Projet d'algorithmique

Matthieu & Nicolas

Table des matières

Introduction

Ce projet nous propose de manipuler des graphes non-orientés connexes et de développer plusieurs algorithmes permettant de résoudre un problème à priori complexe : déterminer un sous-graphe de poids optimal couvrant un sous-ensemble de sommets. C'est pourquoi, nous développerons, comme explicité dans le sujet, des algorithmes sur des cas particuliers simples, pour déboucher sur un cas plus général.

1 Topologie du sous-graphe

- Pour que le chemin soit minimum, il ne faut pas passer deux fois par le même sommet. C'est-à-dire qu'il ne doit pas y avoir de circuit dans le sous-graphe.
- Pour avoir une chance d'être optimal, le sous-graphe doit avoir un poids strictement inférieur au graphe initial.
- Le graphe de départ et d'arrivée doivent obligatoirement être connexes.

2 Le cas $|V'| = 2$

2.1 Algorithme de Dijsktra

Pour trouver un tel sous-graphe, il suffit de trouver un plus court chemin du premier sommet au second. L'algorithme de **Dijsktra** permet de résoudre ce problème en un temps polynômial.

Voici comment s'écrit cet algorithme :

Nom: Dijsktra
Role: Effectue un plus court chemin entre deux sommets
Entrée: G : **Graphe**, d : **Sommet**, a : **Sommet**
Sortie: G : **Graphe**
début
    tant qu´ il existe des sommets non marqués dans G **faire**
        Choisir v tq $\pi(v)$ est minimal et le marquer.
        Pour tous voisins w de v, $\pi(v) \leftarrow \text{Min}(\pi(v), \pi(v) + \omega(v, w))$
    fait
fin

2.2 Implémentation

Pour l'implémentation de tous les algorithmes du projet, définissons un modèle de stockage des graphes[1] :

Structure sommet
    marque : **Booléen**
    suiv : ^**Voisin**
fin structure

Structure voisin
    valeur : **Entier**
    indice : **Entier**
    suiv : ^**Voisin**
fin structure

[1] Un graphe sera donc défini ainsi :
G : **Tableau**[$1 \dots N$] **de Sommets**

Voici une implémentation de cet algorithme :

```
Algorithme Dijsktra (G,d,a)
Paramètres
    (E) G : Graphe
    (E) d,a : Sommet
Variables
    pere :  Tableau[1...N] d' Entiers
    PCC : Pile
    s : Entier
    courant : ^Voisin
début
    initialisation_pere(pere, d)

    tant que( sommets_non_marque(G) ) faire
        s ← donne_sommet_non_marque_min(G, pere)
        G[s].marque ← Vrai
        courant ← G[s].suiv

        tant que( courant ≠ nil ) faire
            si( pere[courant^.indice] > courant^.valeur + pere[s] ) alors
                pere[courant^.indice] ← courant^.valeur + pere[s]
            finsi
            courant ← courant^.suiv
        fait
    fait


    // pere est rempli pour chaque sommet du poids du plus court chemin
    empile(PCC, a)
    s ← a
    tant que( s ≠ d ) faire
        courant ← G[s].suiv
        tant que( courant ≠ nil ) faire
            si( pere[s] − courant^.valeur = pere[courant^.indice] ) alors
                empile(PCC, courant^.indice)
                s ← courant^.indice
                courant ← nil
            sinon
                courant ← courant^.suiv
            finsi
        fait
    fait


    // Affichage du graphe résultant
    tant que( non pile_vide(PCC) ) faire
        écrire(depiler(PCC))
    fait
fin
```

2.3 Complexité

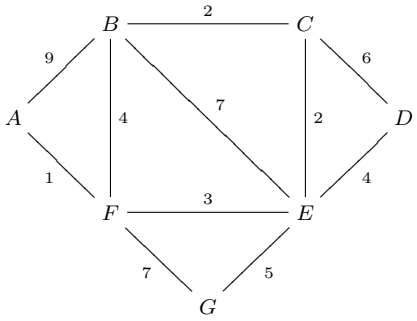
Chaque boucle **tant que** possède en son sein une autre boucle **tant que**. Chacune de ces boucles est de complexité N.

