Thực hành Buổi 5 (Phần 1)

Cây khung có trọng số nhỏ nhất

Mục đích:

- Cài đặt giải thuật Kruskal
- Cài đặt giải thuật Prim

1. Cây khung có trọng số nhỏ nhất

Cây

Cây là đồ thị vô hướng liên thông và không chứa chu trình. Có 6 tính chất liên quan đến cây (xem phần lý thuyết).

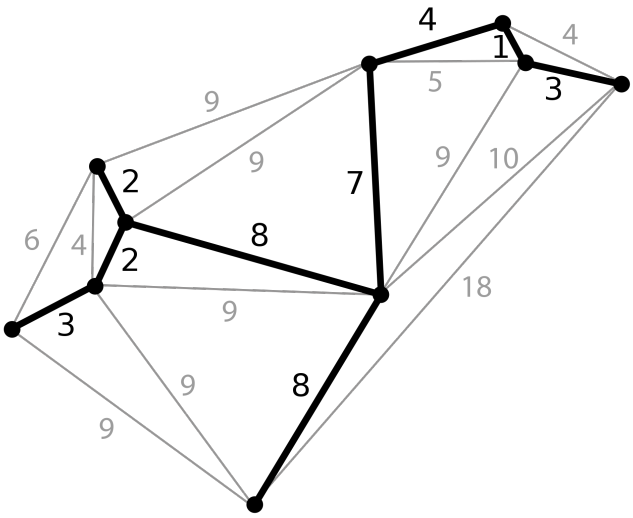
Cây khung

Cây khung (spanning tree) T của một đồ thị (liên thông) G là cây bao gồm tất cả các đỉnh của đồ thị G và một số cung của đồ thị G.

Cây khung có trọng số nhỏ nhất

Cho đồ thị vô hướng G = <V, E>. Mỗi cung của G được gán một trọng số (thường là không âm). Bài toán tìm cây khung có trọng số nhỏ nhất (còn có tên gọi là Cây phủ tối tiểu, cây bao trùm tối tiểu, cây khung tối tiểu. Tên tiếng anh là: Minimum Spanning Tree) là bài toán tìm cây khung của đồ thị G sao cho tổng trọng số các cung trên cây nhỏ nhất.

Ví dụ bên dưới minh hoạ đồ thị vô hướng liên thông và cây khung có trọng số nhỏ nhất (các cung của cây được in đậm) của nó.



2. Giải thuật Kruskal tìm cây khung có trọng số nhỏ nhất

Cho đồ thị vô hướng và liên thông G, ta cần tìm cây khung có trọng số nhỏ nhất của đồ thị này. Kết quả trả về là một cây T (cũng là một đồ thị).

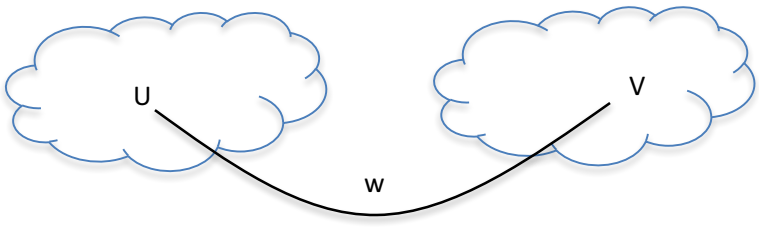
Ý tưởng:

- Sắp xếp các cung của G theo thứ tự trọng số tăng dần
- Cây T không chứa cung nào
- for (các cung e = (u, v; w) theo thứ tự đã sắp xếp)
    - Nếu thêm cung e vào T mà không tạo nên chu trình thì thêm e vào T
    - Ngược lại bỏ qua cung e
- Ta có thể cho giải thuật dừng sớm bằng cách mỗi khi thêm một cung e vào T ta tăng số cung của T lên. Giải thuật sẽ dừng khi T chứa n – 1 cung với n là số đỉnh của G.

