Rapport d'optimisation dans les graphes

# Une méthode d'optimisation pour la planification du câblage de réseau basée sur des arbres de couverture minimale

Professeure Superviseure:

Dominique QUADRI

Par:

Tianyi LI, Shutong ZHENG

October 16, 2022

# Contents

# 1 Description du scénario du problème et modélisation mathématique

## 1.1 Description du scénario du problème

Dans le domaine de l'ingénierie des réseaux, l'architecture et la planification des réseaux sont des techniques essentielles pour assurer le contrôle de la qualité et des coûts des services de réseau. Une architecture de réseau bien planifiée peut réduire considérablement les coûts de mise en place du réseau et garantir une qualité de connexion efficace. Le modèle client/serveur (modèle C/S) est largement utilisé dans la conception des architectures de réseau. Dans ce cas, il est important de s'assurer que le nœud client et le serveur se lient.

Dans ce projet, les liaisons de bout en bout $T$ et de bout en bout $S$ utiliseront des câbles à fibres optiques de communication avancés dont le coût est de k Le coût du câblage entre les nœuds peut être représenté dans la figure $G$.

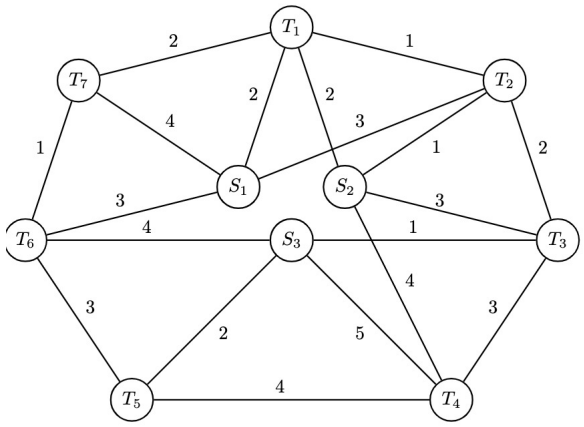


Figure 1: Coûts de câblage représentés dans le graphe

Les calculs effectués par le département financier indiquent que le contrôle des coûts de câblage sera un facteur clé pour déterminer la rentabilité du projet. L'objectif de l'optimisation de l'architecture pour ce projet était donc de réduire autant que possible le coût total du câblage. En même temps, les contraintes suivantes devaient être respectées pour la conception de l'architecture : chaque terminal devait pouvoir être relié directement ou indirectement à d'autres terminaux et à d'autres serveurs.

## 1.2 Modélisation mathématique du problème

Le problème de câblage de ce projet peut être représenté par un schéma. Les clients $T_i$ et les serveurs $S_j$ seront représentés comme des *sommets* dans un diagramme de graphe non orienté, et les câbles de réseau de bout en bout et de bout en bout du serveur seront représentés comme des *arêtes* dans le graphe non orienté. La définition formelle du graphe est donnée ci-dessous.

**Definition 1** *Un graphe est une paire $G = (V, E)$, où $V$ est un ensemble dont les éléments sont des sommets. Les sommets $x$ et $y$ d'une arête $x, y$ sont appelés les extrémités de l'arête*[1].
*En particulier, dans le cadre de ce projet, les éléments qui composent $V$ sont $T_i$ et $S_j$,et les poids de tous les $E$ doivent être des nombres réels positifs.*

Le cadre de ce projet donne les contraintes pour ce problème d'optimisation : trouver un ou plusieurs ensemble d'arêtes $A$ de $E$, $A$ doit joindre tous les sommets de $V$. Une représentation formelle stricte de la contrainte est présentée ci-dessous.

**Constraint 1** *Pour deux sommets $x$ et $y$ quelconques du graphe $G= (V,E)$, il doit exister une séquence de sommets et d'arêtes alternés $\Gamma = (x = v_0 - e_1 - v_1 - e_2 - ... - e_k - v_{k+1} = y)$*[2].

Le cadre de ce projet donne l'objectif de ce problème d'optimisation : trouver un ou plusieurs ensemble d'arêtes $A$ qui satisfont aux contraintes de sorte que la somme des poids de toutes les arêtes de $A$ soit minimisée.

En théorie des graphes, un ensemble d'arêtes $A$ satisfaisant à de telles contraintes et objectifs d'optimisation est également connu sous le nom d'arbre de couverture minimale. C'est un cas particulier d'arbre de Steiner minimal[3].La définition formelle d'un arbre à portée minimale est présentée ci-dessous.

**Definition 2** *Dans un graphe non orienté $G = (V, E)$, $(x, y)$représente l'arête reliant le sommet $x$ au sommet $y$ et $w(x, y)$ représente le poids de cette arête. S'il existe $A$ tel que tous les bords de $A$ sont des sous-ensembles de $E$ et que $A$ ne forme pas un cycle* [4]*, et tel que $w(A)$ qui joint tous les noeuds est minimal, alors ce $A$ est un arbre minimal de $G$.*

$$w(A) = \sum_{(x,y)\in A} w(x, y)$$

Sur la base du scénario du projet et des définitions mathématiques, les propositions suivantes peuvent être faites concernant l'arbre minimal couvrant $A$.

**Proposition 1** *$A$ est une structure arborescente.*

**Proposition 2** *$A$ est un acyclique.*

**Proposition 3** *$A$ relie tous les points de $V$.*

**Proposition 4** *Le nombre de sommets de $A$ est N, le nombre d'arêtes est N-1.*

**Proposition 5** *Parmi tous les arbres de couverture du graphe $G$, $A$ est celui dont la somme des poids est la plus petite.*

En particulier, dans le cadre de ce projet, le nombre d'arêtes de $A$ peut être exprimé en termes de nombre de serveurs $S$ et nombre de clients $T$.

