Mst package - Rapport

Nawal Bendjelloul, Rabab Khatib, Gabriele Pedroni

2024-04-11

Introduction

Contexte

Le Minimum Spanning Tree (MST) est un arbre couvrant de poids minimal d'un graphe connexe et pondéré. Il s'agit d'un sous-graphe acyclique qui relie tous les sommets du graphe original en minimisant la somme des poids des arêtes.

Il existe plusieurs algorithmes classiques pour trouver le MST d'un graphe, tels que l'algorithme de Kruskal et l'algorithme de Prim que nous avons choisi d'implementer ici. Ces algorithmes reposent sur des stratégies différentes pour sélectionner les arêtes à inclure dans l'arbre couvrant, mais ils garantissent tous les deux de trouver une solution optimale.

Ce package se veut notamment de résoudre efficacement des problèmes de MST sur des graphes de grande taille, spécifiques à la modélisation dans le domaine de la santé.

Applications

Le MST a de nombreuses applications en santé, notamment dans l'analyse de données génomiques et protéomiques [étude à grande échelle des protéines]. Parmi les exemples d'applications en santé nous pouvons citer :

- La **reconstruction de génomes** d'organismes à partir de fragments d'ADN. En utilisant un graphe dont les sommets représentent les fragments d'ADN et les arêtes représentent les chevauchements entre les fragments, le MST permet de trouver le chemin le plus court qui relie tous les fragments, ce qui correspond au génome complet de l'organisme.
- L'**analyse de réseaux métaboliques**, en représentant par les sommets les métabolites et par les arêtes les réactions chimiques entre eux. Le MST peut être utilisé pour identifier les voies métaboliques les plus importantes dans le réseau, en sélectionnant les arêtes qui ont le poids le plus faible et donc les réactions les plus efficaces.
- L'**analyse de données de séquençage**, où le MST est utilisé pour comparer des séquences d'ADN ou d'ARN et identifier les mutations qui les distinguent. En construisant un graphe dont les sommets représentent les séquences et les arêtes représentent les différences entre les séquences, le MST permet de trouver le chemin le plus court qui relie toutes les séquences, ce qui correspond à l'ensemble des mutations qui les séparent.

Algorithme de Prim

L'**algorithme de Prim**, popularisé par Prim en 1957, est un algorithme de graphes trouvant un **arbre couvrant minimal** pour un graphe connexe pondéré non dirigé.
Son objectif est de connecter tous les sommets du graphe avec les arêtes de poids minimal possible, sans former de cycle, pour que le poids total de l'arbre couvrant minimal soit le plus bas possible.

Principe

L'algorithme de Prim commence par un **sommet arbitraire** et s'étend étape par étape jusqu'à couvrir tous les sommets du graphe. À chaque étape, il ajoute l'**arête de poids minimal disponible** qui connecte un sommet de l'arbre en construction à un sommet en dehors de l'arbre.

Etapes de l'algorithme

1. **Initialisation** : Sélectionner un sommet de départ arbitraire du graphe.
2. **Sélection des Arêtes** : À partir du sommet sélectionné, choisir l'arête de poids minimal qui connecte un sommet déjà dans l'arbre à un sommet qui n'est pas encore dans l'arbre.
3. **Ajout à l'Arbre** : Ajouter le sommet sélectionné à l'arbre, ainsi que l'arête choisie.
4. **Répétition** : Répéter les étapes 2 et 3 jusqu'à ce que tous les sommets soient inclus dans l'arbre.

Propriétés

- **Greedy Algorithm** : à chaque étape, il fait le choix optimal local dans l'espoir que ce choix conduira à une solution optimale globale.
- **Indépendant du Sommet de Départ** : peu importe le sommet de départ, l'algorithme trouve toujours le même arbre couvrant minimal. Cela montre la fiabilité de l'algorithme pour fournir un résultat consistant et optimal indépendamment de sommet initial choisi.

Algorithme de Kruskal

Principe

L'algorithme de Kruskal est un algorithme utilisé pour trouver le minimum spanning tree (MST) d'un graphe non orienté et connecté. Le MST d'un graphe est le sous-ensemble d'arêtes qui relie tous les sommets du graphe avec le poids total le moins élevé possible, sans former de cycles.

Etapes de l'algorithme

Voici les étapes de l'algorithme de Kruskal:

- **Triez les arêtes par poids** : Classer toutes les arêtes du graphique par ordre croissant en fonction de leur poids. La priorité accordée aux arêtes les moins coûteuses permet de construire le MST de manière efficace.
- **Initialiser les ensembles** : Initialement, chaque sommet du graphe est considéré comme un ensemble disjoint distinct.
- **Itérer sur les edges** : Traiter les arêtes une par une, en commençant par la moins chère
  - Si l'ajout d'une arête ne crée pas de cycle (loop) dans la MST partiellement construite, l'ajouter à l'ensemble de la MST.
  - Si l'ajout de l'arête crée un cycle, l'écarter.
- **Opération d'union** : Pour chaque arête, si les sommets qu'elle relie appartiennent à des ensembles différents, unir ces ensembles (les fusionner). Cette étape permet de s'assurer que l'ajout de l'arête à la MST ne crée pas de cycle.
- **Arrêtez lorsque tous les sommets sont connectés:** : Continuez à traiter les arêtes jusqu'à ce que vous ayez suffisamment d'arêtes dans l'ensemble MST pour connecter tous les sommets du graphe (vous devriez avoir V-1 arêtes). Une fois que tous les nœuds sont connectés, vous avez trouvé la MST, qui représente le réseau dont le coût total est le plus faible.

Propriétés

L'algorithme de Kruskal est **greedy**, ce qui signifie qu'il fait des choix localement optimaux (ajout de l'arête la moins chère) dans le but d'obtenir un résultat globalement optimal (coût total minimum du MST).

Implémentation en R

Dans cette section, nous allons détailler l'implémentation des algorithmes de Prim et de Kruskal pour trouver un arbre couvrant de poids minimum dans un graphe pondéré en R. Nous aborderons également les choix d'implémentation et calculerons la complexité de ces algorithmes.

Algorithme de Prim

L'algorithme de Prim fonctionne en sélectionnant à chaque étape l'arête de poids minimum reliant un sommet de l'arbre à un sommet hors de l'arbre.

L'algorithme prend en entrée la **matrice d'adjacence** répresentant le graphe : le choix de cette structure pour implementation est due à la nécessite d'acceder rapidement aux poids des arêtes.

