SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

POLYTECH SORBONNE

Projet Algorithme avancé: sujet 1

EISE 4

Qianhui JIN

Encadrant: M.Ouzia

2020 - 2021

Sommaire

I. Description du sujet

Le sujet consiste à construire un algorithme qui trouve tous les arbres couvrants de G satisfaisant la valuation Δ et ces arbres doivent aussi être de poids minimum à partir du graphe G = <V, E, π, Δ>. En effet, ce graphe G est un graphe non orienté et connexe. Par définition, un graphe est connexe si pour tout couple de sommets, il existe au moins une chaîne permettant de les relier, c'est-à-dire que l'ensemble de sommets V et l'ensemble des arêtes E forme un graphe complet qui n'est pas séparé. En plus, dans ce graphe, π est une fonction qui donne la valuation pour tous les arêtes, c'est-à-dire que π signifie le poids de tous les arêtes, et les sommets sont valués par la fonction Δ. Par définition, un arbre couvrant de G est un sous-graphe partiel de G, c'est-à-dire que l'ensemble des sommets de l'arbre couvrant est V. Et le poids minimum signifie que la somme des poids de ses arêtes dans l'arbre couvrant doit être le plus petit. Dans notre cas, ces arbres couvrants de poids minimum doit aussi satisfaire une limite de valuation, d(v) ≤ Δ(v), cela signifie que le degré d'un sommet doit être inférieur ou égale à la valuation Δ de cette sommet. Par définition, le degré d'un sommet est égal au nombre d'arêtes incidentes à ce sommet.

Pour être plus claire, on peut prendre un exemple de graphe G dont toutes les arêtes sont de poids 1 comme la figure 1, et si on donne une valuation Δ = (1,3,1,2,2), alors on peut dire que l'arbre couvrant dans le figure 2 n'est pas satisfait la limite de valuation, car pour le sommet 3, il a une limite de valuation qui égale 1, mais dans le figure 2, on peut voir que le nombre d'arêtes incidentes à sommet 3 est 2. Donc il ne satisfait pas la limite de valuation. En revanche, l'arbre couvrant dans le figure 3 satisfait bien la limitation, le nombre d'arêtes incidentes à tous les sommets est inférieur ou égale à la valuation Δ du graphe G. En plus, vu que le poids de chaque arête est pareil, donc le poids minimum de ce graphe G égale (nombre de sommets - 1) = 4, l'arbre A2 est bien un arbre couvrant qui est de poids minimum et satisfait aussi la valuation.

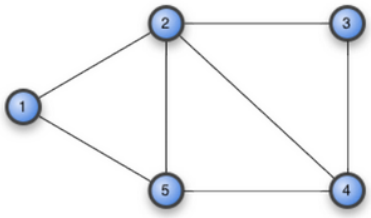

FIGURE 1 – Graphe G

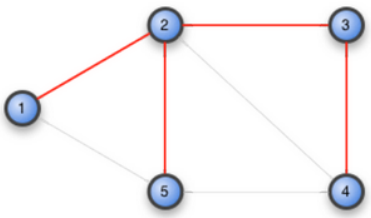

FIGURE 2 – Arbre $\mathcal{A}_1$

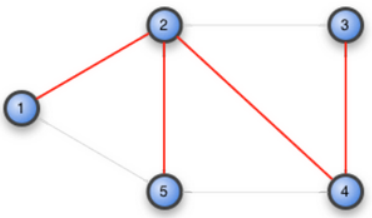

FIGURE 3 – Arbre $\mathcal{A}_2$

II. Analyse du travail

Comme on a présenté dans la première partie, le but qu'on veut atteindre est de trouver tous les arbres couvrants qui satisfont la limite de valuation et qui sont de poids minimum à partir d'un graphe non orienté et connexe. Or un arbre est un graphe connexe et sans cycle, donc un arbre couvrant doit être aussi connexe et sans cycle.

