility function*
4. Một hàm mục tiêu để chỉ định một giá trị cho một giải pháp hoặc một phần giải pháp - *An objective function*
5. Một hàm giải pháp để cho biết khi nào đã xác định ra một giải pháp hoàn chỉnh. - *A solution function*

Ví dụ: Với cách giải quyết số 1 - ưu tiên lấy vật có giá trị lớn - năm phần như sau:

1. Một tập hợp các cấu hình: Tập các tập con một phần tử { 1 }, { 2 }, .. , { n }
2. Hàm lựa chọn cấu hình tốt nhất: Sắp xếp tập cấu hình theo giá trị giảm dần. Đứng đầu là tốt nhất.
3. Hàm khả thi: So sánh trọng lượng của cấu hình hiện tại với khả năng còn lại của túi.
4. Hàm mục tiêu: Thêm giá trị của túi khi khả thi hoặc không thêm giá trị khi không khả thi.
5. Hàm giải pháp: Tổng trọng lượng đã chọn bằng khả năng M của túi.

Các lựa chọn của thuật tham lam:

Thuật toán tham lam có thể đưa ra bất kỳ lựa chọn có vẻ tốt nhất lúc này và sau đó giải quyết các vấn đề phụ phát sinh sau đó. Sự lựa chọn được thực hiện có thể phụ thuộc vào các lựa chọn được thực hiện từ trước đó, không phụ thuộc vào các lựa chọn trong tương lai hoặc tất cả các bài toán con. Hay nói cách khác, nó không bao giờ xem xét lại các lựa chọn của nó. Khác với quy hoạch động, tại mỗi bước quy hoạch động đưa ra quyết định dựa trên tất cả các quyết định thực hiện từ các bước trước đó và có thể xem xét lại cách giải quyết của các bước trước.

2. Một số bài toán áp dụng tham lam

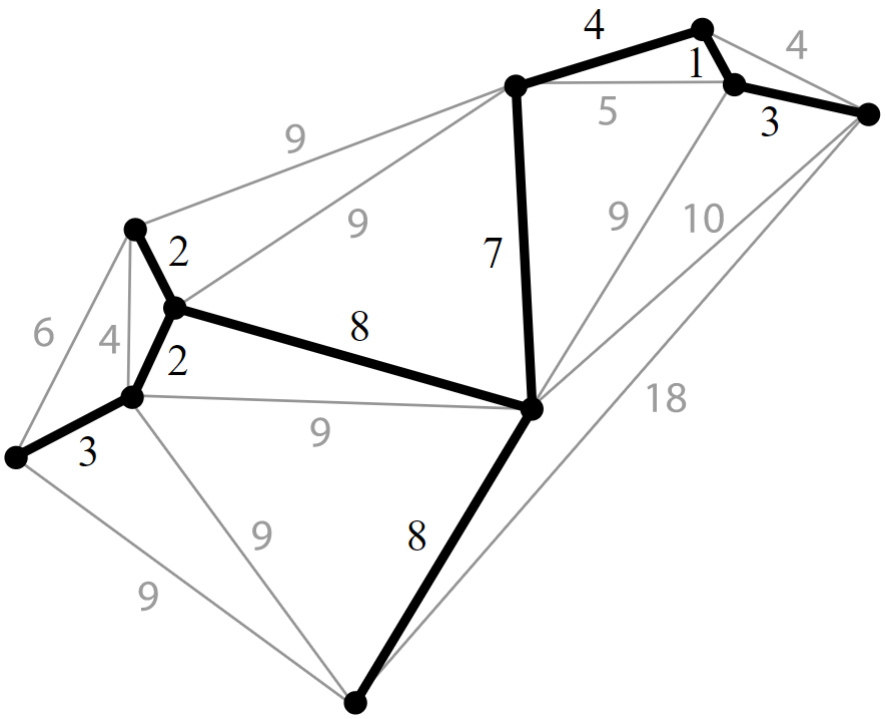
Ứng dụng của thuật toán tham lam là rất rộng, ta sẽ xem vài ví dụ áp dụng thuật toán tham lam sau: Cây khung nhỏ nhất (Minimum Spanning Tree) với hai cách tham lam Kursal và Prim, bài toán tìm đường ngắn nhất giữa hai đỉnh trong một đồ thị có trọng số không âm (Dijkstra).

2.1 Minimum Spanning Tree

Cần tìm hiểu 2.4.2 Graph and Graph Representation (Đồ thị và Biểu diễn của đồ thị) và 2.4.3 Tree (Cây) trước khi tiếp tục.

https://www.spoj.com/problems/MST/

Cây khung nhỏ nhất (MST) hoặc cây khung có trọng số nhỏ nhất là tập hợp con của các cạnh của đồ thị vô hướng có trọng số cạnh được kết nối, kết nối tất cả các đỉnh với nhau, không có bất kỳ chu trình nào và với tổng trọng số cạnh tối thiểu có thể.