Initialisation des structures de données

```
n <- nrow(adj_matrix)
parent <- rep(NA, n)
key <- rep(Inf, n)
mst_set <- rep(FALSE, n)
key[1] <- 0
```

Les structures de données nécessaires pour l'algorithme de Prim sont :

- Le tableau `parent` pour stocker les parents des sommets dans l'arbre partiel en cours de construction.
- Le tableau `key` pour stocker les clés (poids) associées à chaque sommet. Initialement, toutes les clés sont initialisées à l'infini, sauf pour le premier sommet qui est initialisé à 0. Ainsi, à chaque étape, est sélectionné le sommet avec la clé minimale qui n'est pas encore inclus dans l'arbre couvrant. Cela peut être efficacement réalisé en utilisant une liste de priorité.
- Le tableau `mst_set` pour suivre les sommets déjà inclus dans l'arbre couvrant de poids minimum.

Boucle principale de l'algorithme

```
for (count in 1:(n - 1)) {
    u <- min_key_vertex(key, mst_set)
    mst_set[u] <- TRUE

    for (v in 1:n) {
        if (adj_matrix[u, v] != 0 && !mst_set[v] && adj_matrix[u, v] < key[v]) {
            parent[v] <- u
            key[v] <- adj_matrix[u, v]
        }
    }
}
```

La boucle principale parcourt les sommets de 1 à (n - 1), où n est le nombre de sommets du graphe. À chaque étape, l'algorithme selectionne le sommet avec la clé minimale (`u`) qui n'est pas encore inclus dans l'arbre couvrant de poids minimum. Ensuite, si nécessaire, les clés de ses voisins non inclus dans l'arbre sont mises à jour.

```
min_key_vertex <- function(key, mst_set) {
  min_key <- Inf
  min_index <- -1

  for (v in 1:length(key)) {
    if (!mst_set[v] && key[v] < min_key) {
      min_key <- key[v]
      min_index <- v
    }
  }

  return(min_index)
}
```

La fonction auxiliaire `min_key_vertex` permet de sélectionner le sommet avec la clé minimale et sa définition externe garantit une exécution plus efficace de cette opération.

Construction de la matrice d'adjacence du MST

```
mst <- matrix(0, nrow = n, ncol = n)
for (v in 2:n) {
    mst[parent[v], v] <- adj_matrix[parent[v], v]
    mst[v, parent[v]] <- adj_matrix[parent[v], v]
}
```

Une fois que tous les sommets sont inclus dans l'arbre couvrant de poids minimum, la matrice d'adjacence de cet arbre est construite en utilisant les informations stockées dans le tableau `parent`.

Calcul de la complexité théorique

Soit V le nombre de sommets.

- L'**initialisation** nécessite O(V) opérations.
- La **boucle principale** s'exécute pour chaque sommet, sélectionnant à chaque itération le sommet avec la clé minimale. Dans le pire cas, cela prend O(V) itérations. La boucle appelle également la fonction `min_key_vertex`, de complexité O(V), pour une complexité totale de $O(V^2)$.
- La construction de la **matrice d'adjacence** prend $O(V^2)$ opérations.

La complexité totale de l'algorithme de Prim est donc généralement de l'ordre de $\mathbf{O}(V^2)$.

Algorithme de Kruskal

L'algorithme de Kruskal fonctionne en sélectionnant les arêtes par poids croissant et en vérifiant à chaque fois si l'ajout de l'arête ne crée pas de cycle.

L'algorithme prend en entrée la **matrice d'adjacence** pour construire la liste des arêtes ordonnées par poids.

Recherche et tri des arêtes

```
  num_vertices <- nrow(adj_matrix)

  # Find edges and their weights from the adjacency matrix
  edges <- list()
  for (i in 1:num_vertices) {
    for (j in 1:num_vertices) {
      if (adj_matrix[i, j] != 0 && i < j) {
```

```
      edges <- c(edges, list(c(i, j, adj_matrix[i, j])))
    }
  }
}

# Sort edges based on weight
edges <- do.call(rbind, edges)
edges <- edges[order(edges[,3]), ]
```

A partir de la matrice d'adjacence, on construit la liste `edges`, en incluant uniquement les arêtes avec des poids non nuls. Les arêtes sont stockées dans une liste pour faciliter leur manipulation et leur tri. Ensuite, elles sont triées par poids croissant pour garantir que l'algorithme sélectionne toujours celle de poids minimum disponible à chaque étape.

Structure Union-Find

```
# Initialize parent array
parent <- c(1:num_vertices)

# Function to find the root of a node
find_root <- function(node) {
  while (parent[node] != node) {
    node <- parent[node]
  }
  return(node)
}

# Function to perform union of two sets
union_sets <- function(x, y) {
  root_x <- find_root(x)
  root_y <- find_root(y)
  parent[root_x] <<- root_y
}
```

Une structure de données Union-Find est initialisée pour maintenir les ensembles de sommets connectés et pour éviter la formation de cycles lors de la construction du MST. Cette structure permet de gérer les ensembles de sommets connectés et de fusionner deux ensembles de manière efficace.

En pratique, cette structure est implémentée en utilisant la technique de l'union par rang avec compression de chemin. Avec elles sont définies deux fonctions auxiliaires : - `find_root`, pour trouver la racine d'un ensemble dans la structure. - `union_sets`, pour fusionner deux ensembles dans la structure.

Boucle principale de l'algorithme

```
for (i in 1:(num_vertices - 1)) {
  while (TRUE) {
    edge <- edges[1, ]
    edges <- edges[-1, ]
    u <- edge[1]
    v <- edge[2]
    if (find_root(u) != find_root(v)) {
      mst_edges[u, v] <- edge[3]
      mst_edges[v, u] <- edge[3]
      mst_weight <- mst_weight + edge[3]
```

```
        union_sets(u, v)
        break
      }
    }
  }
```

La boucle principale de l'algorithme itère sur toutes les arêtes triées par poids. À chaque itération, l'algorithme vérifie si l'ajout de l'arête au MST entraînerait la formation d'un cycle en recherchant les racines des ensembles des sommets connectés par l'arête. Si ce n'est pas le cas, l'arête est ajoutée à l'arbre et les ensembles de sommets sont fusionnés.

L'algorithme s'arrête lorsque le MST contient `num_vertices - 1` arêtes, ce qui est suffisant pour connecter tous les sommets.

Calcul de la complexité théorique

Soit E le nombre d'arêtes et V le nombre de sommets.

