

UNIVERSITÉ DE SHERBROOKE

IFT-436

ALGORITHMES ET STRUCTURES DE DONNÉES

---

## Devoir 5

---

*Par :*

François BÉLANGER 94 245 437

Jérémie COULOMBE 13 061 991

Geneviève DOSTIE 12 078 306

*Présenté à :*

Richard ST-DENIS

23 juillet 2015

## Introduction

Pour ce dernier travail pratique du cours, il était question d'effectuer une étude de performance de différents algorithmes s'attaquant au même problème. Les équipes étaient libres de choisir un problème de leur choix et de réunir un minimum de trois algorithmes différents pour résoudre celui-ci. Dans le cas présent, le calcul de l'arbre sous-tendant de coût minimal d'un graphe non-orienté connexe valué à été sélectionné. Trois algorithmes classiques – ceux de Borůvka, de Kruskal et de Prim – ont été retenus pour l'étude. Ceux-ci ont été codés et exécutés avec Python.

Dans ce rapport seront présentés les algorithmes choisis, incluant les hypothèses concernant leurs complexités, les outils de travail utilisés, tant pour partager les sources que pour les développer, la conception du générateur d'échantillons aléatoires ainsi que les résultats de l'étude.

## Outils de travail

Pour ce travail, le langage choisi a été Python, sous sa version 2.7. Bien qu'il ne figure pas dans les palmarès des langages les plus performants, l'expérience a été tentée, permettant ainsi pour la plupart des coéquipiers d'en apprendre davantage sur ce langage. De plus, il semblait plus simple – du moins, c'est ce qui était prévu ! – de trouver de bons algorithmes sur internet pour se concentrer plutôt sur les tests et l'analyse des résultats. Hélas, la réécriture de deux des trois algorithmes a été nécessaire. Les tests ont été roulés sur les ordinateurs du D4-1023.

Pour programmer en Python, l'environnement de développement choisi a été PyCharm 4.5.3. PyCharm comporte un outil Git qui a été utile pour le travail ; en effet, le client web GitHub fut utilisé pour le partage des sources, la distribution de tâches et pour la consultation de l'historique des changements pour les cas de nécessité de retour en arrière.

## Nos algorithmes

Le choix des algorithmes a été orienté par la liste des méthodes de calcul de l'arbre sous-tendant à coût minimal d'un graphe présentée en classe, ainsi que par la liste des propositions dans l'énoncé du travail pour le troisième algorithme. Les algorithmes développés par Borůvka, Kruskal et Prim sont tous des stratégies gloutonnes, mais ne partagent pas tout à fait les mêmes complexités algorithmiques.

L'algorithme de Borůvka n'a pas été trouvé sur internet en langage Python, donc une version dérivée du pseudo-code présenté sur la page Wikipédia « Borůvka algorithm » a été programmée. L'algorithme de Kruskal a été tiré et adapté du dépôt GitHub de l'utilisateur « israelst », qui indique l'avoir adapté du livre « Algorithms » de Dasgupta, Papadimitriou et Vazirani. Aucun algorithme efficace n'ayant été trouvé pour la méthode de Prim, une adaptation Python du pseudo-code présenté dans les notes du présent cours a été préconisée. Finalement, une classe représentant un ensemble d'ensembles disjoints a été créée pour unifier le fonctionnement des algorithmes de Borůvka et de Kruskal. Le code a été tiré des sources originales de l'algorithme de Kruskal trouvé sur internet.

## Hypothèses

La complexité des algorithmes devrait être les suivantes (pour un  $m$  signifiant le nombre d'arêtes du graphe et  $n$  le nombre de noeuds) :  $O(m \log n)$  pour Borůvka,  $O(m \log m)$  pour Kruskal et  $O(m \log n)$  pour Prim. Il est donc à comprendre qu'étant de même complexité, Borůvka et Prim auront des courbes de temps d'exécution de comportement similaire. Pour ce qui est de Kruskal, il devrait être le moins performant des trois algorithmes, car son  $\log$  s'applique sur le nombre d'arêtes de non sur le nombre de noeuds.

### Générateur d'échantillons

Cette dernière affirmation est vraie, puisque c'est que le générateur utilisé pour créer les ensembles de données de l'étude a été paramétré pour générer des graphes ayant exactement 10 fois plus d'arêtes que de noeuds. Il fonctionne de la façon suivante : on relie  $n$  noeuds de façon linéaire par des arêtes de poids aléatoires entre 1 et 100 puis on génère des arêtes de même distribution de poids entre des paires noeuds non-reliés choisis aléatoirement jusqu'à atteindre un nombre d'arêtes égal à 10 fois le nombre de noeuds.