**Proposition 6**

$$|A| = S + T - 1$$

Par conséquent, dans ce projet, l'architecture du réseau sera modélisée à l'aide du graphe $G$ et le f-tree minimal $A$ sera trouvé sur le graphe $G$ afin de dériver une solution de planification du réseau qui satisfait aux contraintes et aux objectifs d'optimisation.

# 2 Conception d'algorithmes et de structures de données

## 2.1 Conception d'algorithmes de recherche pour les arbres à couverture minimale

Quatre algorithmes classiques fournissent des solutions pour la recherche d'arbres à couverture minimale:
1. Algorithme de Boruvka
2. Algorithme de Prim
3. Algorithme de Kruskal
4. Algorithme de suppression inverse

Dans ce projet, nous allons d'abord suivre les étapes "manuelles", en utilisant du code pour montrer les détails de l'algorithme Prim en action. Ensuite, nous allons exécuter l'algorithme de Kruskal. Nous comparerons également les points communs et les différences entre les deux algorithmes.

Kruskal et Prim sont tous deux des algorithmes basés sur une stratégie gourmande, et leur stratégie d'exécution consiste à trouver, à chaque boucle, une arête qui ne viole pas une condition invariante de la boucle, puis à ajouter cette arête à l'ensemble $A$ des arbres de couverture que nous souhaitons construire. Leur logique de fonctionnement sera décrite dans les sections suivantes.

Le pseudo-code est donné ci-dessous:

**Algorithm 1** Algorithme de Prim

**Ensure:** Graphe non orienté pondéré G = (V,E,W)
**Require:** Un arbre à couverture minimale A = (V,T)
  //Initialisation
  $cost \leftarrow 0$
  $S \leftarrow 1$
  $T \leftarrow \emptyset$
  //Cycle principal
  **for** 1 to N-1 **do**
    Chercher l'arête (u,v) avec u$\in$ S et v$\notin$ S de poids $W_{u,v}$ minimal
    $T \leftarrow T \cup (u, v)$
    $S \leftarrow S \cup (v)$
    $cost \leftarrow cost + W_{u,v}$
  **end for**
  **return** A

**Algorithm 2** Algorithme de Kruskal

**Ensure:** Graphe non orienté pondéré G = (V,E,W)
**Require:** Un arbre à couverture minimale A = (V,T)
  //Initialisation
  $cost \leftarrow 0$
  $T \leftarrow \emptyset$
  //Cycle principal
  **while** |T|<N-1 **do**
    Chercher la plus petite arête (u,v) n'appartenant pas à T
    **if** $T \cup (u, v)$ ne contient pas de cycle **then**
      $T \leftarrow T \cup (u, v)$
      $cost \leftarrow cost + W_{u,v}$
    **else**
      Eliminer l'arête (u,v)
    **end if**
  **end while**
  **return** A

## 2.2 Structures de données utilisées dans les implementations des graphes

Lors de l'étude des structures de données utilisées dans la mise en œuvre d'un graphe, il est nécessaire de prendre en compte la sparsité du graphe. Nous considérons qu'un graphe $G = (V, E)$ est peu dense s'il satisfait aux conditions suivantes:

**Definition 3**

$$|E| << |V^2|$$

De même, pour G, on considère que G est dense si G satisfait aux conditions suivantes:

**Definition 4**

$$|E| \approx |V^2|$$

Pour un graphe non orienté $G = (V, E)$, il existe deux méthodes qui peuvent être utilisées pour le représenter, un table d'adjacence ou une matrice d'adjacence. Le choix entre ces deux méthodes dépend de la sparsité de $G$. Pour un graphe peu dense, la méthode des tables adjacence est souvent utilisée car elle est considérée comme plus compacte[5]. Alors que pour un graphe dense, la méthode de la matrice d'adjacence est plus souvent employée.

Pour le graphe $G$ à optimiser dans ce projet, il est clair que le graphe$G$est un graphe épars et qu'il est donc plus adapté à la représentation par table d'adjacence. Si l'on utilise une table d'adjacence, alors $G$ sera constitué d'un tableau de longueur $|V|$, où chaque membre contient toutes les arêtes connectées par le nœud correspondant respectivement.

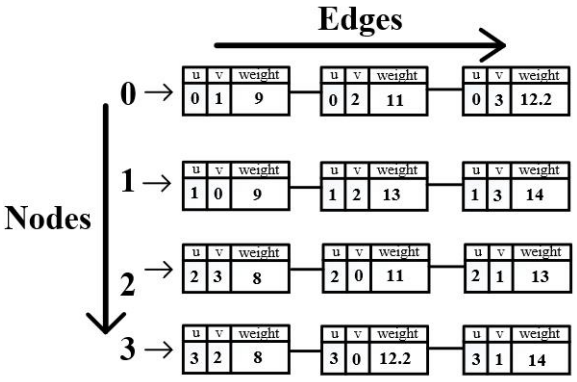


Figure 2: Table d'adjacence

Cette implémentation précédente de la table d'adjacence a une longueur de $2|E|$, et en tant que méthode classique, il en existe trop d'exemples.

Afin de montrer plus clairement le fonctionnement de l'algorithme Prim, nous allons le démontrer à l'aide d'une "méthode manuelle", qui sera présentée dans le chapitre suivant. En particulier, la structure de données utilisée pour représenter le graphe sera également programmée pour être fidèle à la structure de données utilisée dans la "méthode manuelle", nous avons donc apporté quelques modifications à la méthode des tables d'adjacence. Le diagramme ci-dessous montre la structure des données après nos modifications.