Như vậy cây khung nhỏ nhất là một cây có (V-1) cạnh trong đó V là số đỉnh của đồ thị đã cho.

Ứng dụng của bài toán cây khung nhỏ nhất:

- Thiết kế mạng (điện thoại, điện, thủy lực, cáp tivi, máy tính, đường bộ).

  Ví dụ với mạng điện thoại. Bạn có một doanh nghiệp với nhiều văn phòng; bạn muốn thuê đường dây điện thoại để kết nối chúng với nhau; và công ty điện thoại tính các khoản tiền khác nhau để kết nối các cặp văn phòng khác nhau. Bạn muốn một tập hợp các đường kết nối tất cả các văn phòng của mình với tổng chi phí tối thiểu.

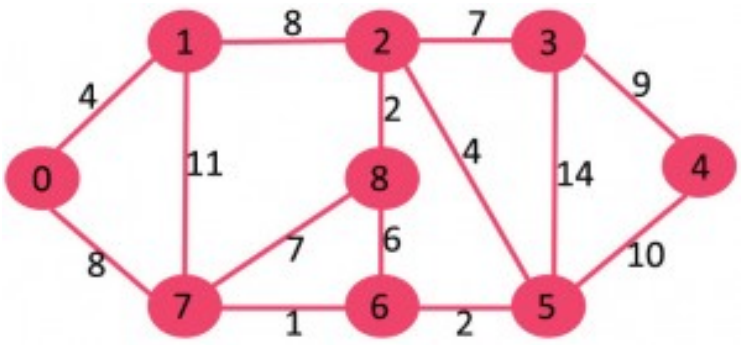
- Giải quyết gần đúng với bài toán hiện tại chưa có lời giải tối ưu (ví dụ bài Travelling Sale Man).
- Và rất nhiều bài toán không trực tiếp khác...

2.2 Kruskal's Algorithm

Thuật toán của Kruskal có ba bước sau:

1. Sắp xếp tất cả các cạnh theo thứ tự tăng dần theo trọng số.
2. Chọn cạnh nhỏ nhất. Kiểm tra xem nó có tạo thành một chu trình với cây khung đang xây dựng hay không. Nếu không tạo thành chu trình, thì thêm cạnh nào vào cây khung đang xây dựng. Ngược lại, loại bỏ cạnh này.
3. Lặp lại bước 2 cho đến khi có (V-1) cạnh trong cây khung.

Ví dụ: Cho đồ thị:



› Giải thích ví dụ cho Kruskal

Bước 1: Sắp xếp tất cả các cạnh theo thứ tự tăng dần theo trọng số.

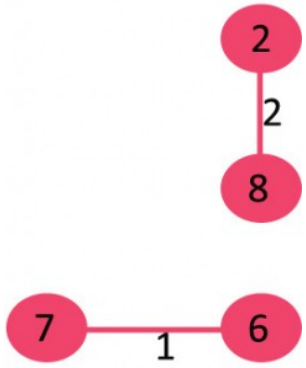
After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

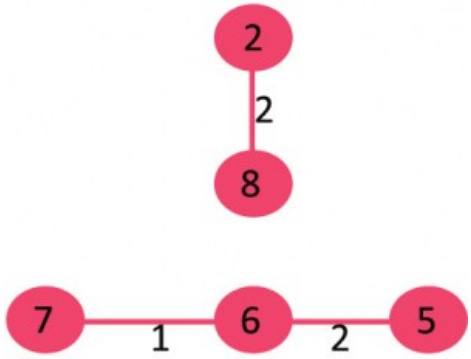
Bước 2 + 3: Chọn (V-1) = 8 cạnh nhỏ nhất trong danh sách sao cho không tạo ra một chu trình