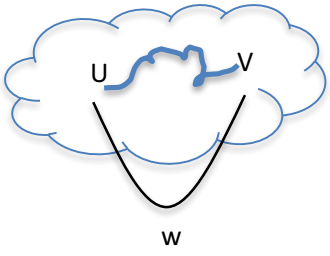
Trong giải thuật này phần khó nhất là kiểm tra việc thêm 1 cung vào cây T (đồ thị) có tạo nên chu trình hay không.

Để kiểm tra khi thêm 1 cung vào một đồ thị có tạo nên chu trình hay không ta có thể làm đơn giản là thêm cung này vào đồ thị, sau đó áp dụng giải thuật kiểm tra chu trình (trong bài trước) để kiểm tra. Tuy nhiên cách này kém hiệu quả vì độ phức tạp cao. Ta xét một phương pháp khác để kiểm tra việc tạo chu trình như sau:
Tại mỗi thời điểm, ta quản lý các bộ phận liên thông của đồ thị T.

- Nếu cung e = (u, v; w) có u và v thuộc về hai bộ phận liên thông khác nhau thì việc thêm e vào T sẽ không tạo chu trình vì chỉ có một đường đi duy nhất từ u đến v thông qua cung e.
- Khi thêm cung e có u và v ở hai bộ phân liên thông khác nhau vào cây T, sẽ gom hai bộ phận liên thông này thành 1 bộ phân liên thông mới.



- Ngược lại, nếu cung e có đỉnh u và đỉnh về cùng thuộc về một bộ phận liên thông thì khi thêm e vào T, sẽ tồn tại ít nhất 2 đường đi khác nhau từ u đến v (một theo bộ phân liên thông và một them cung e) hay sẽ tồn tại chu trình.



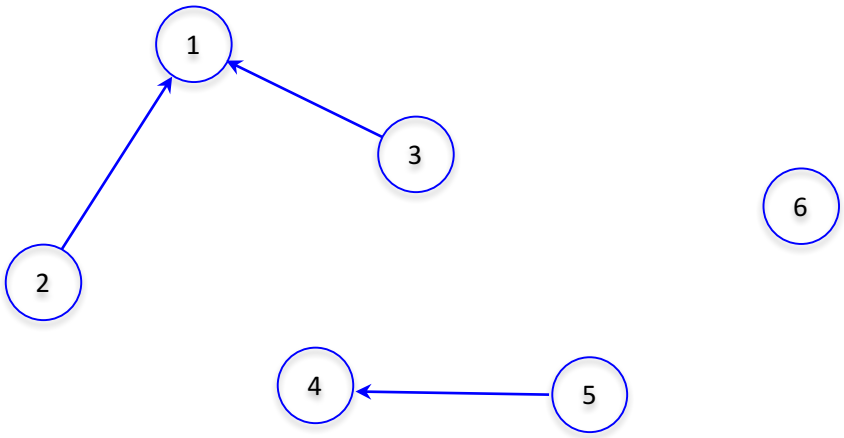
Quản lý các bộ phận liên thông (BPLT) của một cây (hoặc đồ thị): mỗi bộ phận liên thông được biểu diễn dưới dạng 1 cây (xem cấu trúc dữ liệu cây). Để đơn giản ta chỉ cần sử dụng phương pháp biểu diễn bằng mảng (lưu nút cha của các nút).

- parent[u]: đỉnh cha của đỉnh u.
- Quy ước: đỉnh ứng với gốc của cây có cha là chính nó.

Ví dụ: Đồ thị bên dưới có 3 bộ phận liên thông, mảng parent tương ứng của nó sẽ là:

u	1	2	3	4	5	6
parent[u]	1	1	1	4	4	6

3 nút gốc là 1, 4 và 6.