### Générateur de nombres aléatoires

Avant de rouler des tests exigeants, seul le générateur de nombre aléatoire de la librairie standard de Python était utilisée. Celui-ci est basé sur une implémentation en C du *Marsenne Twister*<sup>1</sup>, un algorithme avec une période de  $2^{19937} - 1$  fortement testé depuis plusieurs années. Le générateur de graphe utilisait la fonction *randint* de Python, qui est une application de *random*, qui utilise le *Marsenne Twister*. À la suite d'un profilage des temps d'exécution des tests, un ralentissement majeur a été relevé au niveau de la génération des données aléatoires. Ainsi, un changement vers la librairie *numpy* a été fait pour la plupart des fonctions de génération de nombres aléatoires. Une seule fonction de la librairie standard a été conservée dans « framework.py » puisqu'elle n'avait pas d'équivalent dans la librairie *numpy* (« sample », qui se charge d'échantillonner le tableau des tailles de données possibles). Cette librairie externe se base aussi sur le *Marsenne Twister*.

### Méthodologie

Le script principal, «framework.py», prends 4 paramètres en entrée :

- $n$  : La taille minimum des données à générer (le nombre de noeuds du graphe)
- $N$  : La taille maximum des données à générer
- $s$  : Le nombre d'échantillon de tailles de données à tester entre  $n$  et  $N$  (on en choisi  $s - 2$  de façon aléatoire, car on inclus toujours  $n$  et  $N$  dans les tests)
- $r$  : Le nombre de répétition des tests pour une taille donnée

Ainsi,  $3 * s * r$  tests sont exécutés au total, soit un pour chaque répétition de chaque échantillon de données choisi entre  $n$  et  $N$  pour chacun des trois algorithmes.

Une session de test se déroule ainsi :

- 1) Initialiser le germe du générateur de nombres aléatoires
- 2) Sauvegarder l'état du générateur aléatoire
- 3) Choisir aléatoirement  $s - 2$  tailles des données à tester entre  $n$  et  $N$ , exclus
- 4) Pour chaque algorithme
  - (a) Charger l'état du générateur
  - (b) Pour chaque taille de données
    - i. Prendre une mesure du temps de départ
    - ii. Pour chaque répétition du test ( $r$  fois)
      - A. Générer un graphe de la bonne taille
      - B. Exécuter l'algorithme

---

1. [https://en.wikipedia.org/wiki/Mersenne\\_Twister](https://en.wikipedia.org/wiki/Mersenne_Twister)

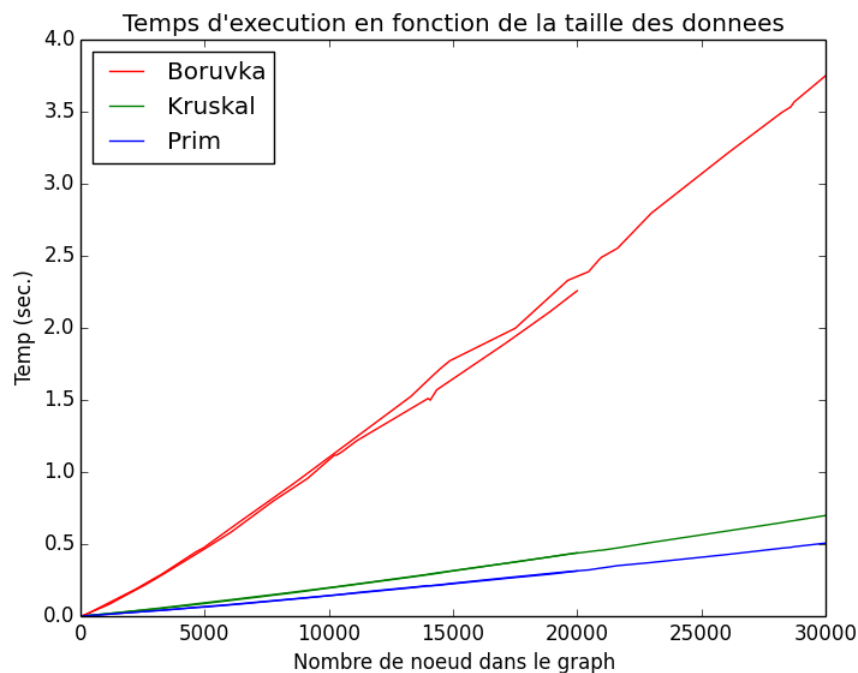
- iii. Prendre la mesure du temps total d'exécution pour cette taille de donnée pour cet algorithme
- 5) Charger l'état du générateur
- 6) Pour chaque taille de données
  - (a) Prendre une mesure du temps de départ
  - (b) Pour chaque répétition du test
    - i. Générer un graphe de la bonne taille
  - (c) Prendre la mesure du temps total d'exécution pour cette taille de donnée
  - (d) Pour chaque algorithme
    - i. Soustraire ce temps du temps d'exécution officiel
- 7) Diviser tous les temps d'exécution par le nombre de répétition des tests
- 8) Sauvegarder les données sur disque