Cạnh thứ 1: 7-6

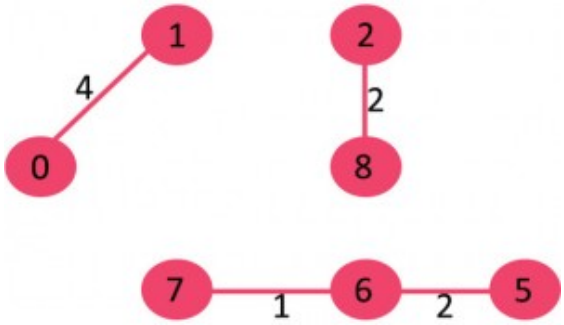
Cạnh thứ 2: 8-2



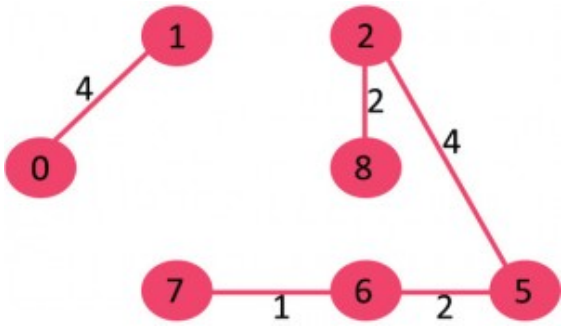
Cạnh thứ 3: 6-5



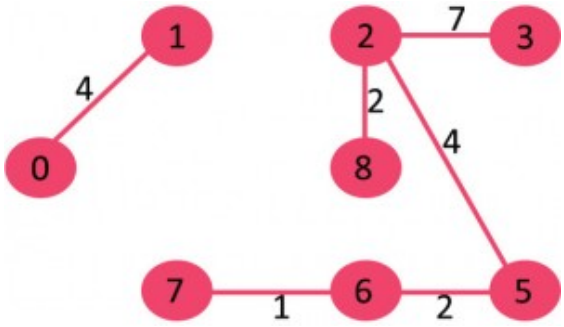
Cạnh thứ 4: 0-1



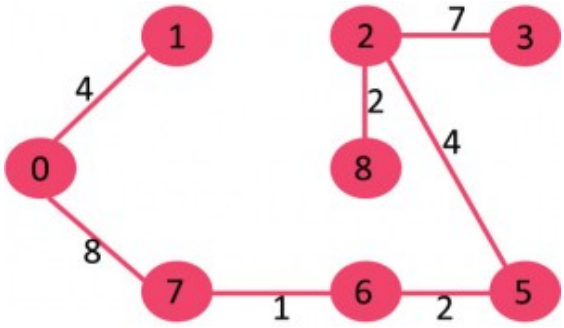
Cạnh thứ 5: 2-5



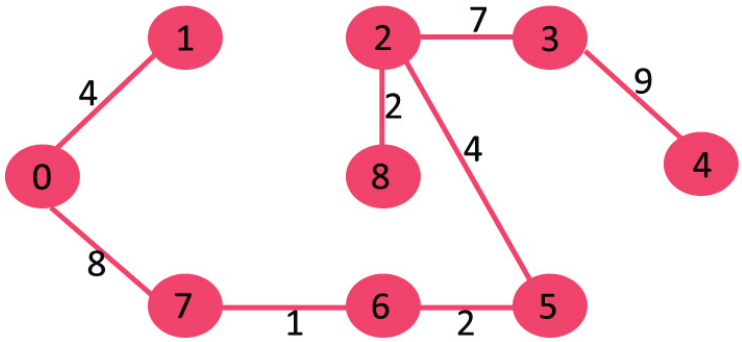
Cạnh thứ 6: 8-6 → tạo ra chu trình → 2-3



Cạnh thứ 7: 7-8 → tạo ra chu trình → 0-7



Cạnh thứ 8: 1-2 → tạo ra chu trình → 3-4



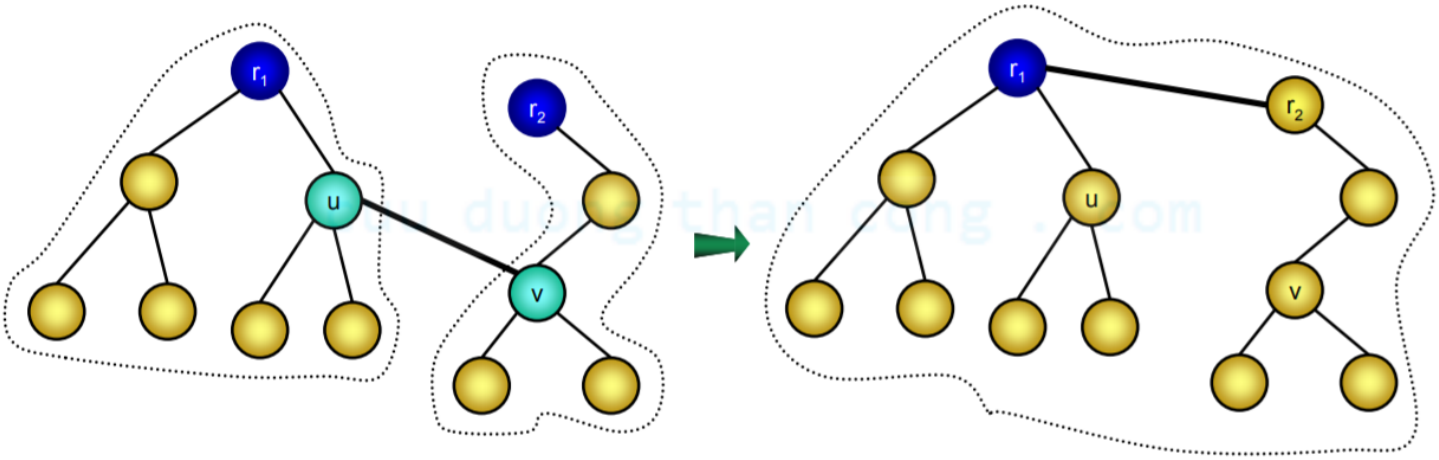
(V-1) 8cạnh đã được chọn → dừng và thông báo kết quả.