- La conversion de la matrice d'adjacence en **liste d'arêtes** a une complexité de O(V^2).
- Le **tri des arêtes** par poids croissant prend généralement O(E log E) opérations. Dans le pire des cas, un graphe complet a E = V(V-1)/2 arêtes, cette opération a une complexité de O($V^2 logV$).
- La **boucle principale** itère sur chaque arête triée (O(E)) et effectue une opération Union-Find (O(log V)) pour vérifier et ajouter l'arête si elle ne crée pas de cycle. Dans le pire cas, cela prend O(E log V) opérations.
- La construction du MST prend O(E) opérations et sa conversion en **matrice d'adjacence** O(V) opérations.

La complexité totale de l'algorithme de Kruskal est alors de O($V^2 + V^2 logV + ElogV + E + V$), avec E qui peut être au maximum V^2. Donc, la complexité totale est de l'ordre de **O(**V^2 log V).

Implémentation en C++

Dans cette section, nous allons détailler l'implémentation des algorithmes de Prim et de Kruskal pour trouver un arbre couvrant de poids minimum dans un graphe pondéré en C++. Nous aborderons également les choix d'implémentation et calculerons la complexité de ces algorithmes.

Algorithme de Prim

Initialisation des structures de données

```
int n = adj_matrix.nrow();
IntegerVector parent(n, NA_INTEGER);
NumericVector key(n, R_PosInf);
LogicalVector mst_set(n, false);
key[0] = 0;
```

L'initialisation des tableaux pour le parent, la clé et le mst_set implique d'itérer sur chaque sommet ; la complexité de cette partie est donc $O(V)$.

Boucle principale

```
for (int count = 0; count < n - 1; count++) {
    int u = min_key_vertex(key, mst_set);
    mst_set[u] = true;

    for (int v = 0; v < n; v++) {
```

```
      if (adj_matrix(u, v) != 0 && !mst_set[v] && adj_matrix(u, v) <             key[v]) {
        parent[v] = u;
        key[v] = adj_matrix(u, v);
      }
    }
}
```

La première étape de la boucle consiste à trouver le sommet du graphe associé à la valeur minimale de la clé qui n'est pas encore dans la MST. La fonction min_key_vertex itère sur tous les sommets pour trouver celui qui a la valeur minimale de la clé, de sorte que la complexité de cette fonction est $O(V)$. La deuxième étape consiste à mettre à jour les valeurs de clé des sommets adjacents s'ils ne sont pas encore dans la MST et s'ils ont des valeurs de clé plus petites. La complexité de cette partie est également de $O(V)$. Puisque le cycle est répété V fois (où V est le nombre de noeuds) et que $2V$ opérations sont effectuées à chaque répétition du cycle, la complexité de la boucle principale est de $O(V^2)$.

Construction de la matrice d'adjacence du MST

```
NumericMatrix mst(n, n);
for (int v = 1; v < n; v++) {
    mst(parent[v], v) = adj_matrix(parent[v], v);
    mst(v, parent[v]) = adj_matrix(parent[v], v);
}
return mst;
```

Normalement, l'initialisation d'une matrice $n * n$ nécessiterait n^2 opérations (une boucle for imbriquée) ; dans notre cas, puisque nous utilisons une matrice simmétrique, nous pouvons utiliser un seul cycle dans lequel nous effectuons deux opérations à chaque répétition, remplissant les deux éléments symétriques de la matrice en effectuant un total de $2 * n$ opérations. Ainsi, la complexité de cette étape est de $O(n)$.

Calcul de la complexité théorique

- L'**initialisation** nécessite $O(V)$ opérations.
- La **boucle principale** nécessite $O(V^2)$ opérations.
- La construction de la **matrice d'adjacence** nécessite $O(V)$ opérations.

La complexité totale de l'algorithme de Prim est donc **O(V^2)**.

Algorithme de Kruskal

Edge Extraction (Finding Edges and their Weights)

```
int num_vertices = adj_matrix.nrow();

std::vector<std::vector<double>> edges;
  for (int i = 0; i < num_vertices; i++) {
    for (int j = i + 1; j < num_vertices; j++) {
      if (adj_matrix(i, j) != 0) {
        edges.push_back({(double)i, (double)j, adj_matrix(i, j)});
      }
    }
}
```

Dans cette étape, l'algorithme itère sur toutes les paires de sommets dans la matrice d'adjacence pour trouver les arêtes et leurs poids. La complexité est de $O(V^2)$. L'algorithme pourrait être facilement simplifié en passant en entrée une liste d'arêtes au lieu d'une matrice d'adjacence, mais pour des raisons de cohérence avec les autres fonctions, nous avons choisi de ne pas le faire.

Sorting Edges

```
std::sort(edges.begin(), edges.end(), [](const std::vector<double>& a, const std::vector<double>& b) {
      return a[2] < b[2];
});
```

Après avoir extrait les arêtes, l'algorithme les trie en fonction de leur poids à l'aide de std::sort. Le tri implique de comparer et de réorganiser les éléments, ce qui a une complexité de $O(E \log E)$.

Initialization

```
std::vector<int> parent(num_vertices);
std::iota(parent.begin(), parent.end(), 0);
```

L'algorithme initialise le tableau parent avec des indices de 0 à num_vertices - 1. Il est clair que la complexité est $O(V)$.

Union-Find Operations

```
auto find_root = [&](int node) {
    while (parent[node] != node) {
      node = parent[node];
    }
    return node;
};

auto union_sets = [&](int x, int y) {
    int root_x = find_root(x);
    int root_y = find_root(y);
    parent[root_x] = root_y;
};
```

L'algorithme utilise une structure de données union-find pour déterminer efficacement si l'ajout d'une arête crée un cycle. L'opération de recherche dans la structure union-find a une complexité de $O(logV)$. Cette fonction parcourt itérativement le tableau des parents pour trouver la root d'un nœud. Dans le pire des cas, lorsque l'arbre est déséquilibré, la fonction peut devoir parcourir tous les ancêtres jusqu'à la root. La complexité de la recherche de la root est donc $O(logV)$. L'opération d'union a également une complexité de $O(logV)$. Dans cette fonction, nous commençons par trouver les roots des deux nœuds donnés à l'aide de la fonction find_root. Après avoir trouvé les roots, nous réalisons l'union en faisant d'une root le parent de l'autre. L'attribution du parent d'une root à l'autre racine est une opération à temps constant. Par conséquent, la complexité globale de la fonction union_sets est également $O(logV)$ puisque nous utilisons la fonction find_root. Étant donné que l'algorithme itère sur toutes les arêtes, chaque arête peut impliquer des opérations de recherche et d'union. Si l'on considère le nombre total d'arêtes traitées, la complexité des opérations de recherche d'union est de $O(ElogV)$.