Au total :

$$\begin{aligned} \mathcal{C} &= O(N^2) + O(N^2) + O(N) \\ &= O(N^2) \end{aligned}$$

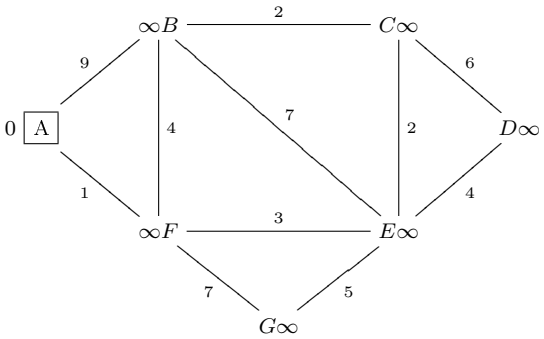
2.4 Déroulement de Dijsktra

Considérons le graphe G suivant :

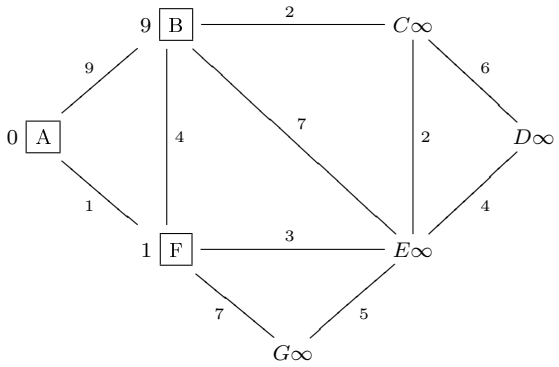


Voilà ce que donne l'exécution de **Dijsktra** avec pour sommet de départ A et d'arrivée D.

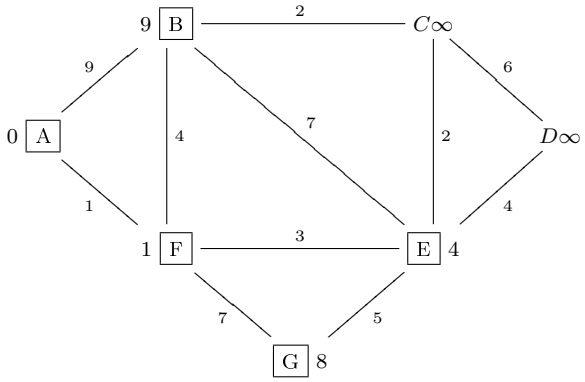
Le tableau pere est initialisé ainsi :



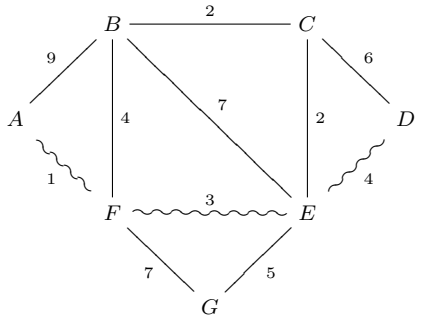
On marque les voisins de A et on met à jour le tableau pere :



On prend les voisins du sommet au coût le plus faible et on les marque.
On met aussi à jour le tableau pere :



A la fin, on obtient le plus court chemin a l'aide des coûts renseignés dans le tableau pere :



3 Le cas $|V'| = |V|$

3.1 Algorithme de Kruskall

On cherche un sous-graphe contenant tous les sommets du graphe initial. On peut associer cela à la recherche d'un arbre couvrant de coût minimal, ce que fait précisément l'algorithme de **Kruskall**.

