BÀI TẬP CÂY KHUNG CÓ TRỌNG LƯỢNG NHỎ NHẤT

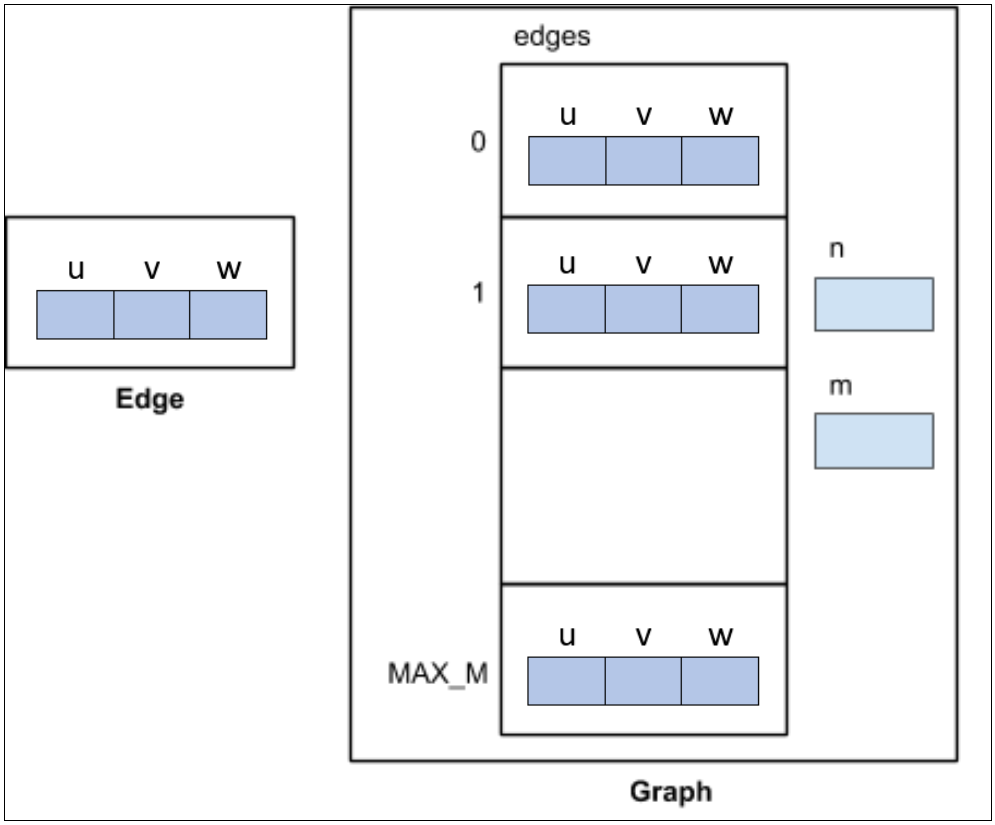
Quy ước chung

- Đánh số các đỉnh từ 1, 2, …, n
- Đỉnh: chỉ cần số lượng đỉnh: **n**
- Cung:
    - Số lượng cung: **m**
    - Chi tiết cung: **tuỳ phương pháp biểu diễn**

1.1 Biểu diễn bằng danh sách cung – trọng số

- **Ý tưởng:**
    - Mỗi cung lưu 2 đỉnh đầu mút (endpoint) của nó
    - Lưu tất cả các cung của đồ thị vào một danh sách (*danh sách đặc* hoặc *danh sách liên kết*)

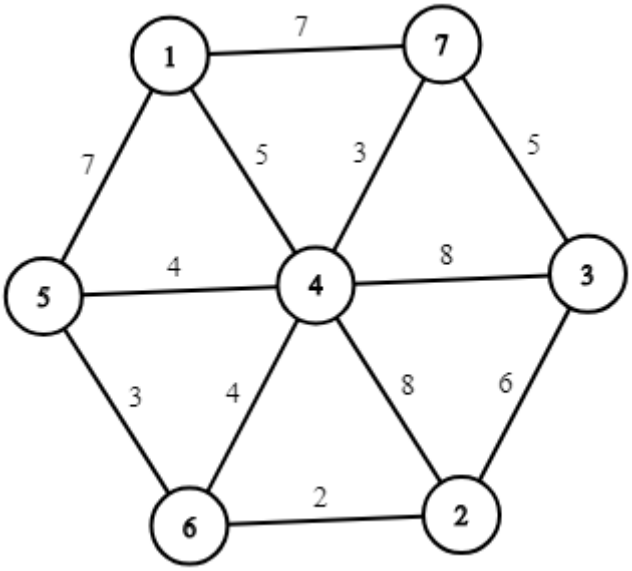
- **Sơ đồ tổ chức dữ liệu (dùng mảng để cài danh sách):**



- **Cài đặt:**

```
#define MAX_M 500
typedef struct {
        int u, v;
        int w; //Trọng lượng của cung
} Edge;
typedef struct {
        Edge edges[MAX_M];
        int n, m;
} Graph;
```

Câu 1: Cho đồ thị như hình vẽ sau:



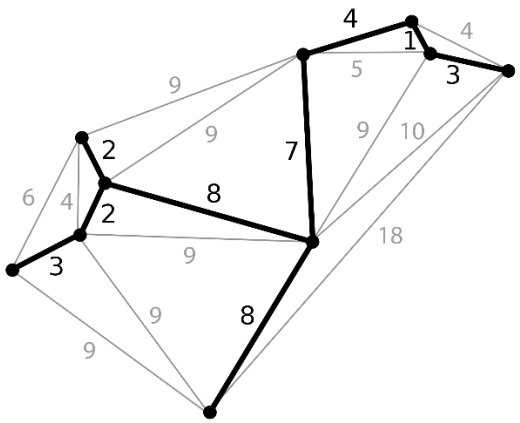
Hãy vẽ sơ đồ tổ chức dữ liệu cho đồ thị trên sử dụng phương pháp danh sách cung – trọng số.

1.2 Giải thuật Kruskal tìm cây khung có trọng số nhỏ nhất

Cây khung có trọng số nhỏ nhất (Minimum Spanning Tree – MST)

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.



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:

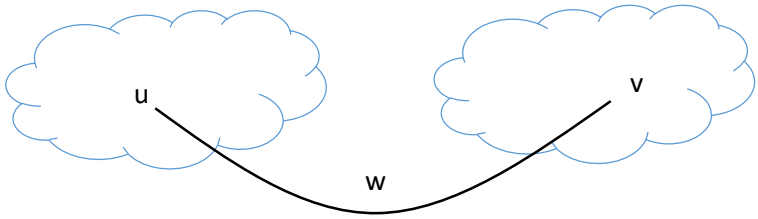
- Sắp xếp các cung của G theo thứ tự trọng số tăng dần
- Khởi tạo cây T gồm các đỉnh của G và không chứa cung nào
- **for** (các cung e = (u, v; w) theo thứ tự đã sắp xếp)
    - Nếu thêm cung e vào T mà không tạo nên chu trình thì thêm e vào T
    - Ngược lại bỏ qua cung e

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.

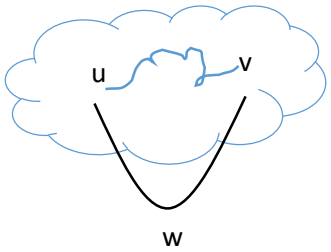
Trong giải thuật này phần khó nhất là *kiểm tra việc thêm 1 cung vào cây T (đồ thị) có tạo nên chu trình hay không*.

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 hiệu quả hơn 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 = (u, v; w) (có u và v ở hai bộ phân liên thông khác nhau) vào cây T, Hai bộ phân liên thông của u và của v sẽ được *gom lại thành 1 bộ phân liên thông mới*.



- Ngược lại, nếu cung e có đỉnh u và đỉnh v *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 đồ thị: mỗi bộ phận liên thông được biểu diễn dưới dạng 1 cấu trúc dữ liệu cây (xem cấu trúc dữ liệu cây trong học phần Cấu trúc dữ liệu). Để đơn giản ta chỉ cần sử dụng phương pháp biểu diễn cây bằng mảng (lưu nút cha của các nút).

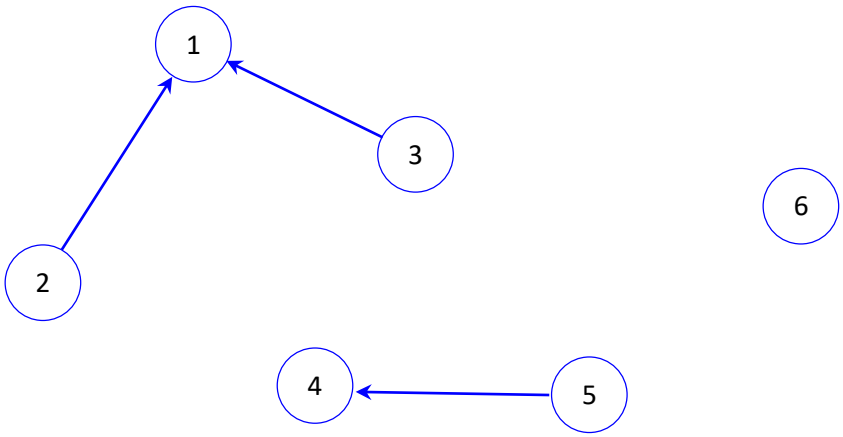
Sử dụng mảng parent để lưu đỉnh cha của các đỉnh.