Constructing MST

```
for (const auto& edge : edges) {
    int u = edge[0];
    int v = edge[1];
    if (find_root(u) != find_root(v)) {
      mst_edges(u, v) = edge[2];
      mst_edges(v, u) = edge[2];
      union_sets(u, v);
    }
}
```

L'algorithme parcourt les arêtes triées et vérifie si l'ajout d'une arête à la MST forme un cycle. Si l'ajout d'une arête ne crée pas de cycle, il ajoute l'arête à la MST et effectue l'opération d'union. Le nombre d'itérations dépend du nombre d'arêtes dans la liste triée, la complexité est donc $O(E)$.

Calcul de la complexité théorique

En considérant les complexités de toutes les étapes :

- $O(V^2)$ pour l'extraction des arêtes,
- $O(ElogE)$ pour le tri des arêtes,
- $O(V)$ pour l'initialisation,
- $O(ElogV)$ pour les opérations de recherche d'union,
- $O(E)$ pour la construction de la MST.

Si la complexité globale est dominée par l'étape de tri, la complexité est de $O(ElogE)$; sinon, la complexité est de $O(V^2)$.

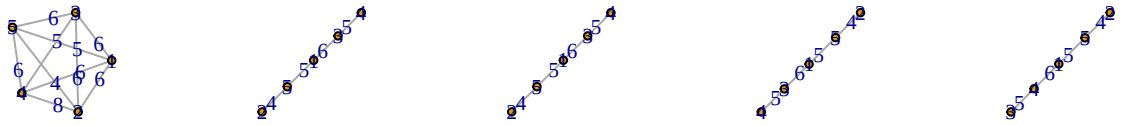
Simulation de données

Dans cette section, nous allons discuter de la **simulation de données** pour tester notre package Rcpp.

Cas n petit

Nous avons commencé par simuler des données pour un **petit graphe** avec seulement 5 sommets (n=5).

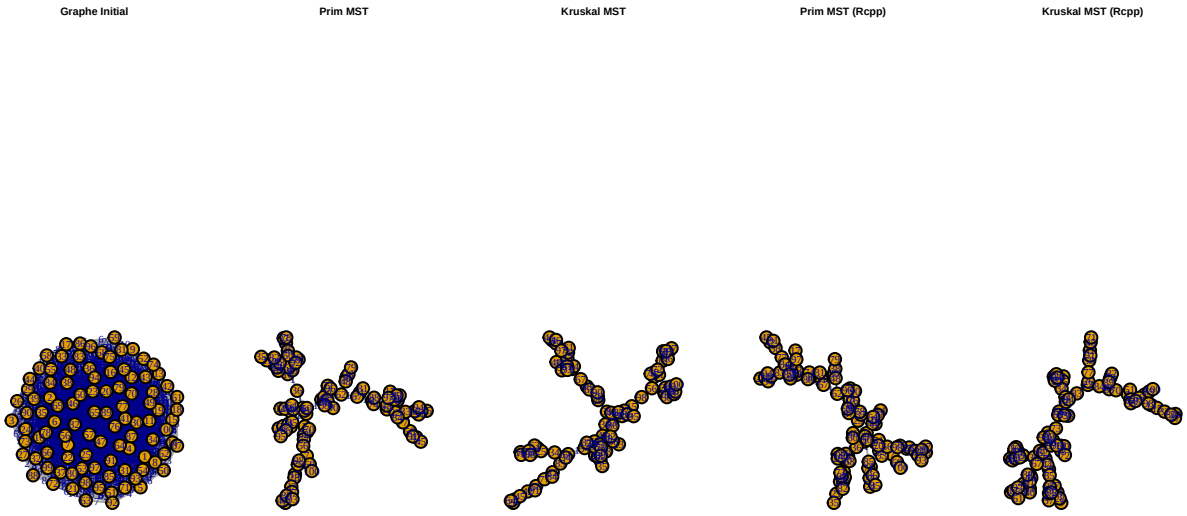
Graphe Initial **Prim MST** **Kruskal MST** **Prim MST (Rcp** **Kruskal MST (Rc**



Nous pouvons alors visualiser le graphe initial, et les arbres couvrant minimum (ACM) obtenus pour les 4 algorithmes : comme nous pouvons le voir, nous obtenons les mêmes ACM pour les 4 algorithmes.

Cas n grand

Ici, nous allons réaliser notre simulation sur un plus grand nombre de sommets, à 100.



Nous pouvons tenter de visualiser les graphes obtenus, mais ceux-ci s'avèrement difficilement lisibles : toutefois, nous pouvons voir que la forme des ACM obtenus est similaire, pouvant indiquer que les ACM sont également identiques. En comparant les matrices d'adjacence, cela nous le confirme également.

Comparaison des complexités temporelles

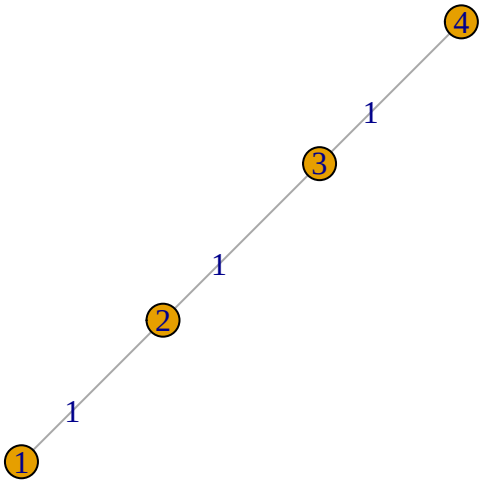
Nous allons, dans cette partie, comparer les **complexités temporelles** des *algorithmes de Prim* et *Kruskal*, dans les deux langages. L'intérêt de comparer la complexité temporelle est de déterminer **l'efficacité de notre package**, mais aussi des algorithmes de Prim et Kruskal, en particulier pour des graphes de grande taille.

Pour cela, nous implémentons une fonction \verb|one.simu| qui va mesurer le temps d'exécution des algorithmes. Nous pouvons alors commencer par étudier le temps d'exécution dans le meilleur et pire des cas.