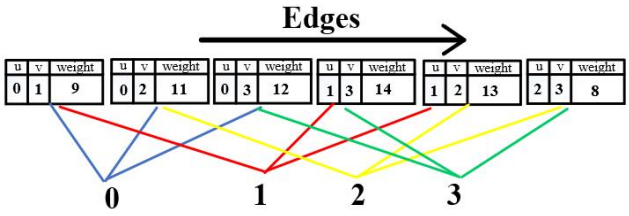


Figure 3: Structure utilisée pour représenter le graphique dans ce projet

Cette structure de données simplifie la table d'adjacence en ne nécessitant que la longueur de $|E|$ pour stocker le graphe. Cependant, cette approche présente des inconvénients lorsqu'il s'agit de rechercher des arêtes qui sont connectées à un sommet particulier. Mais nous voulions montrer plus clairement le fonctionnement de l'algorithme, c'est pourquoi nous avons fait ce choix.

Dans l'implémentation du code, nous définissons des classes Graphe, Edge et de structure Node pour fournir un support de structure de données pour le fonctionnement de l'algorithme Prim.Dans le diagramme d'une classe, le signe "+" indique Public et le signe "-" indique Privé.

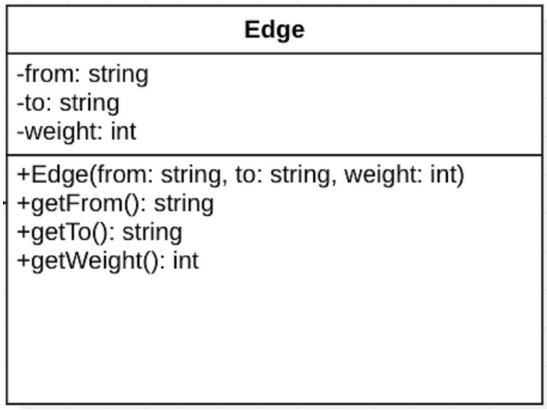


Figure 4: Class Edge

Pour la classe Edge, nous avons écrit des méthodes d'initialisation ainsi que des méthodes d'accès aux données des membres. Ces méthodes sont suffisantes pour que nous puissions exécuter l'algorithme.

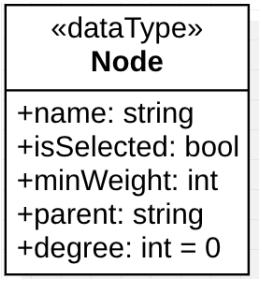


Figure 5: Structure Node

La structure Node possède quatre membres de données. Parmi ceux-ci,
"name" indique l'ID du vertex.
"isSelected" indique si le vertex est sélectionné ou non. Lorsqu'une arête est sélectionnée comme étant l'arête la moins pondérée dans la boucle actuelle,
"minWeight" indique la valeur du poids de l'arête. De plus, comme l'arête sélectionnée a toujours un sommet déjà dans l'arbre de couverture et un autre sommet en dehors de l'arbre de couverture,
"parent" indique le sommet de l'arête qui est à l'intérieur de l'arbre de couverture.
"dgree" est utilisé pour la détection des anneaux dans l'algorithme kruskal. Indique le degré d'un sommet. Lorsque le degré d'un sommet est $\leq 1$, il est temporairement supprimé et le degré des sommets qui lui sont connectés est également réduit de un. À la fin de la boucle de l'algorithme de détection d'anneau, s'il reste des sommets non supprimés, alors l'anneau existe et les sommets sont tous de degr $\geq 2$.

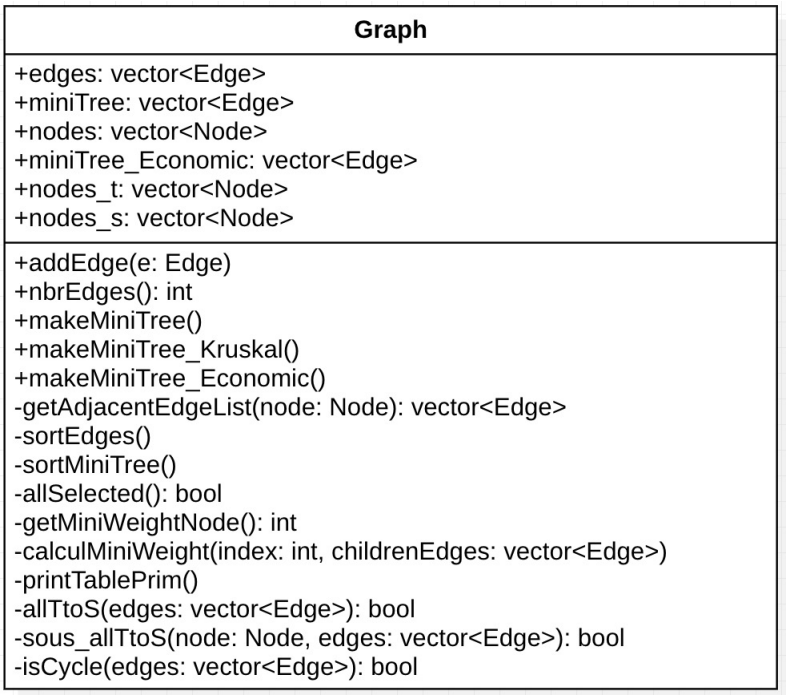


Figure 6: Class Graphe

La classe Graphe a six variables membres, et une série de méthodes qui servent l'algorithme.Pour le conteneur de Edge, nous avons choisi std::Vector plutôt que std::List car la taille des données pour ce projet est petite et Vector est le plus rapide lorsqu'il s'agit de parcourir les données.