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

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 đỉnh 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) {
  while (parent[u] != u)
    u = parent[u];
  return u;
}
```

Cài đặt giải thuật Kruskal

Các biến hỗ trợ

- parent[u]: đỉnh cha của u.
- pG: đồ thị đầu vào
- T: Cây kết quả (có kiểu Graph như pG)

Giải thuật:

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

```
int Kruskal(Graph* pG, Graph* T) {
  //Sắp xếp các cung của G theo thứ tự trọng số tăng dần
  …
  // Khởi tạo T rỗng.
  init_graph(T, pG->n);
  for (u = 1; u <= pG->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 <= pG->m; e++) {
    int u = pG->edge[e].u;
    int v = pG->edge[e].v;
    int w = pG->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;
}
```

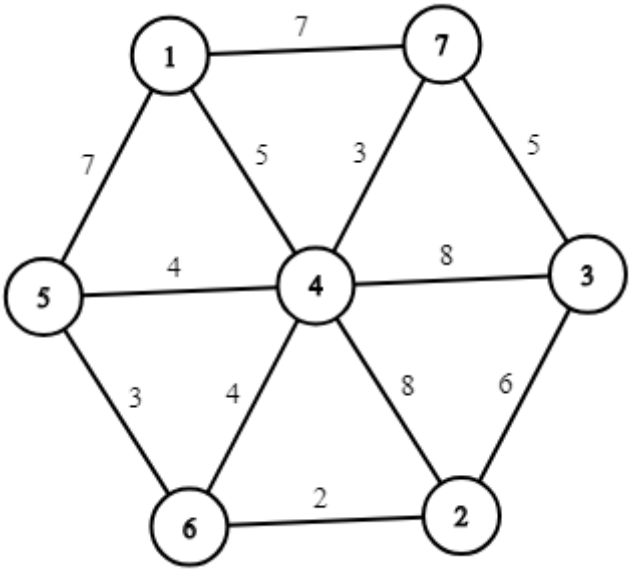
Để sử dụng giải thuật Kruskal, sau khi xây dựng đồ thị ta gọi Kruskal(&G, &T). Cây khung nhỏ nhất sẽ được lưu vào cây T.

Bài tập:

Câu 2: Viết thêm đoạn chương trình để sắp xếp các cung của G theo thứ tự trọng số tăng dần

Câu 3: Khi duyệt qua các cung e đã được sắp xếp. Nếu cung e được thêm vào cây T mà không tạo nên chu trình thì thêm e vào T. Vậy làm thế nào để kiểm tra cung e thêm vào cây T có tạo thành chu trình hay không? Giải thích.

Câu 4: Hãy chạy thủ công giải thuật Kruskal để tìm cây khung có trọng lượng nhỏ nhất cho đồ thị sau đây:



Câu 5: Hãy cho biết trọng lượng nhỏ nhất của cây tìm được bằng bao nhiêu? (sum_w)

Câu 6: Hãy cho biết số lượng cung của đồ thị cây khung tìm được?

1.3 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;
}
```

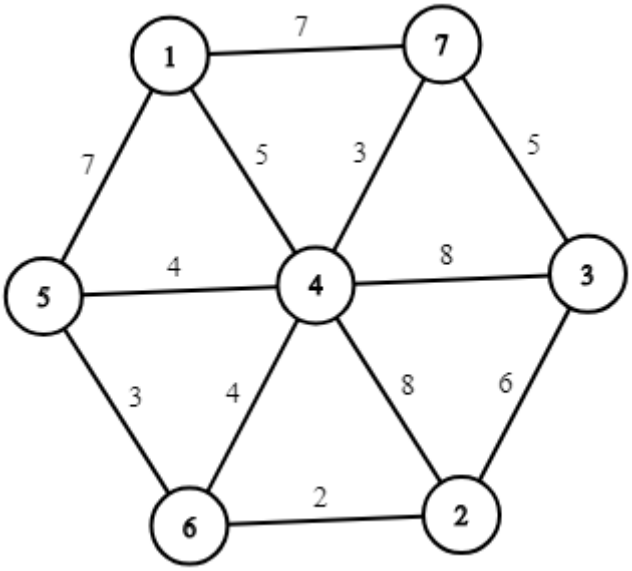
<u>Phiên bản 2:</u>

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[v] và p[v] của các đỉnh kề v của u.

```
int pi[MAXN]; //có thể cho MAXN = 500
int p[MAXN];
int Prim2(Graph* pG, Graph* T) {
  init_graph(T, G->n); //khởi tạo cây T rỗng
  int i, u, v;
  for (u = 1; u <= pG->n; u++)
    pi[u] = INF; //gán pi vô cùng, ví dụ: 999999
  pi[1] = 0;
  for (v = 1; v <= pG->n; v++)
    if (G->A[1][v] != NO_EDGE) {
      pi[v] = pG->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 < pG->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 <= pG->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 (u, p[u], min_dist) vào T
    add_edge(T, u, p[u], min_dist);
    sum_w += min_dist;
    //4. Cập nhật lại pi và p của các đỉnh kề v của u
    for (v = 1; v <= pG->n; v++)
      if (pG->A[u][v] != NO_EDGE && mark[v] == 0)
        if (pi[v] > pG->A[u][v]) {
          pi[v] = pG->A[u][v];
          p[v]  = u;
        }
  }
  return sum_w;
}
```

Bài tập:

Câu 7: Hãy chạy thủ công giải thuật Prim (Phiên bản 1 hoặc Phiên bản 2) để tìm cây khung có trọng lượng nhỏ nhất cho đồ thị sau đây:



Câu 8: Hãy cho biết trọng lượng nhỏ nhất của cây tìm được bằng bao nhiêu? (sum_w)

Câu 9: So sánh kết quả cây khung khi chạy thuật toán Prim với kết quả cây khung khi chạy thuật toán Kruskal.