Làm thế nào để kiểm tra việc thêm một cạnh có tạo thành chu trình hay không?

Để ý rằng nếu hai đỉnh đã thuộc một cây thì chắc chắn nó sẽ tạo thành chu trình. Vậy câu hỏi sẽ là làm thế nào để kiểm tra hai đỉnh thuộc cùng một cây hay không?

Ta thấy rằng nếu hai đỉnh có cùng gốc thì lúc đó hai đỉnh sẽ cùng một cây.

Khi hai đỉnh thuộc hai cây khác nhau, ta phải hợp nhất hai cây nào vào làm một, việc này được thực hiện đơn giản bằng cách chọn một trong hai gốc làm cha của gốc còn lại trở thành gốc của cây mới.



Để tránh việc làm cho việc tìm gốc hoạt động chậm, ta có thể dùng cách cây nào có ít nút hơn sẽ là con của gốc cây kia.

Ta có phần triển khai như sau:

```
bool kruskalMST() {
    int parent[V]; //Chưa có đỉnh nào nằm trong MST
    int count[V]; // count[i] là số lượng nút con nếu coi nút i là gốc

    for (int i = 0; i < V; ++i) {
        parent[i] = -1; // Tất cả là cây có một nút
        count[i] = 1;
    }

    qsort(E);  //Sắp xếp lại các cạnh của đồ thị theo thứ tự tăng dần

    while (mst.size() < V - 1 && E.size() > 0) {
        e = E.front(); //Lấy đỉnh e có trọng số nhỏ nhất
        E.pop();

        if (!makeUnion(e, parent, count)) //Nếu không thể thêm cạnh e vào cây khung nhỏ nhất
            continue;

        mst.push(e); // Thêm cạnh e vào trong MST
    }
    return mst.size() == V - 1; //thông báo kết quả của việc tìm cây khung nhỏ nhất
}

bool makeUnion(Edge e, int parent[] , int count[]) {
    int parentStart = e.start;
    while (parent[parentStart] != -1) parentStart = parent[parentStart]; // Tìm gốc của cây start
    int parentEnd = e.end;
    while (parent[parentEnd ] != -1) parentEnd = parent[parentEnd];      // Tìm gốc của cây end

    if (parentStart == parentEnd) return false; // Nếu hai đỉnh có cùng gốc -> thêm cạnh e sẽ tạo ra

    if (count[parentStart] > count[parentEnd]) { // Nếu cây của nút start có nhiều nút hơn, gộp cây củ
        parent[parentEnd] = parentStart;
        count[parentStart] += count[parentEnd]; //Tăng số lượng nút của cây của nút start
    } else {
        parent[parentStart] = parentEnd;
        count[parentEnd] += count[parentStart]; //Tăng số lượng nút của cây của nút end
    }

    return true;
}
```

Độ phức tạp của thuật toán Kruskal: O(ElogE) hoặc O(ElogV). Trong đó việc sắp xếp tốn O(ElogE) thời gian. Sau khi sắp xếp ta phải duyệt tất cả các cạnh (O(E)), với mỗi cạnh ta hợp nhất các nhóm của hai đỉnh của cạnh này O(V).