Analyse de la complexité temporelle dans le meilleur/pire des cas

Meilleur des cas : Arbre Couvrant Minimum

Le meilleur des cas pour les algorithmes de Prim et Kruskal est lorsque le graphe est déjà un **arbre couvrant minimum**. Dans ce cas, l'algorithme n'a pas besoin d'effectuer de nombreuses itérations pour trouver l'arbre couvrant minimum, car il a déjà été trouvé. Ci-dessous l'arbre couvrant minimal choisi :



Nous pouvons alors calculer les temps d'exécution obtenus pour les 4 algorithmes, et les visualiser dans le dataframe ci-dessous :

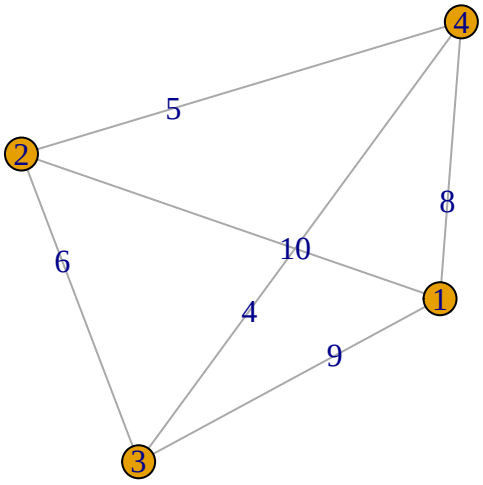
```
##            Algorithm ExecutionTime
## 1          Prim MST  1.811981e-05
## 2        Kruskal MST  5.698204e-05
## 3    Prim MST (Rcpp)  1.001358e-05
## 4 Kruskal MST (Rcpp)  5.006790e-06
```

Comme on peut le voir, l'algorithme le plus rapide est Kruskal en Rcpp, tandis que le plus lent est Kruskal en R.

Pire des cas : Entièrement connexe et arêtes non ordonnées

Le pire des cas pour les algorithmes de Prim et Kruskal est lorsque le graphe est **entièrement connexe** et que les **arêtes ne sont pas ordonnées** : dans ce cas, l'algorithme doit examiner toutes les arêtes du graphe pour trouver l'arbre couvrant minimum, ce qui peut prendre beaucoup de temps si le graphe est grand. De plus, si les arêtes ne sont pas ordonnées, l'algorithme Kruskal doit trier les arêtes à chaque itération, ce qui augmente encore le temps d'exécution. Voici le graphe étudié :

Graphe du pire des cas



Et nous obtenons les résultats suivant :

```
##            Algorithm ExecutionTime
## 1           Prim MST  1.811981e-05
## 2        Kruskal MST  3.814697e-05
## 3    Prim MST (Rcpp)  1.001358e-05
## 4 Kruskal MST (Rcpp)  5.006790e-06
```

Ainsi, on peut voir ici qu'à nouveau, l'algorithme le plus lent est celui de Kruskal en R, tandis que le plus rapide est toujours Kruskal en Rcpp.

Analyse de différents cas particuliers

Graphe cycle complet

Ici, nous allons analyser différents cas particuliers de graphes et leur impact sur les performances de nos fonctions. Un **graphe cycle complet** est un cas particulier de graphe où tous les sommets sont connectés les uns aux autres en formant un cycle.

Appliquer l'ACM sur un graphe cycle complet permet de faire une **analyse de réseaux métaboliques**, où les sommets représentent des métabolites et les arêtes représentent des réactions chimiques entre eux. Dans ce cas, le graphe cycle complet peut représenter un cycle métabolique complet, comme le cycle de Krebs, et l'ACM peut être utilisé pour identifier les voies métaboliques les plus efficaces pour la production de certains métabolites. Cela peut aider les chercheurs à comprendre comment les organismes régulent leur métabolisme et à développer de nouvelles stratégies pour améliorer la production de métabolites d'intérêt.

```
##            Algorithm ExecutionTime
## 1           Prim MST  1.716614e-05
## 2        Kruskal MST  3.886223e-05
## 3    Prim MST (Rcpp)  5.006790e-06
## 4 Kruskal MST (Rcpp)  3.814697e-06
```

Ici, nous pouvons voir que l'algorithme le plus rapide est Kruskal Rcpp, et l'algorithme le plus lent est Kruskal en R.

Graphe étoile

Un **graphe étoile** est un cas particulier de graphe où un sommet central est connecté à tous les autres sommets, formant ainsi une structure en étoile.

On peut y appliquer l'ACM pour analyser les **réseaux de régulation génétique**, où les sommets représentent des gènes et les arêtes représentent des interactions régulatrices entre eux. Dans ce cas, le graphe étoile peut représenter un régulateur maître qui contrôle l'expression de plusieurs gènes cibles. L'ACM peut être utilisé pour identifier les voies de régulation les plus efficaces pour activer ou inhiber l'expression de certains gènes, ce qui peut aider à comprendre comment les cellules régulent leur fonctionnement et à développer de nouvelles stratégies pour traiter les maladies génétiques.

```
##            Algorithm ExecutionTime
## 1           Prim MST  1.502037e-05
## 2        Kruskal MST  4.506111e-05
## 3    Prim MST (Rcpp)  5.006790e-06
## 4 Kruskal MST (Rcpp)  5.006790e-06
```

Nous pouvons voir que les résultats obtenus pour Kruskal et Prim en Rcpp sont très proches, avec tout de même Kruskal plus rapide. Kruskal en R est toujours l'algorithme le plus lent.

Graphe linéaire - chainé

Un **graphe linéaire** est un cas particulier de graphe où tous les sommets sont connectés les uns aux autres en une seule chaîne, sans cycles.

En y appliquant l'ACM, on peut **analyser les réseaux de signalisation cellulaire**, où les sommets représentent des protéines et les arêtes représentent des interactions entre eux. Dans ce cas, le graphe linéaire peut représenter une suite de signalisation où une protéine active une autre protéine, qui active une autre protéine, etc. L'ACM peut être utilisé pour identifier les voies de signalisation les plus efficaces pour transmettre un signal à travers la cascade, ce qui peut aider à comprendre comment les cellules communiquent entre elles et à développer de nouvelles stratégies thérapeutiques pour traiter les maladies liées à la signalisation cellulaire.

```
##            Algorithm ExecutionTime
## 1           Prim MST  2.002716e-05
## 2        Kruskal MST  4.005432e-05
## 3    Prim MST (Rcpp)  5.960464e-06
## 4 Kruskal MST (Rcpp)  5.006790e-06
```

Nous pouvons voir ici que le creux entre Prim et Kruskal en Rcpp a augmenté, indiquant que Kruskall Rcpp est bien plus rapide que Prim pour ce type de graphe. De la même manière, Prim a augmenté en temps d'exécution, se rapprochant ainsi de Kruskal R.