Nom: Kruskall
Role: Cherche un arbre couvrant minimal sur un graphe
Entrée: G : **Graphe**
Sortie: G : **Graphe**
début
    Trier les **Arêtes** par ordre croissant
    Parcourir les **Arêtes** triées et les ajouter à l'arbre couvrant min si elles ne créent pas de cycle.
fin

Pour implémenter cet algorithme, on a besoin de définir une nouvelle structure :

```
Structure Arête
    depart : Entier
    arrivee : Entier
    poids : Entier
fin structure
```

3.2 Implémentation

Le tableau T permet de savoir si l'ajout d'une arête ne crée pas de cycle. Au départ, tous les sommets possèdent un numéro distinct. Par la suite, deux sommets reliés se verront assigner le même numéro et on ne pourra relier 2 sommets que si ils ont un numéro distinct. Sinon, on crée un cycle.

```
Fonction Kruskall (G : Graphe) retourne Graphe
Variables
    T : Tableau[1 ...N] d' Entiers
    L : Liste d'Arêtes
    i : Entier
    G' : Graphe
début
    L ← donne_aretes_triees(G)

    pour i ←1 à N faire
        T[i] ← i
    fait

    i ← 1
    tant que( i ≤ longueur(L) ) faire
        si( T[L[i].depart] ≠ T[L[i].arrivee] ) alors
            mettre_a_jour_T(T, L[i].depart, L[i].arrivee)
            ajoute_voisin(G', L[i].depart, L[i].arrivee, L[i].poids)
            ajoute_voisin(G', L[i].arrivee, L[i].depart, L[i].poids)
        finsi

        i ← i+1
    fait
    retourne G'
fin


Algorithme mettre_a_jour_T (T, d, a)
Paramètres
    (E/S) T : Tableau[1 ... N] de Sommets
    (E) d, a : Entiers
Variables
    val, j : Entiers
début
    j ← 1
    val ← T[a]
    tant que( j ≤ N ) faire
        si( T[j] = val ) alors
            T[j] ← T[d]
        finsi
        j ← j + 1
    fait
fin
```

3.3 Complexité

La fonction **mettre_a_jour_T** est en $O(N)$. La boucle **tant que** de l'algorithme de **Kruskall** est en $O(N)$. On a donc un algorithme en $O(N^2)$.

3.4 Déroulement de Kruskall

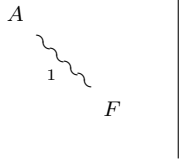
Reprenons le graphe G du §2.4 de la page 4.

On trie d'abord les arêtes dans l'ordre croissant :

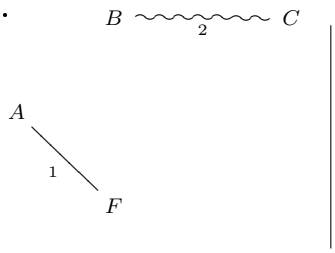
$$\rightarrow AF, BC, CE, FE, BF, ED, GE, CD, FG, BE, AB$$

On ajoute ensuite les arêtes au sous-graphe tant que cela ne produit pas de cycle.

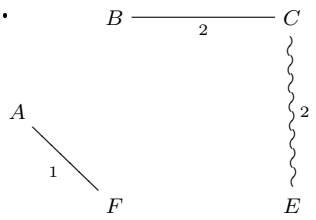
1.



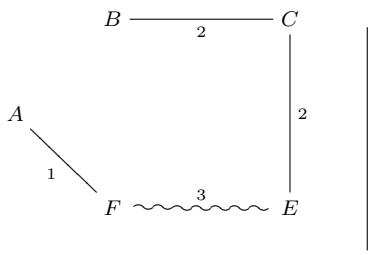
2.



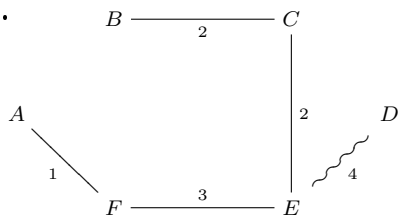
3.



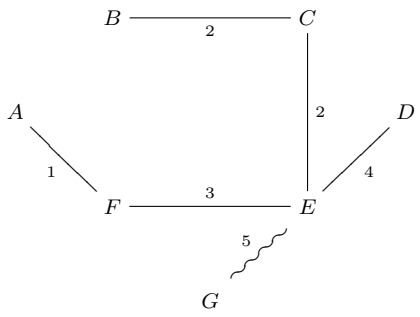
4.



5.



6.



4 Le cas $|V'| = 3$

4.1 Introduction

Ce cas revient à effectuer un algorithme de plus court chemin entre deux sommets en passant pas un autre sommet. C'est-à-dire qu'on doit obliger le passage par un sommet.