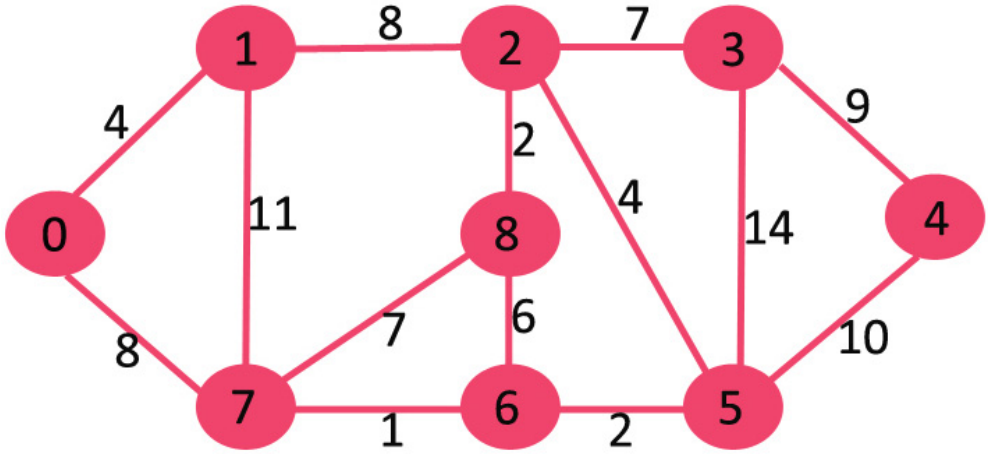
2.3 Prim's Algorithm

Thuật toán của Prim có ba bước sau:

1. Khởi tạo một cây với một đỉnh duy nhất, đỉnh này được chọn tùy ý từ đồ thị V.
2. Phát triển cây bằng cách thêm một cạnh vào cây đang có. Cạnh này là cạnh có trọng số nhỏ nhất trong những cạnh nối với cây từ những đỉnh chưa thuộc cây.

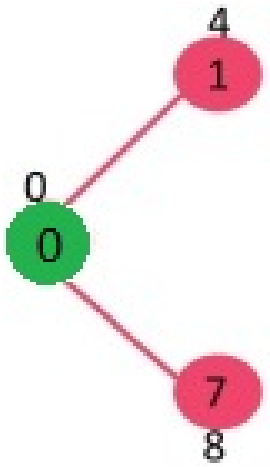
3. Lặp lại bước 2 cho đến khi tất cả các đỉnh đều nằm trong cây (tức là lặp lại V-1 lần).

Ví dụ: Tìm cây khung nhỏ nhất cho đồ thị:

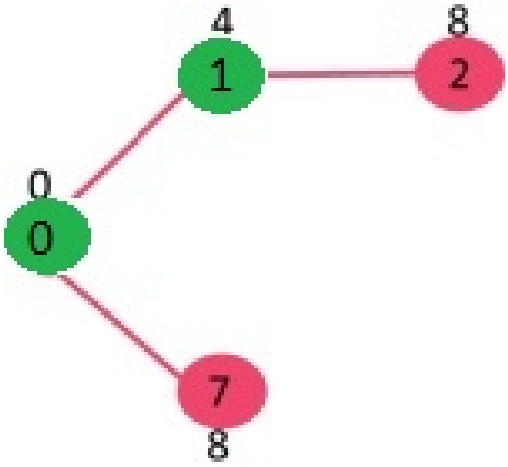


› Giải thích ví dụ cho thuật toán Prim

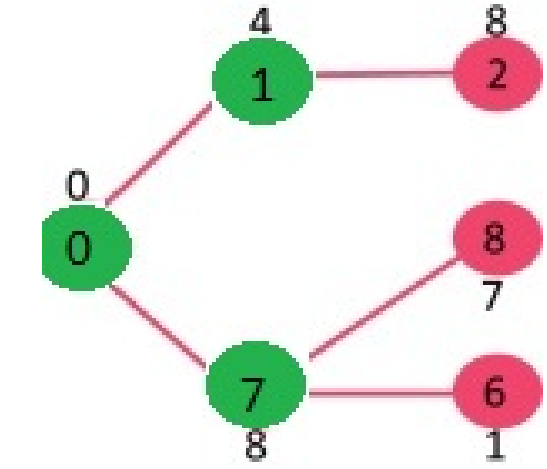
Ta chọn đỉnh **0** là đỉnh đầu tiên của cây, khi đó sẽ có hai cạnh **0-1** và **0-7** là các cạnh nối với cây từ những đỉnh chưa thuộc cây.



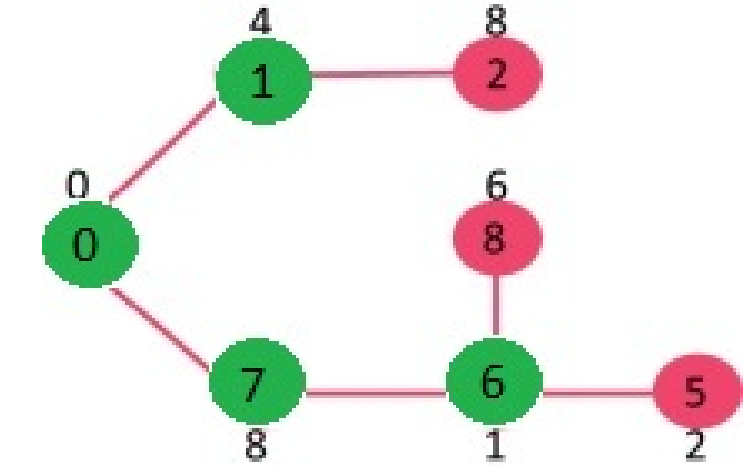
Vì trọng số cạnh **0-1** là **4** là nhỏ nhất trong những cạnh nối với cây từ những đỉnh chưa thuộc cây. Khi đó sẽ có hai cạnh là **1-2** và **0-7** là các cạnh nối với cây từ những đỉnh chưa thuộc cây.