- Parmi ces variables membres,

"edges" représente l'ensemble de toutes les arêtes.
"miniTree" représente l'ensemble de toutes les arêtes de l'arbre à couverture minimale. "nœuds" désigne l'ensemble de tous les sommets.
"miniTree Economic" répond à la demande de la question 2 du document des exigences du projet que ce "miniTree Economic" contient S composants connectés.
"nodes t" désigne l'ensemble de tous les sommets ayant l'attribut "t", c'est-à-dire le client étiqueté T dans le graphe G.
"nodes s" désigne l'ensemble de tous les sommets ayant l'attribut "s", c'est-à-dire le serveur étiqueté S dans le graphe G.

- Parmi ces fonctions membres,

"addEdge (Edge e);" ajoute des arêtes au graphe G et, puisque le graphe est non orienté, les arêtes en double sont supprimées (par exemple Edge("S1", "T1",2) , Edge("T1", "S1",2) n'en conservera qu'un seul).
"nbrEdges();" renvoie le nombre d'arêtes.

"getAdjacentEdgeList(Node node);" renvoie toutes les arêtes connectées à ce sommet.
"allSelected();" vérifie si tous les sommets ont été sélectionnés. Si tous les sommets sont sélectionnés, la méthode renvoie "True", sinon elle renvoie "false".
"printTablePrim();" affiche les étapes de l'exécution de l'algorithme Prim et indique clairement l'état de chaque nœud au moment de la boucle actuelle. Tout comme nous le faisons manuellement dans le chapitre suivant.
"getMiniWeightNode();" trouver les nœuds avec la plus petite somme de poids utilisée dans l'algorithme prim, elle renvoie des nœuds avec une somme de poids minimale.
"calculMiniWeight(int index, vector<Edge> childrenEdges);" prend deux paramètres:"index " indice du nœud courant, "childrenEdges" est l'ensemble des nœuds connectés au noeud courant. Cette méthode calcule le poids minimum du noeud adjacent d'un noeud. Et si la valeur miniWeight du nœud adjacent est supérieure au poids du nœud actuel qui lui est connecté. La valeur du poids du noeud adjacent est remplacée. La valeur de Parent est également remplacée par le noeud courant.
"makeMiniTree();" générer un arbre couvrant minimal par l'algorithme prim.
"sortMiniTree();" trier l'arbre couvrant minimum, par ordre décroissant selon la valeur de poids
"sous allTtoS(Node node, vector<Edge> edges);" est une sous-programme récursif. Vérifier si un noeud peut avoir au moins un S qui lui est connecté directement ou indirectement par récursivité. Il renvoie "true" si le noeud peut être connecté à S. L'inverse renvoie "false".
"allTtoS(vector<Edge> edges);" voir si tous les T ont au moins un S qui peut communiquer. Ils renvoient "true" si ils peuvent tous communiquer.
"makeMiniTree Economic();"répond à la Question 2 des exigences du projet. Cette méthode modifie la contrainte d'être connecté (directement ou indirectement) à au moins un serveur. Le nombre de composants connectés que cette méthode retournera est s.
"makeMiniTree Kruskal();" génération d'un arbre couvrant minimal à l'aide de l'algorithme de Kruskal.
"sortEdges();"trier ensemble d'arètes du plus petit au plus grand en utilisant l'algorithme à bulles.
"isCycle(vector<Edge> edges);"déterminer si un graphe non orienté a des cycles, il renvoie vrai s'il y a une cycle, sinon renvoie faux. Lorsque le degré d'un sommet est $\leq 1$, il est temporairement supprimé et le degré des sommets qui lui sont connectés est également réduit de un. À la fin de la boucle de l'algorithme de détection d'anneau, s'il reste des sommets non supprimés, alors l'anneau existe et les sommets sont tous de degré $\geq 2$.
"deleteNode(Node node,vector<Edge> edges, vector<Node> nodes);" supprimer un nœud dans l'ensemble de nœuds et réduire de 1 le degré d'un autre nœud connecté à ce nœud.
"existNode(vector<Node> nodes, string node);" déterminer si l'ensemble de nœuds contient le nœud requis, il renvoie vrai si le nœud est inclus sinon renvoie faux.

"getDegreeNode(Node node, vector<Edge> edges);" obtenir le degré du nœud, il renvoie le degré du nœud.
"hasDegreeInf1(vector<Node> nodes);" déterminer s'il existe un nœud de degré inférieur ou égal à 1, il renvoie true s'il existe un nœud de degré inférieur ou égal à 1, sinon renvoie false.
"deleteEdge(Node node, vector<Edge> edges);"supprimer une arête contenant un nœud, il renvoie l'ensemble des arêtes après avoir supprimé le nœud.

### 2.3 Structures de données utilisées dans la recherche pour les arbres à couverture minimale

Afin d'implémenter efficacement l'algorithme Prim, une méthode rapide est nécessaire pour sélectionner une nouvelle arête à ajouter à l'arbre formé par l'ensemble A. La préséance est considérée comme un choix approprié. Par conséquent, nous devons choisir une structure de données appropriée pour mettre en œuvre la file d'attente prioritaire afin qu'elle fonctionne bien avec l'algorithme. Le tas minimal binaire est la structure de données utilisée par std::priority queue. En outre, le tas de Fibonacci est considéré comme une mise en œuvre plus efficace lorsqu'il est utilisé conjointement avec l'algorithme Prim.