Pour cela, on peut diviser le chemin en *2 parties*. La première partie correspond au chemin reliant le premier sommet au second. La seconde partie correspond au chemin reliant le second sommet au troisième. Le problème est de savoir lequel des 3 sommets fournis est le 1^{er}, le 2^{e} et le 3^{e}. Car si on échange la position des sommets, le chemin va également voir son coût changer.

La solution est d'effectuer tous les cas possibles, puis pour chaque cas, faire 2 **Dijsktra** et enfin, de *relier* les deux sous-graphes obtenus pour n'en faire qu'un.

Soient a, b et c trois sommets quelconques d'un graphe G.
On a $A_3^3 = 6$ possibilités :

- a, b, c → **Dijsktra**(a,b) puis **Dijsktra**(b,c) $= G_1$
- a, c, b → **Dijsktra**(a,c) puis **Dijsktra**(c,b) $= G_2$
- b, a, c → **Dijsktra**(b,a) puis **Dijsktra**(a,c) $= G_3$
- b, c, a → **Dijsktra**(b,c) puis **Dijsktra**(c,a) $= G_4$
- c, a, b → **Dijsktra**(c,a) puis **Dijsktra**(a,b) $= G_5$
- c, b, a → **Dijsktra**(c,b) puis **Dijsktra**(b,a) $= G_6$

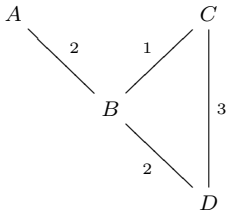
$$G_{PCC} = \min_{1 \leq i \leq 6} (\omega(G_i))$$

4.2 Implémentation

Pour l'instant, un **Graphe** est stocké grâce à un tableau[2]. Chaque sommet est identifié par l'indice du sommet dans le tableau. A partir de maintenant, cette configuration pose problème. En effet, imaginons 2 graphes identiques à 1 sommet près :

Par exemple :

Soit le graphe H :



Le graphe sera stocké ainsi :

A	→	$B,2$	→	**nil**				
B	→	$A,2$	→	$C,1$	→	$D,2$	→	**nil**
C	→	$B,1$	→	$D,3$	→	**nil**		
D	→	$B,2$	→	$C,3$	→	**nil**		

($\star$) Le même graphe sans le sommet A :

B	→	$C,1$	→	$D,2$	→	**nil**
C	→	$B,1$	→	$D,3$	→	**nil**
D	→	$B,2$	→	$C,3$	→	**nil**

Pour construire le graphe ($\star$), nous devons créer un **Tableau[$1 \ldots N-1$] de Sommets** et insérer tous les sommets sauf le sommet A. Mais avec la configuration actuelle, comment connaître, par exemple, le rang du sommet C dans ($\star$) puisque dans H son rang est 3 et que dans ($\star$), c'est 2 ?

La solution est de faire correspondre un **Tableau[$1 \ldots N$] de Caratères** permettant ainsi de *nommer* les indices du tableau de sommets.

Définissons une nouvelle structure :

Structure Graphe
    G : **Tableau[$1 \ldots N$] de Sommets**
    nomsTab : **Tableau[$1 \ldots N$] de Caratères**
fin structure

[2] G : **Tableau[$1 \ldots N$] de Sommets**

On a également de 2 nouvelles fonctions :