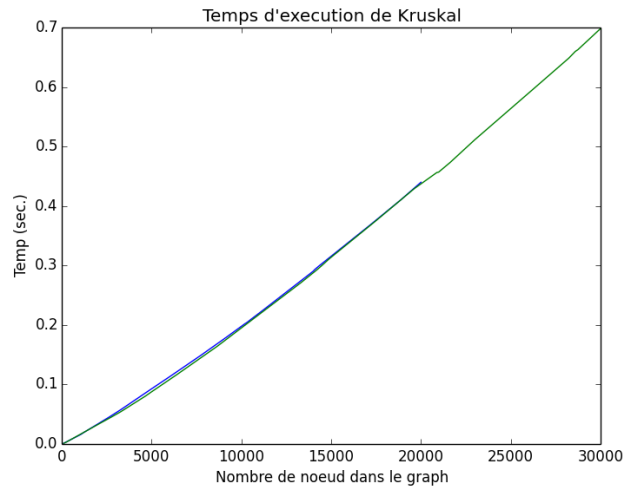
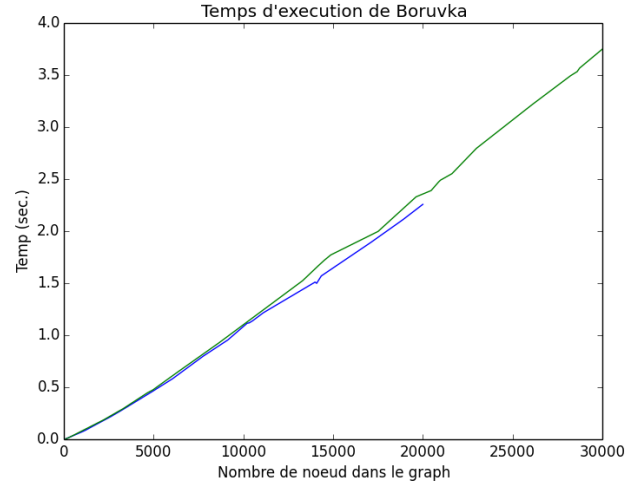
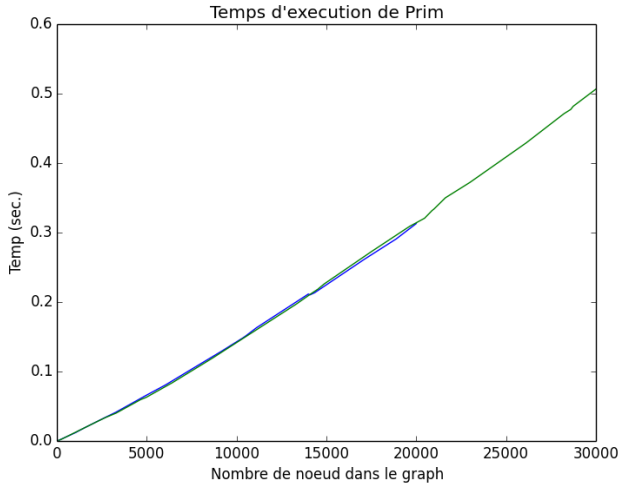
Pour recueillir les données sur le temps d'exécution versus le nombre de sommets des graphes, la librairie *matplotlib* a été importée. Celle-ci permet de tracer des courbes à partir des données brutes recueillies, évitant ainsi la nécessité d'une compilation des résultats dans un fichier pour ensuite produire des graphiques avec un logiciel externe comme Excel.

## Résultats

Les résultats de deux sessions de tests ont été compilé et jumelés dans les mêmes graphiques :

- *pythonframework.py – n100 – N20000 – s20 – r100* : Le graphe de taille minimal avait 100 noeuds, le max en avait 20000, et 18 autres tailles ont été choisies aléatoirement. Pour chaque taille, 100 graphes différents ont été testé et les résultats moyens se retrouvent dans les courbes.
- *pythonframework.py – n100 – N30000 – s25 – r100* : La taille maximale des graphes a été étendue à 30000, et 5 échantillons supplémentaires ont été ajoutés.