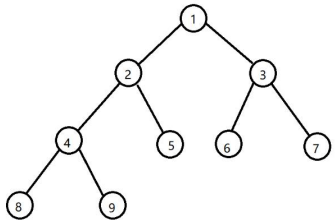


Figure 7: Tas minimal binaires

Toutefois, afin de recréer fidèlement l'"opération manuelle", dans l'implémentation de l'algorithme Prim, nous utilisons une opération de traversée d'une complexité en temps $O(V)$ pour obtenir l'arête ayant le poids le plus faible.

## 3 Description du fonctionnement de l'algorithme avec étapes détaillées de l'exécution

### 3.1 Description de la stratégie générale du fonctionnement des algorithmes.

L'algorithme Prim est basé sur une stratégie gourmande. Les algorithmes basés sur des stratégies gourmandes suivent tous une approche générale : à chaque étape de la boucle, trouver un bord pour l'arbre de couverture minimale et

maintenir un ensemble de bords A tout au long de l'exécution de la stratégie, en s'assurant que l'ensemble de bords A suit toujours l'invariant de la boucle. L'opération clé pour maintenir l'ensemble des arêtes $A$ est de trouver une nouvelle arête $(u, v)$qui satisfait à la condition d'invariant cyclique de sorte que $A \cup \{(u, v)\}$ soit également un sous-ensemble d'un certain arbre de couverture minimale. Un tel bord $(u, v)$ satisfaisant à la condition est appelé "arête sûre".

**Algorithm 3** Stratégie gourmande pour trouver le MST dans G

```
A = ∅
while A ne constitue pas un arbre de couverture do
  Trouver une arête (u,v) qui est "sûre" pour A
  A = A ∪ (u, v)
end while
return A
```

Par conséquent, les règles permettant d'identifier si un bord est un "arête sûre" sont essentielles au succès de l'algorithme. Pour décrire cette règle, nous devons définir la "coupe" du diagramme ainsi que d'autres descriptions étroitement liées.

**Definition 5** *Une "coupe" (S, V-S) d'un graphe non orienté G=(V,E) est une division de l'ensemble V.*

**Definition 6** *Pour une arête (u,v), si un sommet de l'arête est dans l'ensemble S et un sommet est dans l'ensemble V-S, on dit que l'arête "traverse" la coupe (S,V-S).*

**Definition 7** *On dit d'une coupe qu'elle "respecte" l'ensemble A si aucune arête ne "traverse" la coupe dans l'ensemble A.*

**Definition 8** *Parmi tous les arêtes qui "couvrent" une coupe, celle qui a le moins de poids est appelée "arête légère". Les "arêtes légères" ne sont pas forcément uniques.*

Ainsi, dans le processus de recherche de l'arbre de couverture minimum. Si l'ensemble $A$ est un sous-ensemble d'un arbre à couverture minimale, soit $(S, V - S)$ toute coupe dans le graphe non orienté $G$ qui "respecte" l'ensemble $A$, et soit $(u, v)$ une "arête légère" qui traverse cette coupe $(S, V - S)$. Ensuite, l'arête $(u, v)$ est une "arête sûre" pour cet ensemble$A$ : même après $A \cup \{(u, v)\}$, $A$ reste un sous-ensemble d'un arbre de couverture minimal.

Les algorithmes de Kruskal et de Prim sont tous deux basés sur une stratégie avide de trouver des " arêtes sûres ". Dans l'algorithme de Prim, on veut construire un ensemble $A$ qui ne forme qu'un arbre. Une "arête sûre" ajoutée à l'ensemble $A$ est toujours l'arête qui relie $A$ à un sommet de poids minimum qui n'est pas dans $A$. Dans l'algorithme de Kruskal, l'ensemble $A$ est une forêt, et l'"arête sûre" ajoutée à l'ensemble $A$ est toujours l'arête de poids minimum qui relie deux branches connectées différentes dans le graphe $G$.

## 3.2 Description du fonctionnement de l'algorithme Prim

Dans l'algorithme Prim, nous utilisons la file d'attente de priorité minimale $Q$pour trouver le "arête sûre" ayant le poids le plus minime. À chaque étape de l'algorithme, nous aurons une coupe $(V-Q,Q)$, qui est le sommet qui a été ajouté à l'ensemble des arbres de couverture minimale. Pour toutes les arêtes " traversant " cette coupure, elles ont deux sommets : un sommet appartient à $V-Q$ et l'autre à $Q$. La file d'attente de priorité minimale $Q$ trouve le sommet qui appartient à l'" arête légère " et on transfère ce sommet de $Q$ à $V-Q$, complétant ainsi un cycle. A la fin de toutes les boucles, $V-Q=V$ et Q=$\emptyset$.

En particulier, dans l'algorithme Prim, il est important de noter que l'ajout aveugle de toutes les arêtes "à travers" $(V-Q,Q)$ à la file d'attente prioritaire directement lors de l'implémentation du code entraînera une complexité temporelle de $O(ElgE)$. Nous devons ajouter l'arête " traversante " $(u,v)$ à la file d'attente prioritaire avec le sommet $v$ qui n'appartient pas à $V-Q$. De cette façon, lorsque nous trouvons une arête plus courte vers $v$, nous pouvons mettre à jour les poids des arêtes pointant vers$v$ dans la file d'attente, ce qui nous donne une complexité temporelle de $O(ElgV)$. (Puisque $|E|>|V|$ dans la plupart des graphes, nous considérons que $O(V)$ est supérieur à $O(E)$ dans l'algorithme des graphes.)