En conséquence, pour trouver tous les arbres couvrants ce qui satisfait la demande, on doit d'abord trouver un arbre de couvrant qui satisfait la limitation, c'est une version naïve. Puis à partir de cet arbre couvrant, on fait de l'optimisation pour trouver les autres arbres couvrants qui satisfont à la fois la limitation et le poids minimum. Alors pendant l'optimisation on modifie la structure de l'arbre couvrant de version naïve pour obtenir les autres arbres couvrants, c'est-à-dire qu'on ajoute ou supprime un arête dans l'arbre formé. Et en même temps, on doit aussi tester si ces arbres couvrants modifiés ont bien satisfait sa limite de valuation, comparer ces arbres couvrants obtenues sont bien de poids minimum et vérifier si ces arbres couvrant n'ont pas formé un cycle. Pour réaliser ces modifications, on doit connaître toutes les arêtes dans le graphe, savoir comment ils sont reliés et combien de poids ils portent. Ensuite à partir de ces informations, on peut déterminer quelle arête doit être supprimer selon la situation. Et à la fin, on enregistre tous les arbres ce qui satisfait les besoins.

III. Les structures de données

Alors pour atteindre notre but, on a défini les structures des données comme la figure 4. En entrée, on doit avoir le nombre de sommets V, le nombre d'arêtes E, un matrice d'adjacent qui représente le graphe G = <V, E, π, Δ> et un tableau de limitation qui donne la limite de valuation du chaque sommet. En sortie, on utilise une liste pour stocker tous les arbres de poids minimum qui satisfont la limitation.

Entrée: G = <V, E, π, Δ> : un graphe non orienté valué connexe (matrice)
V : le nombre de sommets
E : le nombre d'arête
k[V] : tableau de la limitation pour chaque sommet

Sortie: vector<arbre> : liste de tous les arbres couvrant de poids minimum respectant la limitation

figure 4: Les structures des données

En plus, on a défini aussi 3 classes comme le figure 5, ce sont la classe de l'arête (Edge) qui donne la source, la destination et le poids de l'arête, la classe arbre et la classe graphe. Pour les deux dernière classes, ils contiennent tous un tableau de sommets, la matrice d'adjacence et un map qui stocke les arêtes. Et leurs différence est que la classe arbre a un tableau qui stock la limitation déjà utilisé pour chaque sommet, le nombre de poids totale, un table qui stocke la valeur maximum des arêtes choisi, un tableau stocke ces arêtes choisi et une arête qui lie le sommet indiqué et la classe graphe contient le nombre de sommets et d'arête, tableau de parent pour chaque sommet, tableau de limitation, la valeur minimum du poids, liste de sous ensemble de graph, liste des arêtes qui ne peut plus utilisé et une liste des arbres.

```
// classe de arete
class Edge {
public:
    int src, dest, poids; //source,destination et poids pour chaque arete
```

```
// classe de l'arbre
class Arbre {
public:
    int sommets[N]; // tableau du sommet
    int g[N][N];    // matrice d'adjacent
    //vector <class Edge> edges; //liste d'arête
    map<pair<int,int>,class edges> mapE; // map: source et destinataire comme key et edges comme la valeur
    int m[N];   // tableau du limite déjà utilisé
    Edge dp1[N];    //tableau pour stocker les arêtes de poids maximum
    Edge dp2;   //l'arête qui lie le point indiqué
    int poids;
```

```
// classe pour le graphique
class Graph {
public:
  int V,E;    //nombre de sommets et d'aretes
  int sommets[N];
  //vector <class Edge> edges;
  map<pair<int,int>,class edges> mapE;
  int g[N][N];
  int father[N]; // parent pour les sommets
  int limite[N]; //limite de degre pour chaque sommet
  int mini; // le poids minimum
  vector<class Graph> sousG; //les sous-ensembles de graphe
  vector<Edge> nonEdges; // liste des arête qui ne peut plus utilisé
  vector<class Arbre> Tmini; //les arbres couvrants de poids minimum
```