- **donne_indice(c :Caratère)** qui renvoi l'indice du sommet dont le nom est passé en paramètre.
- **donne_nom(i : Entier)** qui renvoi le nom du sommet dont l'indice est passé en paramètre.

```
Fonction 3sommets (G : Graphe, s1 : Sommet, s2 : Sommet, s3 : Sommet) retourne Graphe
Variables
    lesG : Tableau[1 ...6] de Graphes
    lesCouts : Tableau[1 ...6] d' Entiers
    g1, g2 : Graphes
    i : Entier
début
    i ← 1

    // 1er cas s1 s2 s3
    g1 ← Dijsktra(G, s1, s2)
    g2 ← Dijsktra(G, s2, s3)
    lesG[i] ← fusion(g1, g2)
    lesCouts[i] ← calcule_cout(lesG[i])
    i ← i+1

    // 2e cas s1 s3 s2
    g1 ← Dijsktra(G, s1, s3)
    g2 ← Dijsktra(G, s3, s2)
    lesG[i] ← fusion(g1, g2)
    lesCouts[i] ← calcule_cout(lesG[i])
    i ← i+1

    // 3e cas s2 s1 s3
    g1 ← Dijsktra(G, s2, s1)
    g2 ← Dijsktra(G, s1, s3)
    lesG[i] ← fusion(g1, g2)
    lesCouts[i] ← calcule_cout(lesG[i])
    i ← i+1

    // 4e cas s2 s3 s1
    g1 ← Dijsktra(G, s2, s3)
    g2 ← Dijsktra(G, s3, s1)
    lesG[i] ← fusion(g1, g2)
    lesCouts[i] ← calcule_cout(lesG[i])
    i ← i+1

    // 5e cas s3 s1 s2
    g1 ← Dijsktra(G, s3, s1)
    g2 ← Dijsktra(G, s1, s2)
    lesG[i] ← fusion(g1, g2)
    lesCouts[i] ← calcule_cout(lesG[i])
    i ← i+1

    // 6e cas s3 s2 s1
    g1 ← Dijsktra(G, s3, s2)
    g2 ← Dijsktra(G, s2, s1)
    lesG[i] ← fusion(g1, g2)
    lesCouts[i] ← calcule_cout(lesG[i])

    i ← indice_min(lesCouts)
    retourne lesG[i]
fin
```

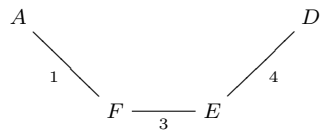
```
Fonction fusion (g1 : Graphe, g2 : Graphe) retourne Graphe
Variables
    i, j, k : Entiers
    v1, v2 : ^Voisins
    trouve : Booléen
    G : Graphe
début
    i ← 1
    // G est initialisé avec un  Tableau[1...M] de Sommets  "vide"
    // (ne contenant aucun voisin) correspondant aux sommets de g1 et g2 rassemblés
    // en un seul tableau sans doublon.

    tant que( i ≠ nombre_sommets(G) ) faire
        j ← donne_indice(g1, donne_lettre(G, i))
        k ← donne_indice(g2, donne_lettre(G, i))

        si( j ≠ -1 ) alors
            v1 ← g1.G[j].liste
        sinon
            v1 ← nil
        finsi

        si( k ≠ -1 ) alors
            v2 ← g2.G[k].liste
        sinon
            v2 ← nil
        finsi

        tant que( v1 ≠ nil ) faire
            ajoute_voisin(G, i, donne_indice(G, donne_nom(g1, v1^.indice)), v1^.valeur)
            v1 ← v1^.suivant
        fait

        tant que( v2 ≠ nil ) faire
            v1 ← G.G[i].liste
            trouve ← Faux

            tant que( v1 ≠ nil et non trouve ) faire
                si( donne_nom(g2, v2^.indice) = donne_nom(G, v1^.indice) ) alors
                    trouve ← Vrai
                finsi

                v1 ← v1^.suivant
            fait

            si( non trouve ) alors
                ajoute_voisin(G, i, donne_indice(G, donne_nom(g2, v2^.indice)), v2^.valeur)
            finsi

            v2 ← v2^.suivant
        fait
        i ← i+1
    fait
    retourne G
fin
```

4.3 Déroulement

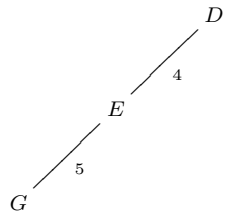
Reprenons le graphe G du §2.4 de la page 4.

Déterminons à l'aide de cet algorithme un plus court chemin passant par A, D et G.
L'algorithme va effectuer **Dijsktra**(A,D) puis **Dijsktra**(D,G) et enfin va fusionner les 2 sous-graphes obtenus et les stocker.
Idem pour tous les autres cas. La graphe retourné sera celui au poids le plus faible.