### 3.3 Description du fonctionnement de l'algorithme Kruskal

Dans l'algorithme de Kruskal, à chaque tour, nous trouvons l'arête $(u, v)$ ayant le plus petit poids parmi toutes les arêtes reliant deux arbres quelconques de la forêt comme "arête sûre" et nous ajoutons cette "arête sûre" à la forêt en croissance. La "cupidité" de Kruskal se reflète dans le fait qu'à chaque étape de l'algorithme, les poids des arêtes ajoutées à la forêt sont aussi faibles que possible.

L'algorithme de Kruskal nécessite l'utilisation d'une structure de données d'ensembles disjoints. Pour une "arête légère" $(u, v)$, l'ensemble disjoint nous permet de déterminer si $u$ et $v$ appartiennent au même ensemble, si non, nous savons que $u$ et $v$ n'appartiennent pas au même arbre, et nous pouvons alors fusionner les deux arbres et mettre à jour l'ensemble disjoint.

$\mathrm{P}_0 = \{\{S_1\}, \{S_2\}, \{S_3\}, \{T_1\}, \{T_2\}, \{T_3\}, \{T_4\}, \{T_5\}, \{T_6\}, \{T_7\}\}$
$P_1 = \{\{S_1\}, \{S_2\}, \{S_3\}, \{T_1, T_2\}, \{T_3\}, \{T_4\}, \{T_5\}, \{T_6\}, \{T_7\}\}$
$\mathrm{P}_2 = \{\{S_1\}, \{S_2\}, \{S_3\}, \{T_1, T_2\}, \{T_3\}, \{T_4\}, \{T_5\}, \{T_6, T_7\}\}$
$P_3 = \{\{S_1\}, \{S_2\}, \{S_3, T_3\}, \{T_1, T_2\}, \{T_4\}, \{T_5\}, \{T_6, T_7\}\}$
$P_4 = \{\{S_1\}, \{S_3, T_3\}, \{T_1, T_2, S_2\}, \{T_4\}, \{T_5\}, \{T_6, T_7\}\}$
$P_5 = \{\{S_1\}, \{S_3, T_3, T_5\}, \{T_1, T_2, S_2\}, \{T_4\}, \{T_6, T_7\}\}$
$P_6 = \{\{S_3, T_3, T_5\}, \{T_1, T_2, S_2, S_1\}, \{T_4\}, \{T_6, T_7\}\}$
$P_7 = \{\{S_3, T_3, T_5\}, \{T_1, T_2, S_2, S_1, T_6, T_7\}, \{T_4\}, \}$
$P_8 = \{\{T_1, T_2, S_2, S_1, T_6, T_7, S_3, T_3, T_5\}, \{T_4\}, \}$
$P_9 = \{\{T_1, T_2, S_2, S_1, T_6, T_7, S_3, T_3, T_5, T_4\}, \}$

Figure 8: Etape de Kruskal

## 3.4 Étapes détaillées de l'exécution de l'algorithme Prim

Afin de montrer clairement le fonctionnement de l'algorithme Prim, nous avons "exécuté manuellement" l'algorithme et répertorié l'état de chaque nœud à chaque boucle.

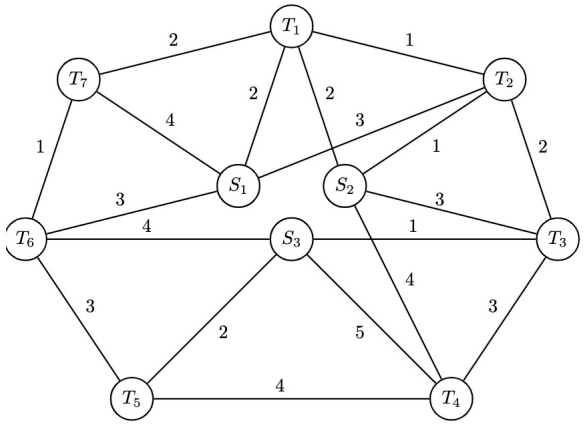


Figure 9: Graphe G à optimiser

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | S1 | S2 | S3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Selected | T | F | F | F | F | F | F | F | F | F |
| miniDis | / | 1 | | | | | 2 | 2 | 2 | |
| Parent | | T1 | | | | | T1 | T1 | T1 | |

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | S1 | S2 | S3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Selected | T | T | F | F | F | F | F | F | F | F |
| miniDis | / | 1 | 2 | | | | 2 | 2 | 1 | |
| Parent | | T1 | T2 | | | | T1 | T1 | T2 | |

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | S1 | S2 | S3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Selected | T | T | F | F | F | F | T | F | T | F |
| miniDis | / | 1 | 2 | 4 | | 1 | 2 | 2 | 1 | |
| Parent | | T1 | T2 | S2 | | T7 | T1 | T1 | T2 | |

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | S1 | S2 | S3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Selected | T | T | F | F | F | T | T | F | T | F |
| miniDis | / | 1 | 2 | 4 | 3 | 1 | 2 | 2 | 1 | 4 |
| Parent | | T1 | T2 | S2 | T6 | T7 | T1 | T1 | T2 | T6 |

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | S1 | S2 | S3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Selected | T | T | T | F | F | T | T | F | T | F |
| miniDis | / | 1 | 2 | 3 | 3 | 1 | 2 | 2 | 1 | 1 |
| Parent | | T1 | T2 | T3 | T6 | T7 | T1 | T1 | T2 | T3 |