figure 5: Les différentes classes

IV. Pseudo-code

```
Début

// initialisation
Graphe ← V, E, k[V], g[V][V]

//creatEdgesG(Graphe)
//créer un liste d'arête à partir la matrice d'adjacence du graphe G
//retourner le nombre totale d'arête
Pour (i allant de 0 à V-1) faire
        Pour (j allant de i+1 à V-1) faire
            Si (g[i] [j] ≠ NULL) alors
                p(src) ← i
                p(dest) ← j
                p(poids) ← g[i][j]
                edges ← edges + p // liste d'arête du graphe
            Fin Si
        Fin Pour
Fin Pour

// trouver(Graphe)
// trouver l'arbre couvrant de version naïve en testant s'il est satisfait la limitation
Pour (iterE allant de 0 à strlen(edges)) faire
        // vérifier si la source et destinataire existe ou leur limite est la même que le tableau de limite utilisé
        Si ((src && dest) || (m[src] == k[src] || m[dest] == k[dest])) alors
            nonEdges ← nonEdges + iterE  // tableau des arêtes qui ne peuvent plus utilisé
        Fin Si
        PlusEdge(arbre, iterE) // ajouter l'arête iterE dans l'arbre
Fin Pour

 // optimizer(Graphe)
 //optimiser l'arbre couvrant pour trouver les autres arbres couvrants de poids minimum
 Pour (iterE allant de 0 à strlen(nonEdges)) faire
        Pour (iterA allant de 0 à strlen(Tmini)) faire
            Si (m[src] < k[src] && m[dest] < k[dest]) alors
                max ← dfs1(iterA, iterE(src), iterE(dest), -1)
                Si (poids ≥ max) faire
                    Pour (itE allant de 0 à strlen(dp1[iterE(dest)])) faire
                        minusEdge(arbre, itE)      //supprimer l'arête itE de l'arbre
                        plusEdge(arbre, iterE)
                        Tmini ← Tmini + arbre // tableau des arbres couvrants
                    Fin Pour
                Fin Si
            Fin Si
            Si (m[src] == k[src] && m[dest] < k[dest]) alors
                max ← dfs2(iterA, iterE(src), iterE(src), -1, iterE(dest))
            Fin Si
            Si (m[src] < k[src] && m[dest] == k[dest]) alors
                max ← dfs2(iterA, iterE(dest), iterE(dest), -1, iterE(src))
            Fin Si
            Si (poids ≥ max) faire
                Pour (itE allant de 0 à strlen(dp1[iterE(dest)])) faire
                    minusEdge(arbre, itE)      //supprimer l'arête itE de l'arbre

                    plusEdge(arbre, iterE)
                    Tmini ← Tmini + arbre // tableau des arbres couvrants
                Fin Pour
            Fin Si
        Fin Pour
 Fin Pour

 Fin
```

figure 6: Pseudo-code

```
// dfs1(arbre, src, dest, precedent)
// Trouver des arêtes qui sont de poids maximum permis les arêtes pour allant de src à dest
Pour (i allant de 0 à N-1) faire
        Si (i ≠ precedent && g[src][i]) alors
                Si (dp[i] == -1) alors
                        Si (dp[i] > g[src][i]) alors
                                Pour (itE allant de 0 à strlen(dp1[src])) faire
                                        dp[i] ← dp[i] + itE
                                Fin Pour
                                dp[i] ← dp[src]
                        Sinon
                                p(src) ← src
                                p(dest) ← dest
                                p(poids) ← g[src][i]
                                dp[i] ← dp[i] + p
                        Fin Si
                Fin Si
                dfs1(arbre, i, dest, src)
        Fin Si
Fin Pour


/ dfs2(arbre, src, src2,  precedent, dest)
// Trouver un arête qui est de poids maximum et relié avec point src parmis les arêtes pour allant de  // src au dest

Pour (i allant de 0 à N-1) faire
        Si (i ≠ precedent && g[src][i]) alors
                Si (i == dest) alors
                        flag ← 1
                Sinon
                        dfs2(arbre, i, src2,  src, dest)
                Fin Si
                Si (src == src2 && flag) alors
                        dp2(src) ← src
                        dp2(dest) ← i
                        dp2(poids) ← g[src][i]
                Fin Si
        Fin Si
Fin Pour
```