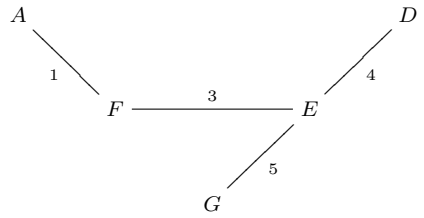
Voyons ce qui se passe pour le premier cas :
Le graphe retourné par **Dijsktra**(A,D) est le suivant :



Celui retourné par **Dijsktra**(D,G) est le suivant :



Après fusion de ces 2 sous-graphes, on a :



Tous les autres cas doivent être calculés pour savoir lequel des sous-graphes est de poids minimal.

4.4 Complexité

4.4.1 La fonction fusion

- Les 2 **tant que** les plus imbriqués effectuent au pire N itérations (Si le sommet est relié à tous les autres) $\rightarrow \mathcal{C} = O(N)$
- La boucle la plus à l'exterieur effectue N itérations.

Au total :

$$\boxed{\mathcal{C} = O(N^2)}$$

4.4.2 La fonction 3sommets

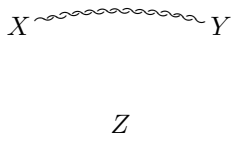
Dijsktra est en $O(N^2)$. On en effectue 12 en tout.
fusion est en $O(N^2)$. On en effectue 6 en tout.

Au total :

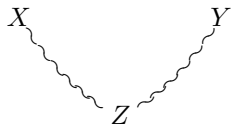
$$\boxed{\begin{aligned} \mathcal{C} &= 12N^2 + 6N^2 \\ &= 18N^2 \end{aligned}}$$

4.5 Preuve

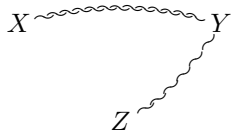
Il nous faut aller d'un sommet à un autre en passant par un autre sommet.
Soient $X, YetZ$ trois sommets. On veut donc que le plus court chemin passe par X, Y et par Z.
Dijsktra(X,Y) donne un plus court chemin de X à Y ne passant pas nécessairement par Z :



Il suffit donc de trouver un plus court chemin de X à Y pour que la condition soit respectée :



Mais il est peut-être plus rapide d'aller de X à Y puis de Y à Z :



C'est pourquoi on doit effectuer tous les cas.

Montrons maintenant que le chemin trouvé est bien le plus court.
Soient $X \rightsquigarrow Y \rightsquigarrow Z$ le plus court chemin.

Si ce chemin n'est pas le plus court, il existe un autre chemin passant par ces 3 sommets qui est plus court. Ces chemins peuvent être les suivants :

- $X \rightsquigarrow Z \rightsquigarrow Y$
- $Z \rightsquigarrow X \rightsquigarrow Y$
- $Z \rightsquigarrow Y \rightsquigarrow X$
- $Y \rightsquigarrow X \rightsquigarrow Z$
- $Y \rightsquigarrow Z \rightsquigarrow X$

Ils correspondent donc à tous les autres cas. Or la fonction retourne le graphe de poids minimum, donc $X \rightsquigarrow Y \rightsquigarrow Z$ est le plus court chemin car tous les autres cas ont également été explorés et leurs poids ont été calculés.

5 Le cas $|V'| = |V| - 1$

5.1 Implémentation

Dans cet algorithme il faut prendre tous les sommets sauf 1 choisi et on effectue un **Kruskall** sur ce sous-graphe. Si le graphe résultant est connexe, on le retourne sinon, on effectue **Kruskall** sur le graphe initial et on retourne le résultat. En effet, si le graphe n'est pas connexe, cela veut dire qu'il n'existe pas de solution optimale pour ce cas.

```
Fonction tousSaufUn (G : Graphe, S : Sommet) retourne Graphe
Variables
    G2 : Graphe
début
    G2 ← supprime_sommet(G, S)
    G2 ← Kruskall(G2)

    si( connexe(G2) ) alors
        retourne   G2
    sinon
        retourne   Kruskall(G)
    finsi
fin


Fonction supprime_sommet (G : Graphe, S : Entier) retourne Graphe
Variables
    G2 : Graphe
    i, j : Entiers
    courant : ^Voisin
début
    i ← 1
    j ← 1

    tant que( i ≠ nb_sommets(G) ) faire
        si( i ≠ S ) alors
            courant ← G.G[i].suiv
            tant que( courant ≠ nil ) faire
                si( courant^.indice ≠ s ) alors
                    ajoute_voisin(G2, j, donne_indice(G2, donne_nom(G, courant^.indice)),
                    courant^.valeur)
                finsi
                courant ← courant^.suiv
            fait
            j ← j+1
        finsi
        i ← i+1
    fait
    retourne G2
fin


Fonction connexe (G : Graphe) retourne Booléen
Variables
    L : Liste d' Entiers
début
    L ← parcours_profondeur(G)

    si( nombre_sommets(G) = nombre_elts(L) ) alors
        retourne Vrai
    sinon
        retourne Faux
    finsi
fin
```