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | S1 | S2 | S3 |
|---|---|---|---|---|---|---|---|---|---|---|
| Selected | T | T | T | T | T | T | T | T | T | T |
| miniDis | / | 1 | 2 | 3 | 3 | 1 | 2 | 2 | 1 | 1 |
| Parent | | T1 | T2 | T3 | T6 | T7 | T1 | T1 | T2 | T3 |

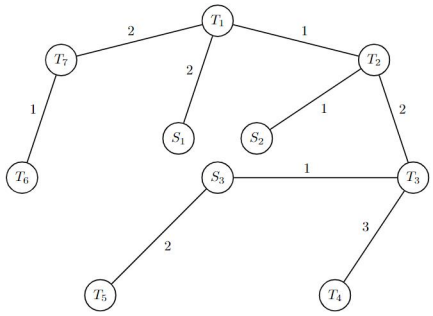


Figure 10: Arbre de couverture minimale pour la graphe G

# 4 Analyse de la complexité de l'algorithme et comparaison des effets des algorithmes de Prim et de Kruskal

## 4.1 Analyse de la complexité de l'algorithme Prim

Dans le code, l'algorithme Prim est en fait exécuté par la fonction "makeMiniTree". Dans ce cas, nous appelons plusieurs fonctions pour effectuer le calcul.

1. La méthode "getAdjacentEdgeList" produit une collection d'arêtes adjacentes aux sommets. Cette fonction utilise une boucle for pour augmenter le nombre d'arêtes de 0 à 1 à chaque fois, pour un total de E boucles. Par conséquent, sa complexité temporelle est $O(E)$.
2. la méthode "calculMiniWeight" peut être utilisée pour trouver le poids minimum. La fonction utilise 2 boucles for imbriquées l'une dans l'autre, chacune sans interférer avec l'autre. La complexité temporelle est $O(V * E)$.
3. "printTablePrim" est utilisé pour imprimer la table Prim. La complexité temporelle est de$O(V)$.

En plus de cela, au début de la fonction, nous effectuons un jugement de sommet pour déterminer si un sommet a été sélectionné, avec une complexité temporelle de $O(V)$. À la fin de toutes les boucles, nous insérons le résultat dans le tableau miniTree, avec une complexité temporelle de $O(V)$.
Ainsi, la complexité en temps de "makeMiniTree" est:

$$O(V * (E + V * E + V) + V) = O(E * V^2)$$

Nous obtenons cette complexité temporelle parce que nous voulons reproduire fidèlement notre recherche "manuelle" du bord minimal. Si nous utilisons une file d'attente à priorité minimale de type arbre binaire, la complexité temporelle sera de $O(ElgV)$, et si nous utilisons un tas de Fibonacci, la complexité temporelle sera de $O(E + VlgV)$.

Pour répondre à la question 2 du document des exigences du projet, nous avons écrit la méthode "makeMiniTree Economic". Il appelle également plusieurs fonctions pour effectuer le calcul.

1. "sortMiniTree" utilise la méthode de tri à bulles, qui trie chaque arête en fonction de son poids, du plus grand au plus petit, de sorte que la complexité temporelle est $O(E^2)$.
2. "allTtoS" détermine si tous les $T$ sont connectés à au moins un $S$. Dans la fonction, une boucle for est utilisée pour faciliter tous les sommets $T$, et la fonction récursive "sous allTtoS" est appelée dans la boucle pour effectuer des opérations sur les sommets T." sous allTtoS" a une complexité temporelle de $O(E^2)$, donc "allTtoS" a une complexité temporelle de $O(T * E^2)$.

En outre, nous classons les "nœuds s" et "nœuds t" avec une complexité temporelle de $O(V)$. Nous effectuons un partitionnement S-1 du miniThree. Et enfin insérer les composantes connectées partitionnées dans le tableau bidimensionnel de miniTree Economic et imprimer le résultat.

Ainsi, la complexité temporelle finale de la méthode "makeMiniTree Economic" est:

$$O(E^2 + T * E^2 + S^2 + (S - 1 + V)) = O(T * E^2)$$

## 4.2 Analyse de la complexité de l'algorithme Kruskal

La fonction "makeMiniTree Kruskal" exécute réellement l'algorithme de Kruskal. Elle appelle plusieurs fonctions.
1. "sortEdges" effectue un tri à bulles sur l'ensemble des arêtes avec une complexité temporelle de $O(E^2)$.
2. La fonction "isCycle" détermine s'il y a une boucle ou non. Sa complexité temporelle est de $O(E + V^2 + V(E + E)) = O(V^2)$. Cette fonction est exécutée V fois durant tout l'algorithme.
3. Imprimer l'arbre de couverture minimale avec une complexité temporelle de $O(V)$.

Ainsi, la complexité temporelle finale de "makeMiniTree Kruskal" est de

$$O(E^2 + V * V^2 + V) = O(V^3)$$

La complexité temporelle de l'algorithme de Kruskal dépend de la manière dont la structure de données des ensembles disjoints est implémentée. L'algorithme de Kruskal peut atteindre une complexité temporelle de $O(ElgV)$ s'il est implémenté en code en utilisant une forêt d'ensembles disjoints avec fusion par rang et compression de chemin.