### Analyse des résultats

La première chose qui saute aux yeux en voyant ces courbes est la grande variabilité du temps d'exécution de l'algorithme de Borůvka. En effet, même si 100 tests différents sont exécuté pour chaque point de la courbe, un aspect irrégulier se manifeste a partir des graphes de tailles 12000 environ. De plus, les résultats pour les deux sessions de tests sont relativement différentes pour ce même algorithme de Borůvka, alors que pour les autres les courbes sont lisses comme on pourrait s'y attendre. Cela peut être expliqué en partie par le fait que son implémentation diffère assez du pseudo-code original et qu'il pourrait y avoir quelques optimisations possibles. Il reste qu'en général cet algorithme prends relativement plus de temps que les autres.

Les hypothèses évoquées précédemment semble être confirmées par les résultats obtenus. Les trois courbes affichent une progression d'ordre  $n \log n$ . Il était annoncé que Kruskal ait une plus forte pente que Prim, ce qui semble être le cas ; une amélioration qui aurait pu être fait aurait été de tester des données d'encores plus grand taille pour voir les tendances à plus long terme pour justifier cette affirmation.

## Annexes

### boruvka.py

```
__author__ = 'Francois_Belanger_94_245_437' \
             'Genevieve_Dostie_12_078_306' \
             'Jeremie_Coulombe_13_061_991'

from disjoint_set import DisjointSet

# adapted from https://en.wikipedia.org/wiki/Bor%C3%AFvka%27s_algorithm)

def boruvka(adj_list):
    disj_set = DisjointSet()

    for u in adj_list.keys():
        disj_set.make_set(u)

    min_span_tree = []
    while True:
        minima = {}
        for u in adj_list.keys():
            root = disj_set.find(u)
            for v in adj_list[u]:
                if disj_set.find(v) != root and (root not in minima or adj_list[u][v] < minima[root][0]):
                    minima[root] = (adj_list[u][v], u, v)

        if len(minima) == 0:
            break

        for edge in minima.items():
            if disj_set.union(edge[0], edge[1][2]):
                min_span_tree.append(edge[1])

    return min_span_tree
```

### kruskal.py

```
__author__ = 'Francois_Belanger_94_245_437' \
             'Genevieve_Dostie_12_078_306' \
             'Jeremie_Coulombe_13_061_991'

from disjoint_set import DisjointSet

# adapted from https://github.com/israelst/Algorithms-Book-Python/blob/master/5-Greedy-algorithms/kruskal.py

def kruskal(args):
    vertex_list, edge_list = args
    disj_set = DisjointSet()
    min_span_tree = []

    for u in vertex_list:
        disj_set.make_set(u)

    edges = list(edge_list)
    edges.sort()

    for edge in edges:
        weight, u, v = edge
        if disj_set.find(u) != disj_set.find(v):
            disj_set.union(u, v)
            min_span_tree.append(edge)
```

```
return min_span_tree
```

## prim.py

```
__author__ = 'Francois_Belanger_94_245_437' \
             'Genevieve_Dostie_12_078_306' \
             'Jeremie_Coulombe_13_061_991'
```

```
import sys
from heapq import heappop, heappush
```

*# adapted from Algorithmes et structures de donnees, IFT436, Chap. 3, page 34, Richard St-Denis*

```
def prim(adj_list):
    queue = []
    costs = []
    parent = []

    for v in adj_list.keys():
        c = 0 if v is 0 else sys.maxint
        queue.append((c, v))
        costs.append(c)
        parent.append(None)

    while len(queue) > 0:
        best = heappop(queue)
        cost, u = best
        if cost is sys.maxint:
            break
        costs[u] = None
        for edge in adj_list[u].items():
            v, weight = edge
            if costs[v] is not None and weight < costs[v]:
                parent[v] = u
                costs[v] = weight
                heappush(queue, (weight, v))

    return parent
```

## disjoint\_set

*# adapted from <https://github.com/israelst/Algorithms-Book-Python/blob/master/5-Greedy-algorithms/kruskal.py>*

```
class DisjointSet:

    def __init__(self):
        self.parent = {}
        self.rank = {}

    def make_set(self, vertex):
        self.parent[vertex] = vertex
        self.rank[vertex] = 0

    def find(self, vertex):
        parent = self.parent[vertex]
        return vertex if vertex == parent else self.find(parent)

    def union(self, vertex1, vertex2):
        root1 = self.find(vertex1)
        root2 = self.find(vertex2)
        if root1 == root2:
```

```

        return False

    if self.rank[root1] > self.rank[root2]:
        self.parent[root2] = root1
    else:
        self.parent[root1] = root2
        if self.rank[root1] == self.rank[root2]:
            self.rank[root2] += 1

    return True