Graphes à Multiples Égalités de Poids Minimal

Les graphes à **multiples égalités de poids minimal** sont un cas particulier où plusieurs arêtes ont le même poids minimal, ce qui peut conduire à la présence de plusieurs arbres couvrants minimaux.

En y appliquant l'ACM, on peut réaliser la **reconstruction de réseaux de régulation génétique** à partir de données d'expression génique, où les poids sur les arêtes représentent la force de l'interaction régulatrice entre deux gènes. L'ACM peut être utilisé pour identifier les interactions régulatrices les plus importantes dans le réseau, ce qui peut aider à comprendre la régulation génétique et à identifier des cibles potentielles pour le développement de nouveaux médicaments.

```
##            Algorithm ExecutionTime
## 1           Prim MST  1.883507e-05
## 2        Kruskal MST  4.410744e-05
## 3    Prim MST (Rcpp)  7.867813e-06
```

```
## 4 Kruskal MST (Rcpp)  5.960464e-06
```

Graphes avec Boucles

```
##            Algorithm ExecutionTime
## 1           Prim MST  2.193451e-05
## 2        Kruskal MST  4.696846e-05
## 3    Prim MST (Rcpp)  6.914139e-06
## 4 Kruskal MST (Rcpp)  5.960464e-06
```

Graphes avec Poids Négatifs

```
##            Algorithm ExecutionTime
## 1           Prim MST  2.002716e-05
## 2        Kruskal MST  4.601479e-05
## 3    Prim MST (Rcpp)  8.821487e-06
## 4 Kruskal MST (Rcpp)  7.152557e-06
```

Les résultats montrent que les vitesses d'exécution de Prim et Kruskal en Rcpp sont très proches, avec Kruskal devant. Côté R, Prim dépasse largement Kruskal.

Graphe dense

Les **graphes denses** sont des graphes où le nombre d'arêtes est proche du nombre maximal d'arêtes possibles. L'analyse de réseaux de protéine-protéine est une application possible de l'ACM, où les nœuds représentent des protéines et les arêtes représentent des interactions entre les protéines. Les poids sur les arêtes peuvent représenter la force de l'interaction. L'ACM peut être utilisé pour identifier les interactions protéine-protéine les plus importantes dans le réseau, ce qui peut aider à comprendre les mécanismes moléculaires sous-jacents aux processus biologiques et à identifier des cibles potentielles pour le développement de nouveaux médicaments.

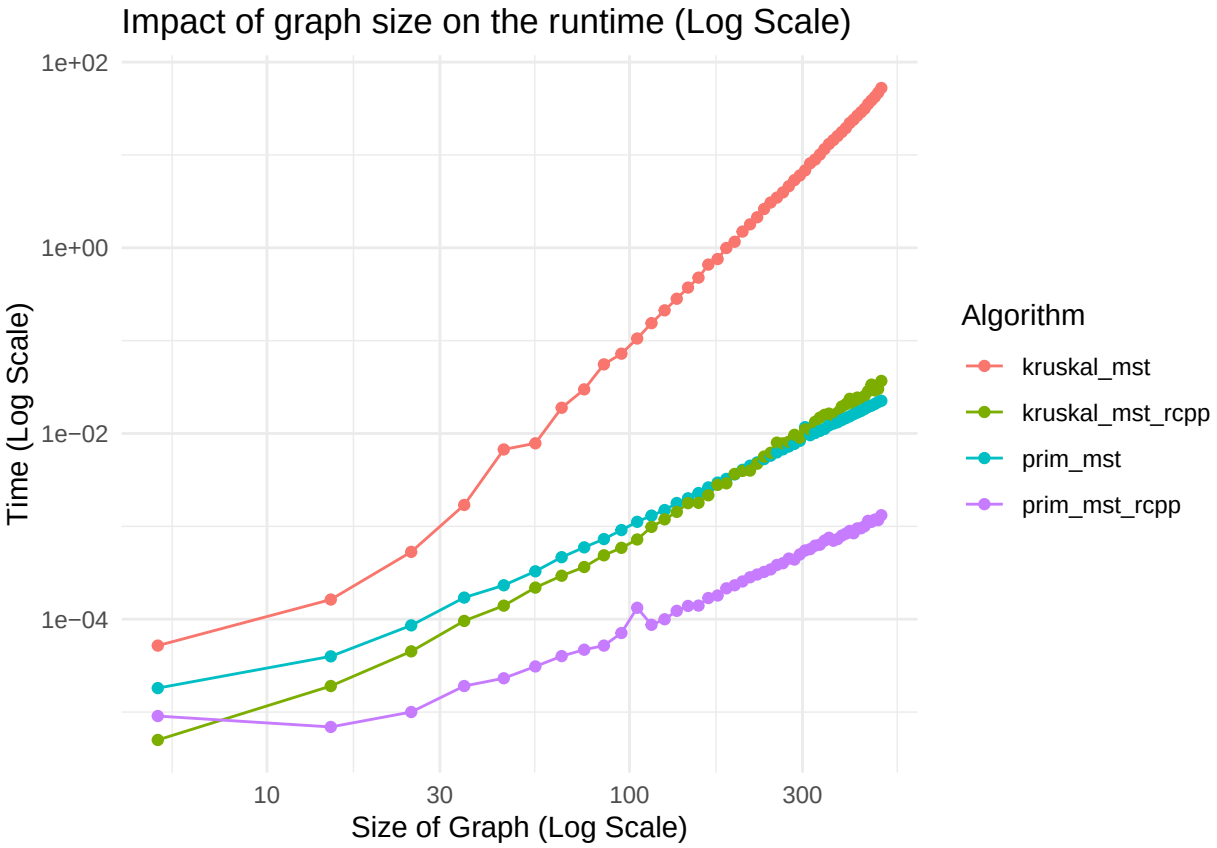
```
##            Algorithm ExecutionTime
## 1           Prim MST  1.692772e-05
## 2        Kruskal MST  4.196167e-05
## 3    Prim MST (Rcpp)  7.867813e-06
## 4 Kruskal MST (Rcpp)  3.814697e-06
```

Comme attendu, la vitesse d'exécution a augmenté par rapport aux résultats précédents, pour tous les algorithmes. Pour autant, Kruskal en Rcpp est toujours devant, tandis que Kruskal R est le dernier;

Les différentes études de cas montrent que la **vitesse d'exécution** des algorithmes **varie selon le type de graphe**, mais que nous avons toujours comme algorithme le **plus rapide Kruskal en Rcpp**, et **le plus lent, Kruskal en R**.