5.2 Complexité

La boucle **tant que** la plus à l'intérieur effectue au pire N itérations.
La boucle **tant que** extérieure effectue N itérations.

Au total :

$$\boxed{\mathcal{C} = O(N^2)}$$

5.3 Preuve

En supprimant le sommet du graphe, on le déconnecte peut-être. Si le graphe n'est pas connexe, il faut passer par un autre sommet (celui supprimé) pour avoir un arbre couvrant minimal.

Si après la suppression, le graphe est toujours connexe, alors, il existe bien un arbre couvrant minimal. **Kruskall** ayant déjà été démontré, le sous-graphe obtenu est bien un sous-graphe optimal.

6 Le cas général

On peut utiliser la même méthode qu'à la *question 4*. C'est-à-dire, effectuer un **Dijsktra** sur tous les cas possibles et ne garder que le graphe au coût le plus faible.

6.1 Implémentation

Il nous faut d'abord déterminer un algorithme capable de déterminer tous les échanges possibles entre n sommets. Cet algortihme ne présentant pas d'interêt au niveau de l'algorithmique de graphes, il ne sera pas détaillé ici.
Rappel : Il y a A_n^n possibilités, c'est-à-dire $\frac{n!}{(n-n)!} = \frac{n!}{0!} = n!$ possibilités.

On passera à la fonction **calculeTousLesCas** un **Tableau[**$1 \dots N$**] d' Entiers** correspondant aux indices des sommets devant être dans le sous-graphe. La fonction retournera un **Tableau[**$1 \dots N!$**][**$1 \dots N$**] d' Entiers** .

```
Fonction casGeneral (G : Graphe, T : Tableau[1...N] d' Entiers ) retourne Graphe
Variables
    T2 : Tableau[1...N!][1...N] d' Entiers
    i, j : Entiers
    G1, G2 : Graphes
    TG : Tableau[1...N!] de Graphes
    TC : Tableau[1...N!] d' Entiers
début
    T2 ← calculeTousLescas(T)

    pour i ←1 à N! faire
        pour j ←1 à N-1 faire
            G1 ← Dijsktra(G, T2[i][j], T2[i][j+1])
            G2 ← fusion(G1, G2)          // La fusion avec un graphe vide donne le graphe
            initial
        fait

        TG[i] ← G2
        init(G2)          // init sert à réinitialiser un graphe comme à sa déclaration
    fait

    pour i ←1 à N! faire
        TC[i] ← calcule_cout(TG[i])
    fait

    retourne minimum(TC)
fin
```

6.2 Complexité

Les deux boucles **pour** imbriquées de la fonction **casGeneral** effectuent $(N-1)N!$. La 2e boucle **pour** effectue quand à elle $N!$ itérations.
Rappelons qu'on ne prend pas en compte la fonction **calculeTousLesCas**.

Au total :

$$\begin{aligned} \mathcal{C} &= (N-1)N! + N! \\ &= N!(N-1+1) \\ &= NN! \end{aligned}$$

Conclusion

Ce projet nous a permis de nous plonger dans l'algorithmique de graphes et de nous faire travailler des algorithmes importants comme **Dijsktra** et **Kruskall**. Il nous a également permis de réfléchir sur un sujet intérressant car il n'existe pas d'algorithme polinômial donnant un sous-graphe de poids minimal.

On pourrait maintenant essayer d'appliquer ces algorithmes dans un projet plus concret, comme par exemple la génération d'itinéraires de trains. En effet, les principaux problèmes de ces projets résident dans la lenteur des algorithmes pour la génération de ces itinéraires.