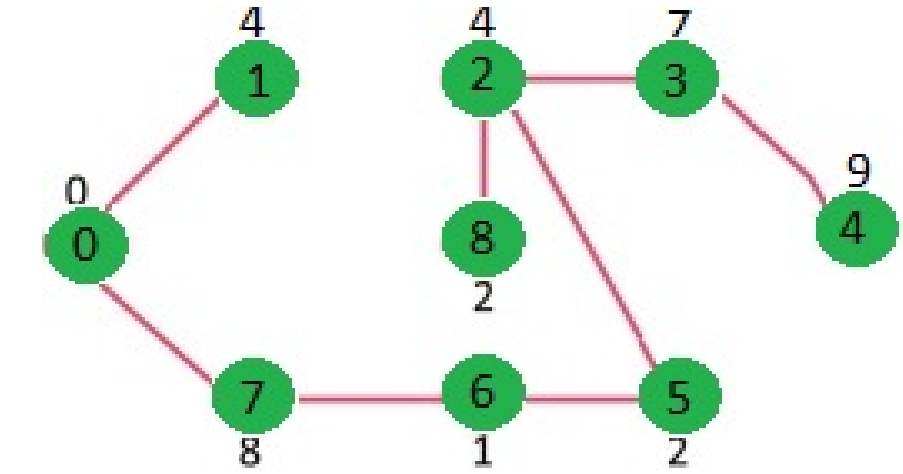
Vì trọng số cạnh **0-7** là **8** là nhỏ nhất trong những cạnh nối với cây từ những đỉnh chưa thuộc cây. Khi đó sẽ có hai cạnh là **1-2**, **7-8** và **7-6** là các cạnh nối với cây từ những đỉnh chưa thuộc cây.



Vì trọng số cạnh **7-6** là 1 là nhỏ nhất trong những cạnh nối với cây từ những đỉnh chưa thuộc cây. Khi đó sẽ có hai cạnh là **1-2**, **6-8** và **6-5** là các cạnh nối với cây từ những đỉnh chưa thuộc cây.



Lặp lại các bước như vậy, sau V-1 lần ta sẽ chọn được V-1 cạnh vào cây như sau:



Việc triển khai thuật toán Prim, ta có thể triển khai như sau:

```
bool primMST() {
    int parent[V]; // Cây khung sẽ được lưu trên mảng parent, với ý nghĩa nốt cha của nốt i trong cây
    int key[V];    // Mảng lưu trữ khoảng cách với ý nghĩa key[i] là khoảng cách từ các đỉnh i đến cây
    bool mstSet[V];// Mảng kiểm tra với ý nghĩa đỉnh thứ i có mstSet[i] == true nghĩa là đỉnh i đang n

    for (int i = 0; i < V; ++i) { //Khởi tạo ban đầu
        key[i] = INT_MAX;  // Đầu tiên chưa có đỉnh nào được chọn, tất cả khoảng cách là vô cùng
        mstSet[i] = false; // Khởi tạo đỉnh i chưa được chọn vào cây khung.
```

```
    }

    key[0] = 0;     // Chọn đỉnh bắt đầu là đỉnh 0. Khoảng cách từ điểm 0 đến cây khung hiện tại là 0.
    parent[0] = -1; // Vì 0 là đỉnh bắt đầu nên là gốc của cây. Nên nó không có cha.

    for (int i = 0; i < V - 1; ++i) {
        int u = minKey(key, mstSet); // Lấy ra đỉnh u có khoảng cách nhỏ nhất đến cây khung hiện tại.

        if (u == -1) return false;   // Nếu không tìm ra được đỉnh nào -> đồ thị không liên thông nên

        mstSet[u] = true;            // Đánh dấu đỉnh u đã được chọn vào cây khung

        for (int v = 0; v < V; ++v) {
            /* Từ đỉnh u đã chọn vào cây, kiểm tra tất cả các cạnh nối với đỉnh u.
               Nếu khoảng cách từ đỉnh vừa được chọn vào cây u đến đỉnh v không thuộc cây mà nhỏ hơn k
               key[v] hiện tại là khoảng cánh nhỏ nhất từ đỉnh v đến cây khi đỉnh u chưa được thêm vào
               Thì cập nhật lại giá trị cho key[v]. */
            if (mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;        // Cập nhật lại parent của đỉnh v là đỉnh u
                key[v] = graph[u][v]; // Cập nhật lại giá trị cho key[v]
            }
        }
    }
    return true;
}

int minKey(int key[], bool mstSet[]) {
    int min = INT_MAX;
    int min_index = -1;
    for (int v = 0; v < V; ++v) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
}
```

Như vậy độ phức tạp của thuật toán là O(V^2).