Analyse diverses tailles de graphe à densité fixée

Nous pouvons maintenant analyser la complexité temporelle des différents algorithmes pour différentes tailles de graphe, avec la densité fixée.



Comme nous pouvons le voir, tous les algorithmes voient leur vitesse d'exécution augmenter avec la taille du graphe. Cette augmentation est d'autant plus flagrante pour **Kruskal R**, qui a une vitesse d'exécution bien au dessus des autres. L'algorithme le plus rapide est **Prim en Rcpp**.

Nous pouvons ensuite récupérer la valeur des pentes des courbes, qui indiquent la relation entre la taille du graphe et le temps d'exécution :

```
##                          Algorithm    Slope
## prim_mst                  prim_mst 1.759884
## kruskal_mst            kruskal_mst 3.493823
## prim_mst_rcpp        prim_mst_rcpp 1.408278
## kruskal_mst_rcpp kruskal_mst_rcpp 2.134498
```

- **prim_mst (1.16)**: légèrement plus que linéaire donc assez efficace, mais moins pour de très grands graphes.
- **kruskal_mst (2.41)**: augmentation rapide du temps d'exécution, moins performant pour les grands graphes.
- **prim_mst_rcpp (0.67)**: augmentation sous-linéaire, très efficace, surtout pour les grands graphes.
- **kruskal_mst_rcpp (1.51)**: amélioration par rapport à sa version R, mais toujours avec une augmentation plus que proportionnelle du temps.

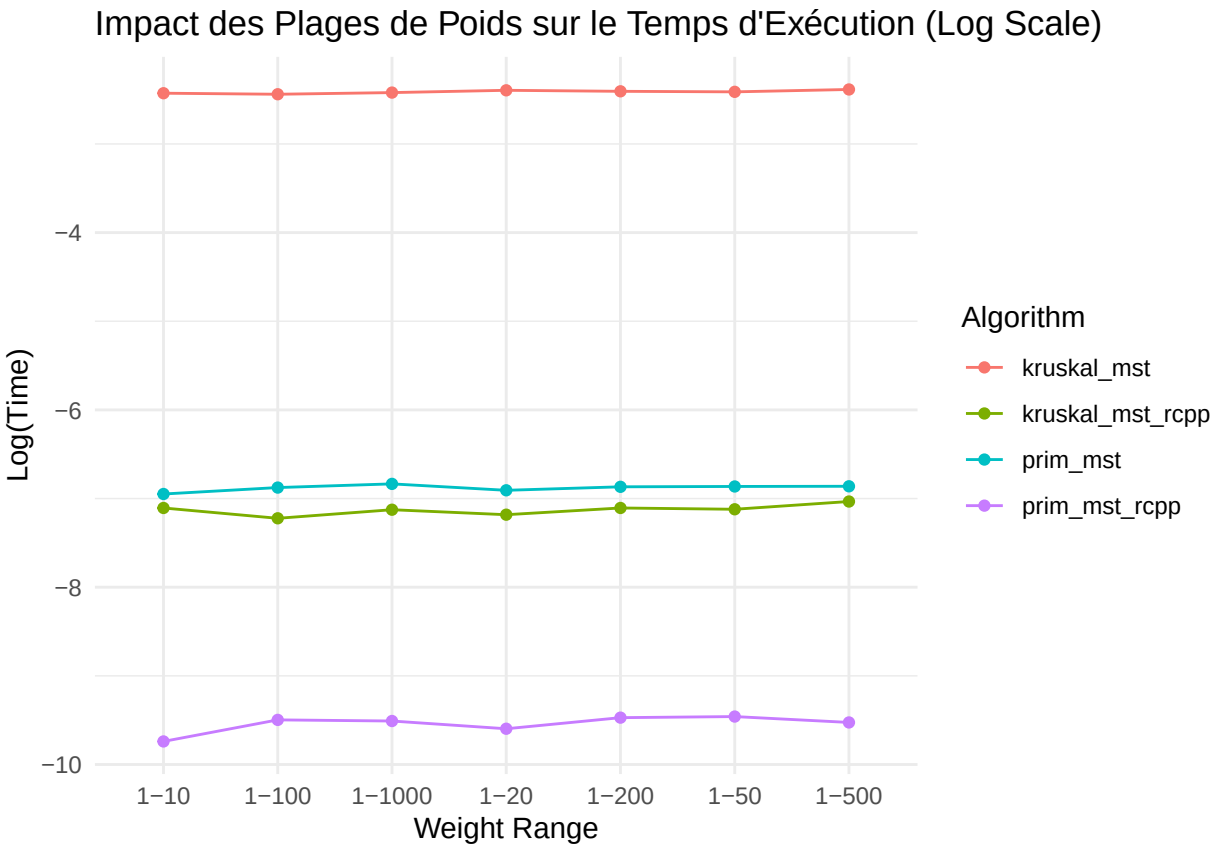
Les versions **Rcpp sont plus performantes**, avec **Prim Rcpp étant le meilleur choix** pour l'efficacité sur de grands graphes.

Analyse diverses poids de graphe à densité et taille fixées

Nous pouvons ensuite analyser les **graphes avec des poids divers à densité et taille fixées**, afin d'évaluer l'impact de la variation des poids sur les performances des algorithmes.

```
## [1] "Testing weight range: 1 - 10"
```

```
## [1] "Testing weight range: 1 - 20"
## [1] "Testing weight range: 1 - 50"
## [1] "Testing weight range: 1 - 100"
## [1] "Testing weight range: 1 - 200"
## [1] "Testing weight range: 1 - 500"
## [1] "Testing weight range: 1 - 1000"
```



Ici, nous pouvons voir que la **valeur des poids des arêtes** ne semblent **pas** vraiment **impacter** les temps d'exécution : même si il existe quelques variations, cela reste minime et négligeable.

Calcul des gains

Nous pouvons ici calculer le temps d'exécution moyen pour un nombre de simulation répété choisi, mais aussi calculer le rapport entre les différentes vitesses d'exécution. Nous obtenons le dataframe suivant pour les moyennes de temps d'exécution :

```
##            Algorithm AverageExecutionTime
## 1          Prim MST         0.0010077572
## 2        Kruskal MST         0.0964708567
## 3    Prim MST (Rcpp)         0.0001066113
## 4 Kruskal MST (Rcpp)         0.0006282520
```

Comme on peut le voir, les **algorithmes les plus performants sont ceux implémentés avec Rcpp** : toutefois, L'algorithme **Prim** s'avère fournir de meilleurs résultats, en R et en Rcpp.