Như thế mỗi BPLT là một cây. Đỉnh (nút) gốc của cây là đại diện của BPLT tương ứng. Để tìm xem đỉnh u thuộc về BPLT nào ta chỉ cần tìm gốc của cây chứa u.

```
int findRoot(int u) {
  if (parent[u] = u)
    return u;
  return findRoot(parent[u]);
}
```

Cài đặt giải thuật Kruskal

Các biến hỗ trợ

- parent[u]: đỉnh cha của u.
- G: đồ thị đầu vào
- T: Cây kết quả

Giải thuật:

Sử dụng phương pháp **danh sách cung** để biểu diễn đồ thị

```
int Kruskal(Graph* G, Graph* T) {
  //Sắp xếp các cung của G theo thứ tự trọng số tăng dần
  …
  // Khởi tạo T rỗng.
  init_graph(T, G->n);
  for (u = 1; u <= G->n; u++)
    parent[u] = u; //Mỗi đỉnh u là một bộ phận liên thông
  int sum_w = 0;
  //Duyệt qua các cung của G (đã sắp xếp)
  for (e = 1; e <= G->m; e++) {
    int u = G->edge[e].u;
    int v = G->edge[e].v;
    int w = G->edge[e].w;
    int root_u = findRoot(u);
    int root_v = findRoot(v);
    if (root_u != root_v) {
      add_edge(T, u, v, w);
      //Gộp 2 BPLT root_u và root_v lại
      parent[root_v] = root_u;
      sum_w += w;
    }
  }
  return sum_w;
}
```

Để sử dụng giải thuật Kruskal, sau khi xây dựng đồ thị ta gọi Kruskal(&G, &T). Cây khung nhỏ nhất sẽ được lưu vào cây T.

Ta có thể liệt kê các cung (cùng với trọng số) của cây T.

```
int main() {
  Graph G, T;
  int n, m, u, v, w, e;
  //Đoạn này đọc dữ liệu từ bàn phím.
  //Bạn có thể sửa lại để đọc dữ liệu từ file
  scanf("%d%d", &n, &m);
  init_graph(&G, n);
  for (e = 1; e <= m; e++) {
    scanf("%d%d%d", &u, &v, &w);
    add_edge(&G, u, v, w);
  }

  int sum_w = Kruskal(&G, &T);
  printf("Tổng trọng số của cây T là: %d\n", sum_w);

  for (e = 0; e < T.m; e++)
    printf("(%d, %d): %d\n", T.edges[e].u, T.edges[e].u,
                                                T.edges[e].w);
  return 0;
}
```

Bài tập 1. Viết chương trình đọc đồ thị từ file, áp dụng giải thuật Kruskal tìm cây khung có trọng số nhỏ nhất và in kết quả ra màn hình (như chương trình mẫu).

3. Giải thuật Prim tìm cây khung có trọng số nhỏ nhất

Ý tưởng:

- Chọn 1 đỉnh bất kỳ, đưa nó vào danh sách S
- Lặp (n – 1 lần) làm các công việc sau
    - Tìm đỉnh u không có trong S và **gần với S nhất**. Gọi v là đỉnh trong S mà u gần với v nhất.
    - Đưa u vào S
    - Đưa cung tương ứng (u, v) vào T

Cài đặt giải thuật

- Sử dụng phương pháp ma trận trọng số để biểu diễn đồ thị

Mấu chốt của giải thuật Prim là việc **tìm đỉnh u không thuộc S gần với S nhất**.

Phiên bản 1.

Phiên bản cài đặt này bám sát vào ý tưởng của giải thuật Prim: thiết kế một hàm tính khoảng cách từ u đến S trong đồ thị G đang xét. Hàm này sẽ trả về đỉnh v thuộc S gần với u nhất. Nếu không có cung nối từ u đến bất cứ đỉnh nào trong G ta trả về -1.

```
int distanceFrom(int u, List* S, Graph* G) {
  int min_dist = INF;
  int min_v = - 1;
  int i;
  for (i = 1; i <= S->size; i++) {
    int v = element_at(S, i);
    if (G->A[u][v] != NO_EDGE && min_dist > G->A[u][v]) {
      min_dist = G->A[u][v];
      min_v = v;
    }
  }
  return min_v;
}
```

Các biến hỗ trợ

- S: danh sách các đỉnh đã chọn
- mark[u]: đánh dấu u đã có trong S.