## 4.3 Comparaison des effets des algorithmes de Prim et de Kruskal

La complexité asymptotique en temps des algorithmes de Kruskal et de Prim peut être de $O(ElgV)$ si la structure de données idéale est utilisée. Cependant, la complexité temporelle de Kruskal est en fait $O(ElgE)$, qui ne peut être réduite à $O(ElgV)$ que si $|E| < |V^2|$, donc la complexité temporelle de Kruskal est en fait indépendante du nombre de V. Par conséquent, Kruskal est plus adapté aux "graphes épars" où le nombre d'arêtes est relativement faible ; Prim est un algorithme dépendant des sommets, qui convient mieux aux graphes à densité de points.

# 5 Réponse : les questions de la PARTIE 1 dans la description de la tâche

## 5.1 Réponse à la question 1

Question 1:
Que doit on chercher dans G ? Exprimer en fonction de s et t et le nombre de liaisons r ealiser.

Reponse:

Dans le graphe $G$, nous voulons trouver un arbre couvrant minimum$G'$avec les poids les plus petits.
C'est-à-dire que le graphe $G'$ doit satisfaire les points suivants :
1. $G'$ est une structure arborescente, il ne peut donc y avoir de cycles dans$G'$.
2. Tous les sommets de $G'$ doivent être connectés.
3. Le nombre de sommets est $N$, alors le nombre d'arêtes est $N-1$
4. $G'$ est l'arbre avec le plus petit poids parmi tous les arbres couvrant minimum générés par $G$
La somme des arêtes de $G'$ est $s+t-1$

## 5.2 Réponse à la question 2

Question 2 partie 1:
On consid'ere l'exemple illustratif repr esent e par le graphe G1. determinier, en utilisant l'algorithme de Prim une solution optimale et le coût correspondant.
Reponse:

Après avoir exécuté l'algorithme de Prim, nous constatons que le graphe G n'a qu'un seul arbre de couverture minimum.Il est illustré dans la figure suivant:

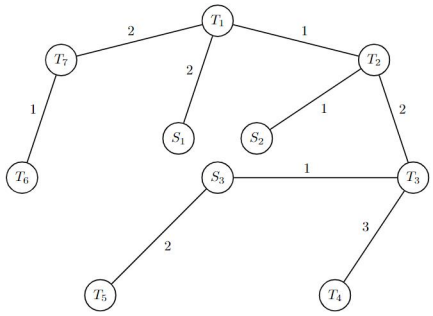


Figure 11: Arbre de couverture minimale pour la graphe G

Question 2 partie 2:
En fait le concepteur du r eseau souhaite, pour des raisons economiques, etudier des architectures ou' chaque terminal ne serait reli e de fa con directe ou indirecte qu'a' un et un seul serveur.

Reponse:
Nous avons divisé le diagramme. Le partitionnement présenté ci-dessous minimise le coût total, tout en garantissant que tous les $T$ peuvent être reliés directement ou indirectement à un seul $S$.

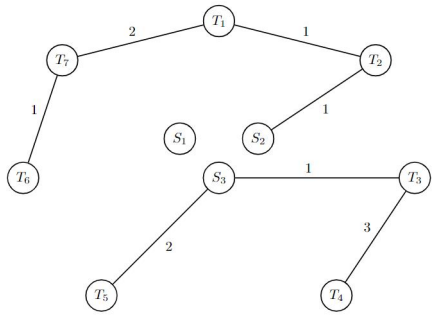


Figure 12: Une solution de segmentation qui minimise les coûts

## 5.3 Réponse à la question 3

Question 3 (a):
Le nombre de composante connexes du réseau à constituer. Dans un graphe non orienté, une composante connexe est un sous- graphe induit maximal connexe, c'est-à-dire un ensemble de points qui sont reliés deux à deux par un chemin. On peut ainsi regrouper les sommets d'un graphe selon leur appartenance à la même com- posante connexe.

Reponse:
Nombre de group:

$$p = s$$

p est le nombre de composantes connexes de G
Parce que chaque terminal sera directement ou indirectement connecté à un seul et un seul serveur.

Question 3 (b):
Le nombre de liaisons à réaliser (on pourra utiliser ce que l'on appelle le nombre cyclomatique d'un graphe quelconque G, noté $V(G)$qui est défini par:
$V(G) = M - N - P$
ou m est le nombre d'aretes de G, n le nombre de sommets de G et p le nombre de composantes connexes de G.
De plus, il vérifie la propriété suivante :$v(G) = 0$ si et seulement si G ne contient pas de cycle. Si l'on note m* le nombre de liaisons à réaliser, vous pourrez définir que l'on a:

$$V(F^*) = m^* - (s + t) + s = 0$$

où $F^*$ est le graphe partiel de G correspondant au réseau recherché et est sans cycle car chaque terminal doit être relié de façon unique à son serveur.

Reponse:
Nous établissons les définitions suivantes:
p est le nombre de composantes connexes de G
m est le nombre d'arêtes de G
n est le nombre de sommets de G
Donc,

$$p = s$$
$$m = s + t - s = t$$
$$n = s + t$$

$$\begin{aligned} V(G) &= m - n + p \\ &= t - s - t + s = 0 \end{aligned}$$

Du coup, le graphe G n'a pas cycle.

# 6 Bibliographie

# References


[1] Trudeau, Richard J. "Introduction to Graph Theory" (1993) p.19.

[2] Balakrishnan, V. K. "Graph Theory" (1997)

[3] Jillian Beardwood, John H Halton and John Michael Hammersley, " The shortest path through many points" *Volume 55 of Mathematical Proceedings of the Cambridge Philosophical Society* (1959) p.299-327.

[4] Numpy and Scipy Documentation "scipy.sparse.csgraph.minimum spanning tree" (2021)

[5] CLRS the third edition (1990) p.341.