figure 7: Pseudo-code Bis

On a fait un pseudo-code qui est représenté dans les figures au-dessus. Au début, on a configuré les paramètres du graphe G. Puis on a utilisé la fonction *creatEdegesG(Graphe)* pour trouver la liste d'arête(edges) et le nombre totale d'arête d'une graphe en examinant son matrice d'adjacence. Ensuite, on utilise la fonction *trouver(Graphe)* qui permet de trouver un arbre couvrant qui satisfait la limitation. Dans cette fonction, on parcourt la liste d'arête du graphe et tester si la source et destinataire existe ou leur limite a atteint le maximum, sioui on l'ajoute dans le tableau des arêtes qui ne peuvent plus utilisé (nonEdges), à la fin on ajoute l'arête choisi dans l'arbre et ajoute l'arbre formé dans la liste de l'arbre (Tmini). Après cela on utilise la fonction *optimizer(Graphe)* qui sert à optimiser l'arbre couvrant obtenue dans la fonction *trouver(Graphe)* en changeant la structure de l'arbre afin de trouver tous les arbres couvrant de poids minimum qui satisfont la limitation. Dans cette fonction, on parcourt la liste des arêtes (nonEdges) et la liste des arbres (Tmini) et on répartie en 2 parties. La première partie est que si la limite de la source et la destinataire n'est pas encore atteint alors on utilise la fonction *dfs1(arbre, src, dest, precedent)* pour trouver les arêtes qui portent de poids maximum parmis les arêtes allant de la source vers la destinataire. La deuxième

partie est pour que si la limite de la source ou la destinataire a atteint et l'autre n'est pas encore atteint, on utilise la fonction *dfs2(arbre, src, src2, precedent, dest)* pour trouver un arête de poids maximum qui lie avec la source parmis les arêtes allant de la source vers la destinataire. Ensuite on utilise ces informations obtenus afin de supprimer l'arête qui est de poid maximum et ajouter un autre arête qui n'est pas encore présent, pour faire l'optimisation et trouver les autres arbres couvrants qui satisfait la demande.

V. Analyse de complexité

Pour le calcul de la complexité, on considère qu'on est dans le pire cas. D'après le pseudo-code on peut observer que la fonction *creatEdgesG* doit parcourir au plus V^2. Et pour la fonction *trouver*, elle doit parcourir la liste des arêtes (edges) donc dans le pire cas elle doit parcourir E fois. Dans la fonction *optimizer*, il y a les fonctions *dfs1* et *dfs2*, elles doivent parcourir au plus V^2 dans ces deux fonctions. Et dans la partie principale de la fonction *optimizer*, elle a parcour [(E-V+1) * {E!/((V-1)!*(E-V+1)!)}*V] Donc la complexité totale de ce programme est la somme de ces fonctions qui égale V^2 + E + [(E-V+1) * {E!/((V-1)!*(E-V+1)!)}*V] * V^2 ≈ V^(E+3).

VI. Les fonctions les plus intéressantes

Dans notre cas, les fonctions les plus intéressantes sont la fonction *optimizer* et la fonction *dfs1* qui sont représentées par les figures en dessous. Ces 2 fonctions sont principales et importantes pour notre projet. La fonction *optimizer* est indispensable, elle nous permet de trouver les autres arbres couvrant ce qui satisfait la demande. Comme on l'a présenté dans la partie pseudo-code, dans cette fonction, on a répartie en 2 parties, les 2 parties ont un traitement similaire *(dfs1 et dfs2)*, la seule différence est leur condition: le premier partie est pour les deux sommets ont atteint leur limite, la deuxième partie est pour un des deux sommets a atteint sa limite. Elle permet aussi de parcourir tous les arbres couvrants trouvés en lui ajoutant ou supprimer les arêtes et tester le respect de la limitation.