Giải thuật Prim chi tiết

```
int Prim(Graph* G, Graph* T) {
  init_graph(T, G->n); //khởi tạo cây T rỗng
  List S;
  make_null_list(&S);
  int i, u;
  for (i = 1; i <= G->n; i++)
    mark[i] = 0;
  push_back(&S, 1);   //Chọn đỉnh 1 và đưa vào danh sách
  mark[1] = 1;

  int sum_w = 0; //Tổng trọng số của cây T
  for (i = 1; i < G->n; i++) {
    //1. Tìm u gần với S nhất => min_u, min_v, min_dist
    int min_dist = INF, min_u, min_v;
    for (u = 1; u <= G->n; u++)
      if (mark[u] == 0) {
        int v = distanceFrom(u, &S, G);
        if (v != -1 && G->A[u][v] < min_dist) {
          min_dist = G->A[u][v];
          min_u = u;
          min_v = v;
        }
      }
    //2. Đưa min_u vào S
    push_back(&S, min_u);
    mark[min_u] = 1;
    //3. Đưa cung (min_u, min_v, min_dist) vào T
    add_edge(T, min_u, min_v, min_dist);
    sum_w += min_dist;
  }
  return sum_w;
}
```

Bài tập 2. Tương tự bài tập 1 nhưng thay giải thuật Kruskal bằng giải thuật Prim.

Phiên bản 2: phiên bản này cài đặt giải thuật Prim tương tự như giải thuật Dijkstra. Thay vì phải tính lại khoảng cách từ 1 đỉnh u đến S trong mỗi lần lặp, ở mỗi đỉnh ta cần lưu **pi[u]** là **khoảng cách từ u đến S** và **p[u]** là **đỉnh trong S gần với u nhất**. Trong mỗi lần lặp, ta sẽ chọn đỉnh u chưa đánh dấu có pi[u] nhỏ nhất và cập nhật lại pi[.] và p[.] của các đỉnh đề với u.

```
int pi[MAXN]; //có thể cho MAXN = 500
int p[MAXN];

int Prim2(Graph* G, Graph* T) {
  init_graph(T, G->n); //khởi tạo cây T rỗng

  int i, u, v;
  for (u = 1; u <= G->n; u++)
    pi[u] = INF; //gán pi vô cùng, ví dụ: 999999

  pi[1] = 0;

  for (v = 1; v <= G->n; v++)
    if (G->A[1][v] != NO_EDGE) {
      pi[v] = G->A[1][v]; //gán pi[v] = trọng số cung (1,v)
      p[v] = 1;       //đỉnh trong S gần với v là đỉnh 1
    }

  for (i = 1; i <= G->n; i++)
    mark[i] = 0;
  mark[1] = 1; //Chọn đỉnh 1 và đánh dấu nó

  int sum_w = 0; //Tổng trọng số của cây T
  for (i = 1; i < G->n; i++) { //lặp n - 1 lần
    //1. Tìm u gần với S nhất (tìm u có pi[u] nhỏ nhất)
    int min_dist = INF, min_u;
    for (u = 1; u <= G->n; u++)
      if (mark[u] == 0) {
        if (min_dist > pi[u]) {
          min_dist = pi[u];
          min_u = u;
        }
      }
    u = min_u; //đỉnh u có pi[u] nhỏ nhất
    //2. Đánh dấu min_u
    mark[min_u] = 1;

    //3. Đưa cung (min_u, p[min_u], min_dist) vào T
    add_edge(T, min_u, p[min_u], min_dist);
    sum_w += min_dist;
    //4. Cập nhật lại pi và p của các đỉnh kề với u
    for (v = 1; v <= G->n; v++)
      if (G->A[u][v] != NO_EDGE && mark[v] == 0)
        if (pi[v] > G->A[u][v]) {
          pi[v] = G->A[u][v];
          p[v]  = u;
        }
  }
  return sum_w;
}
```