Nếu biểu diễn đồ thị dưới danh sách kề và việc tìm minKey dùng binary heap (priority queue) thì độ phức tạp sẽ là O(ElogV) → Bài tập nâng cao về nhà.

2.4 Dijkstra's shortest path algorithm

Bài toán tìm đường đi ngắn nhất là tìm đường đi giữa hai đỉnh (hoặc hai nút) trong đồ thị sao cho tổng trọng số cấu thành của các cạnh của nó là lớn nhất.

Bài toán tìm đường đi ngắn nhất thường áp dụng cho việc tìm đường giữa các điểm trên thực tế chẳng hạn như tìm đường lái xe trên các bản đồ như GoogleMap, Apple Map,....

Hoặc trong hệ thống mạng và viễn thông, đó là việc tìm đường đi với độ trễ nhỏ nhất.

Bài toán được phát biểu dưới dạng tổng quát như sau:

Cho đồ thị có trọng số G = (V, E), hãy tìm một đường đi ngắn nhất xuất phát từ đỉnh S € V đến đỉnh F € V.

Độ dài của đường đi này ta sẽ ký hiệu là d[S,F] và gọi là khoảng cách từ S đến F. Nếu không tồn tại đường đi từ S tới F thì ta sẽ đặt khoảng cách đó là +∞.

Nếu như đồ thị có chu trình âm (chu trình có độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này với số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào đó.

Trong trường hợp như vậy, có thể đặt vấn đề tìm đường đi cơ bản (đường đi không có đỉnh lặp lại) ngắn nhất. Đây là một vấn đề phức tạp ta không đề cập ở đây.

Chúng ta sẽ xét đến đồ thị không có chu trình âm. Với dạng đồ thị này, nếu biết được khoảng cách từ S đến tất cả những đỉnh khác thì đường đi ngắn nhất từ S đến F có thể dễ dàng tìm qua một thuật toán sau:

Gọi c[u,v] là trọng số của cạnh [u,v], quy ước rằng c[v,v] = 0 với mọi v € V và c[u,v] = +∞ nếu như (u,v) không nằm trong tập E (không có đường đi từ u đến v). Đặt d[S,v] là khoảng cách từ S đến v. Để tìm đường đi từ S tới F, ta có thể thấy rằng luôn luôn tồn tại đỉnh F1 ≠ F sao cho:

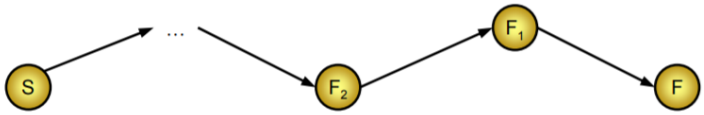
$$d[S,F] = d[S,F1] + c[F1,F]$$

(độ dài đường đi ngắn nhất từ S → F = độ dài đường đi ngắn nhất từ S → F1 + Chi phí đi từ F1 → F)

Đỉnh F1 đo là đỉnh liền kề trước F trong đường đi ngắn nhất từ S tới F. Nếu F1 ≡ S thì đường đi ngắn nhất là đường đi trực tiếp theo cạnh (S,F). Nếu F1 ≠ F thì vấn đề trở thành tìm đường đi ngắn nhất từ S đến F1. Ta lại tìm được một F2 khác F và F1 để:

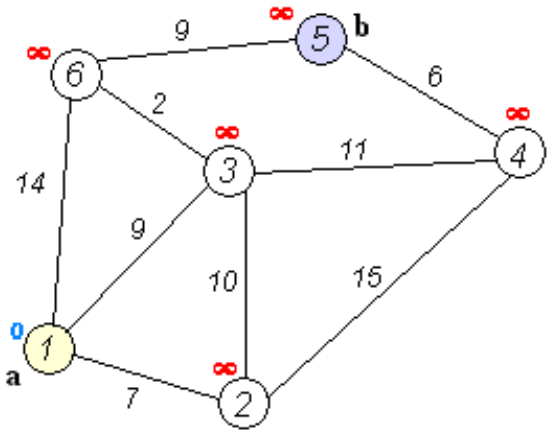
$$d[S,F1] = d[S,F2] + c[F2,F1]$$

Cứ tiếp tục như vậy theo một hữu hạn số bước, ta có dãy F,F1,F2,... không chứa đỉnh lặp lại và kết thúc ở S. Lật ngược thứ tự thì ta có đường đi ngắn nhất từ S đến F:



Thuật toán của Dijkstra là một thuật toán tìm đường đi ngắn nhất trong đồ thị (không có trọng số âm) cũng dựa trên nguyên lý trên. Các bước của thuật toán như sau:

1. Đánh dấu toàn bộ các nút là chưa thăm. Tạo ra một tập chứa toàn bộ các nút chưa thăm gọi là *unvisited set*.
2. Gán cho mọi nốt khoảng cách dự kiến từ S. Đặt bằng 0 cho nút S d[S] = 0 và +∞ cho tất cả các nút khác d[i] = +∞ với mọi i ≠ S. Đặt nút hiện tại u là S.
3. So sánh khoảng cách dự kiến hiện tại của các đỉnh v lân cận với u và khoảng cách nếu đi từ S → u → v. Nếu đi qua u gần hơn, cập nhật lại khoảng cách dự kiến của v là d[v].
4. Loại bỏ nút u ra khỏi *unvisited set*.
5. Nếu u là nút đích (u = t) hoặc khoảng cách đến nút u là vô cùng (d[u] = +∞) kết thúc thuật toán.
6. Lấy nút u nằm ở trong *unvisited set* mà có d[u] là nhỏ nhất.



Cài đặt của thuật toán dijkstra:

```
void dijkstra(int graph[V][V], int s) { // Tìm đường đi ngắn nhất trên đồ thị V đến tất cả các đỉnh củ
    int d[V];    // Mảng lưu trữ khoảng cách từ đỉnh s đến các đỉnh còn lại trên đồ thị
    int pre[V];  // Mảng truy vết đường ngắn nhất
```

```
    bool unvisited[V];  // Mảng để kiểm tra đỉnh đã được thăm hay chưa
    for (int i = 0; i < V; ++i) {
        dist[i] = INT_MAX;  // Ban đầu tiên chưa biết đường đi.
        unvisited[i] = true;// Chưa đỉnh nào được thăm cả
        pre[i] = s;
    }

    d[s] = 0;    // Đường đi từ s đến chính nó luôn luôn là 0
    pre[s] = -1; // s là điểm đầu tiên, nên trước nó không có đỉnh nào

    for (int i = 0; i < V - 1; ++i) {
        int u = minDistance(d, unvisited); // Tìm đỉnh chưa thăm mà có khoảng cách từ s đến ngắn nhất.

        if (d[u] == INT_MAX) break;        // Dừng lại nếu không có đường đi

        unvisited[u] = false;              // Đánh dấu đã thăm

        for (int v = 0; v < V; ++v) {      // Kiểm tra tất cả đỉnh lân cận với u, cập nhật lại khoảng
            if (unvisited[v] == true && graph[u][v] != INT_MAX && d[u] + graph[u][v]  < d[v]) {
                d[v] = d[u] + graph[u][v];
                pre[v] = u;   // lưu vết đường đi.
            }
        }
    }

    printSoulution(d);
}

int minDistance(int dist[], int unvisited[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; ++v) {
        if (unvisited[v] == true && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }
    return min_index;
}

void printSolution(int dist[], int pre[])
{
    for (int i = 0; i < V; ++i) {
        printf("Distance from source to %d is %d", i, dist[i]);
    }
}
```

Như vậy độ phức tạp của thuật toán là O(V^2).

Chúng ta có thể cải thiện thuật toán thành O(ElogV) với cách sử dụng binary heap (priority queue).

3. Summary

Thuật toán tham lam là bất kỳ thuật toán nào mà áp dụng kinh nghiệm giải quyết vấn đề để đưa ra tối ưu cục bộ.

Việc sử dụng tham lam sẽ giúp giải quyết vấn đề nhanh hơn với một kết quả gần đúng.

Để sử dụng tốt thuật toán tham lam, đòi hỏi người lập trình có kiến thức toán học tốt cùng với sự rèn luyện.

4. Discussion questions

1. Thuật toán tham lam có những phần nào?
2. Hãy chỉ ra năm phần của thuật toán Kruskal?
3. Hãy chỉ ra năm phần của thuật toán Prim?
4. Hãy chỉ ra năm phần của thuật toán Dijsktra?
5. Khi nào bạn sẽ áp dụng tham lam để giải quyết bài toán?

No labels