La fonction *dfs1* est utilisée dans la première partie de la fonction *optimizer,* c'est une fonction récursive et on a implémenté l'algorithme de parcours en profondeur dans cette fonction. Elle permet de partir du point de la source vers la prochaine point qui est un point intermédiaire allant de la source vers la destinataire, et tester s'il existe des arêtes qui ont plus grand poids que l'arête allant de la source vers le point intermédiaire. Sioui, on ne fait rien. Sinon, on supprime les arêtes qui ont de poids inférieur à cette nouvelle arête.

```
// optimiser l'arbre couvrant pour trouver les autres arbres couvrants de poids minimum
void optimizer(Graph& G){
    for(auto iterE:G.nonEdges){
        tmp.clear();
        for(auto iterA:G.Tmini){
            int max = 0;    //poids d'arête à supprimer

            //tous les deux sommets sont full
            if(iterA.m[iterE.src]==G.limite[iterE.src] && iterA.m[iterE.dest]==G.limite[iterE.dest]){
                continue;
            }

            //tous les deux sommets ne sont pas full
            if(iterA.m[iterE.src]<G.limite[iterE.src] && iterA.m[iterE.dest]<G.limite[iterE.dest]){

                memset(iterA.dp,-1,N*sizeof(int));
                iterA.dp[iterE.src] = -INF;

                max = dfs1(iterA,iterE.src, iterE.dest, -1);
                if (iterE.poids < max) continue;
                if (iterE.poids == max) {
                    for (auto itE : iterA.dp1[iterE.dest]) {
                        Arbre* pa = new Arbre(iterA);
                        minusEdge(*pa, itE);
                        plusEdge(*pa, iterE);
                        tmp.push_back(*pa);
                        delete pa;
                    }
                }else{    //iterE.poids > max
                    G.Tmini.clear();
                    for (auto itE : iterA.dp1[iterE.dest]) {
                        Arbre* pa = new Arbre(iterA);
                        minusEdge(*pa, itE);
                        plusEdge(*pa, iterE);
                        tmp.push_back(*pa);
                        delete pa;
                    }
                }
                continue;
            }
```

```
            flag = 0;
            //src est full et dest n'est pas full
            if(iterA.m[iterE.src]==G.limite[iterE.src] && iterA.m[iterE.dest]<G.limite[iterE.dest]){
                max = dfs2(iterA,iterE.src, iterE.src,iterE.dest, -1);
            }
            //src est full et dest n'est pas full
            if(iterA.m[iterE.src]<G.limite[iterE.src] && iterA.m[iterE.dest]==G.limite[iterE.dest]){
                max = dfs2(iterA,iterE.dest, iterE.dest,iterE.src, -1);
            }

            if (iterE.poids > max) continue;
            if (iterE.poids == max) {
                Arbre* pa = new Arbre(iterA);
                minusEdge(*pa, iterA.dp2);
                plusEdge(*pa, iterE);
                tmp.push_back(*pa);
                delete pa;
                continue;
            }
            else{    //iterE.poids < max
                G.Tmini.clear();
                G.mini = G.mini + iterE.poids - max;
                Arbre* pa = new Arbre(iterA);
                minusEdge(*pa, iterA.dp2);
                plusEdge(*pa, iterE);
                tmp.push_back(*pa);
                delete pa;
            }
        }
        for(auto it:tmp){
            G.Tmini.push_back(it);
        }
    }
}
```

figure 8: La fonction optimizer

```
// trouver des arêtes qui sont de poids maximum parmis les arêtes pour allant de point x au point s
int dfs1(Arbre& a, int x, int s, int pre){
    int i;
    for (i = 0; i <= N; i++){
        if (i != pre && a.g[x][i]){
        if (a.dp[i] == -1){
                // comparer le poids de l'arête du point précédent du x à x et l'arête allant de x à i
                if (a.dp[x] > a.g[x][i]){
                    a.dp1[i].clear();
                    for (auto itE : a.dp1[x]) {
                        a.dp1[i].push_back(itE);
                    }
                    a.dp[i] = a.dp[x];
                }
                else if(a.dp[x] == a.g[x][i]){
                    for(auto itE: a.dp1[x]){
                    a.dp1[i].push_back(itE);
                    }
                    Edge* pe = new Edge;
                    pe->poids = a.g[x][i];
                    pe->src = x < i ? x : i;    // enregistre cet arête
                    pe->dest = x > i ? x : i;
                    a.dp1[i].push_back(*pe);
                    delete pe;
                    a.dp[i] = a.dp[x];
                }
                else {  //a.dp1[x].poids < a.g[x][i]
                    a.dp1[i].clear();
                    Edge* pe = new Edge;
                    pe->poids = a.g[x][i];
                    pe->src = x < i ? x : i;
                    pe->dest =x > i ? x : i;
                    a.dp1[i].push_back(*pe);
                    a.dp[i] = a.g[x][i];
                    delete pe;
                }
            }
            dfs1(a, i, s, x);   // récursive pour parcourir tous les sommets de x à s
        }
    }
    return a.dp[s];
}
```