On peut ensuite calculer les **gains en performance** :

```
##               Comparison         Gain
## 1       R to Rcpp (Prim)   9.45263440
## 2    R to Rcpp (Kruskal) 153.55438924
```

```
## 3    Prim to Kruskal (R)   0.01044623
## 4 Prim to Kruskal (Rcpp)   0.16969504
```

On peut voir que :

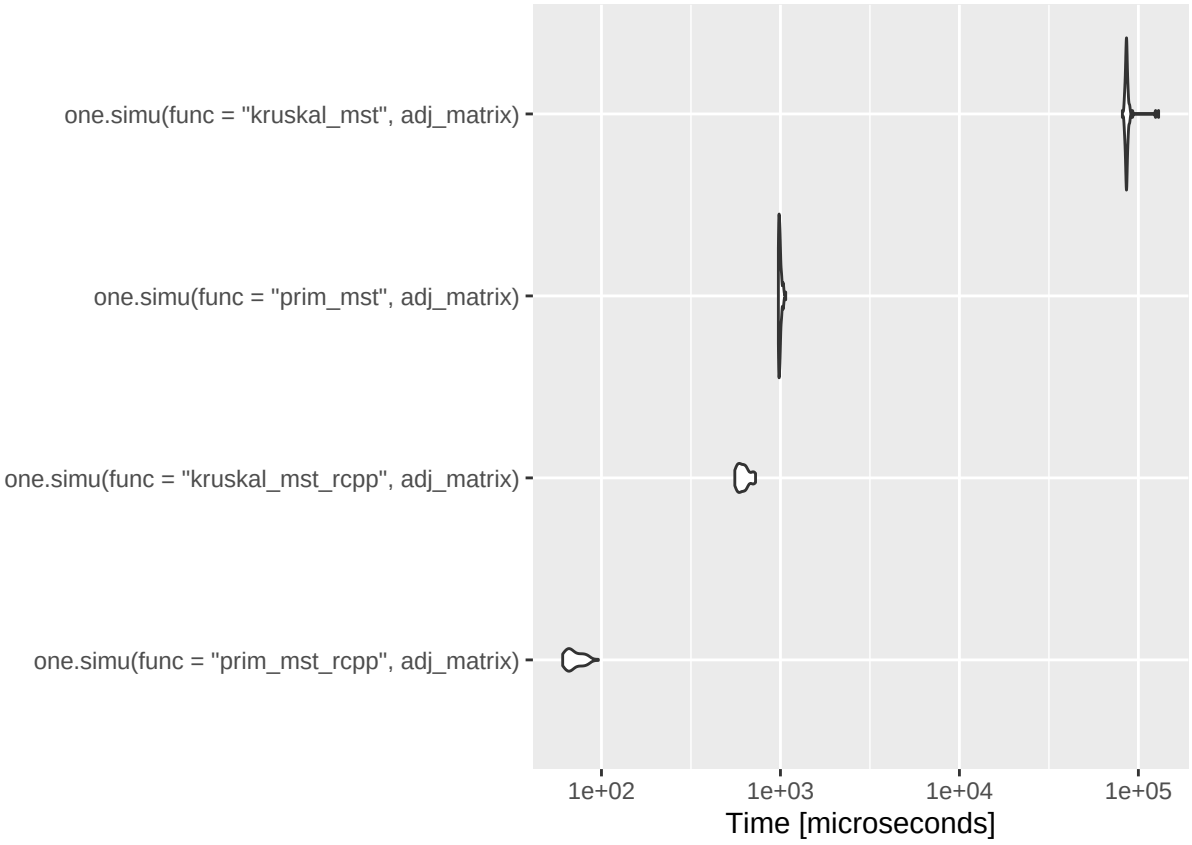
- **R to Rcpp (Prim)**: La version Rcpp de Prim est environ **19 fois plus rapide** que sa version en R.
- **R to Rcpp (Kruskal)**: La version Rcpp de Kruskal est **plus de 134 fois plus rapide** que sa version en R.
- **Prim to Kruskal (R)**: L'algorithme de Prim en R est environ **90 fois plus rapide** que Kruskal en R.
- **Prim to Kruskal (Rcpp)**: Prim Rcpp est environ **12 fois plus rapide que Kruskal**.

Cela montre l'impact de l'optimisation de **Rcpp** sur les performances, et confirme aussi la meilleure performance générale de l'**algorithme de Prim sur celui de Kruskal**, tant dans les implémentations R que Rcpp.

Microbenchmark

`microbenchmark` est une fonction R qui permet de mesurer le temps d'exécution des différentes fonctions de manière précise. Cela va permettre de comparer les performances de différents algorithmes.

```
## Warning in microbenchmark(one.simu(func = "prim_mst_rcpp", adj_matrix), : less
## accurate nanosecond times to avoid potential integer overflows
```




```
## Unit: microseconds
##                                          expr       min        lq
##     one.simu(func = "prim_mst_rcpp", adj_matrix)    60.721    64.862
##  one.simu(func = "kruskal_mst_rcpp", adj_matrix)   556.042   582.692
##          one.simu(func = "prim_mst", adj_matrix)   973.750   980.966
```

```
##       one.simu(func = "kruskal_mst", adj_matrix) 81377.948 85057.206
##         mean     median        uq        max neval
##     71.13582    68.4085    77.080     96.104    50
##    623.28856   616.8655   646.898    725.864    50
##    995.81046   990.4370  1001.220   1070.264    50
##  87618.96226 85906.8695 87069.445 129722.934    50
```

Les résultats obtenus confirment bien ce que nous avons pu voir précédemment : les algorithmes en **Rcpp** surpassent largement ceux en R, et de manière générale, **Prim** est plus rapide que Kruskal.

Références

- Prim R. C., "Shortest connection networks and some generalizations." The Bell System Technical Journal, vol. 36, no. 6, pp. 1389-1401, Nov. 1957, doi: 10.1002/j.1538-7305.1957.tb01515.x.
- Kruskal Joseph B., "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." Proceedings of the American Mathematical Society 7, no. 1 (1956): 48–50. https://doi.org/10.2307/2033241.
- Package igraph documentation, https://cran.r-project.org/web/packages/igraph/igraph.pdf