figure 9: La fonction dfs1

VII. Résultats obtenus

Pour le graphe qui sont représenté dans le figure 1, on a obtenu un résultat satisfaisant après l'exécution du notre programme comme le figure 10. Ce graphe contient 6 arbres couvrant de poids minimum (4) qui satisfait bien la limitation Δ=(1,3,1,2,2). Pour être clair, on les représente en forme graphique comme le figure 11. Et dans le figure 10, on peut aussi obtenir le temps d'exécution du programme, qui égale 0,243ms, on en déduit que la performance du programme n'est pas mal. En plus, on a aussi essayé de changer le poids du graphique ou le nombre d'arête et sommet, on a obtenu les résultats correspondants. Le temps d'exécution dépend du nombre d'arbres qu'on a obtenu et de la taille du matrice qu'on a eu, si ces deux paramètres sont plus grands, alors le programme doit dépenser plus de temps pour exécuter.

```
qh@qhdeMacBook-Air Degre-limit-tree-main % ./a.out
temps:0.000243
Arbre 1: Le poids est 4
Les aretes sont:
0-1
1-2
1-3
3-4

Arbre 2: Le poids est 4
Les aretes sont:
0-4
1-2
1-3
3-4

Arbre 3: Le poids est 4
Les aretes sont:
0-1
1-2
1-4
3-4

Arbre 4: Le poids est 4
Les aretes sont:
0-4
1-2
1-3
1-4

Arbre 5: Le poids est 4
Les aretes sont:
0-1
1-4
2-3
3-4

Arbre 6: Le poids est 4
Les aretes sont:
0-4
1-3
1-4
2-3
```

Figure 10: résultat pour le graphe G exemplaire

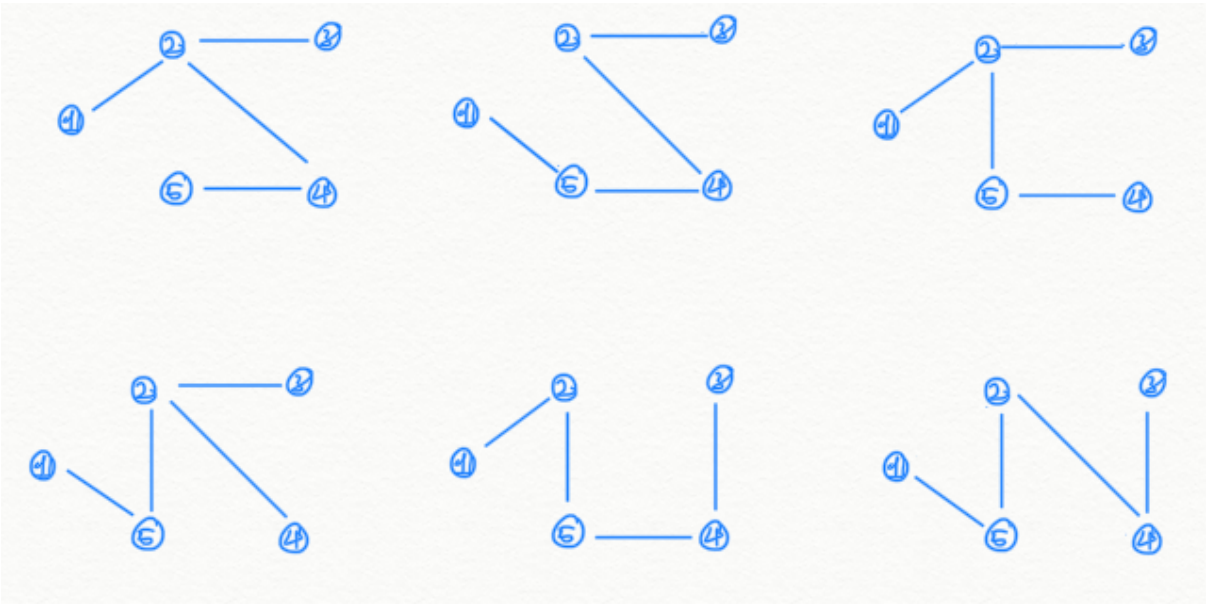

Figure 11: résultat présente en graphique

VIII. Les difficultés rencontrés

Pendant le projet, on a rencontré quelque difficultés qui sont listé suivant:

- Manque de connaissance
- Ne maîtrise pas trop bien la langage C++
- Difficile de réunir dans la situation sanitaire
- Manque de temps

IX. Bibliographie

- https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/
- https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
- https://programmersought.com/article/5558659458/