```

## graph\_generator.py

```

__author__ = 'Francois_Belanger_94_245_437' \
             'Genevieve_Dostie_12_078_306' \
             'Jeremie_Coulombe_13_061_991'

import numpy as np

# Maximum weight of an edge
MAX_WEIGHT = 100

# Average degree of vertices
AVG_DEG = 10

def generate_graph(n):
    # Graph structures that must be generated in parallel
    adj_list = {}
    edges = []

    # Initialize adjacency lists for each vertex
    for u in xrange(0, n):
        adj_list[u] = {}

    # Connect each vertex to its "next" vertex to make the graph connected
    for u in xrange(0, n-1):
        weight = np.random.randint(1, MAX_WEIGHT)
        adj_list[u][u+1] = adj_list[u+1][u] = weight
        edges.append((weight, u, u+1))

    # Add random edges until graph is of correct size
    num_edges = n * min(0.2*(n-1), AVG_DEG)
    while len(edges) < num_edges:
        # Generate a random edge
        u = np.random.randint(0, n-1)
        v = np.random.randint(0, n-1)

        # Skip edge for any of these conditions:
        # - It is linking a vertex with itself
        # - It is linking a vertex with a vertex that has a "lower" value (to keep the graph undirected)
        # - It is linking a vertex with the "next" one
        # - It was already added
        if u > v - 2 or v in adj_list[u]:
            continue

        # Add edge
        weight = np.random.randint(1, MAX_WEIGHT)
        adj_list[u][v] = adj_list[v][u] = weight
        edges.append((weight, u, v))

    return adj_list, (range(0, n), edges)

if __name__ == "__main__":
    generate_graph(10000)

```



## framework.py

```
__author__ = 'Francois_Belanger_94_245_437' \
             'Genevieve_Dostie_12_078_306' \
             'Jeremie_Coulombe_13_061_991'

import argparse
import random
import time
import timeit

import numpy as np

from graph_generator import *
from boruvka import *
from kruskal import *
from prim import *

nb_algo = 3

def parm():
    parser = argparse.ArgumentParser(description="put_something_here")
    parser.add_argument('--n_min', '-n', type=int, default=100, help="Min_data_size")
    parser.add_argument('--n_max', '-N', type=int, default=10000, help="Max_data_size")
    parser.add_argument('--nb_sample', '-s', type=int, default=100, help="Number_of_random_samples")
    parser.add_argument('--nb_repetition', '-r', type=int, default=1000, help="Number_of_test_at_each_data_size")
    args = parser.parse_args()

    return args

def test(args):
    # initializing pseudo random generator
    np.random.seed(0)
    random_state = np.random.get_state()

    step_sizes = random.sample(xrange(args.n_min+1, args.n_max), args.nb_sample-2)
    step_sizes.append(args.n_min)
    step_sizes.append(args.n_max)
    step_sizes.sort()

    timed = np.zeros((nb_algo, len(step_sizes)))

    fct = [boruvka, kruskal, prim]

    for algo_idx in range(nb_algo):
        np.random.set_state(random_state)

        for sz_idx in range(len(step_sizes)):
            start_time = timeit.default_timer()
            for test_iter in xrange(0, args.nb_repetition):
                graph = generate_graph(step_sizes[sz_idx])
                #execute the algo on the graph
                fct[algo_idx](graph[algo_idx % 2])

            #timer stop
            timed[algo_idx][sz_idx] = timeit.default_timer() - start_time

    np.random.set_state(random_state)
    for sz_idx in range(len(step_sizes)):
        start_time = timeit.default_timer()

        for test_iter in xrange(0, args.nb_repetition):
            graph = generate_graph(step_sizes[sz_idx])
```

```

        delta = timeit.default_timer() - start_time

        for i in range(nb_algo):
            timed[i][sz_idx] -= delta

    timed /= args.nb_repetition

    stamp = str(time.time())

    np.save('data_'+stamp, timed)
    np.save('step_'+stamp, step_sizes)

if __name__ == "__main__":
    arg = parm()
    test(arg)

```

## plotter.py

```

__author__ = 'Francois_Belanger_94_245_437' \
             'Genevieve_Dostie_12_078_306' \
             'Jeremie_Coulombe_13_061_991'

import numpy as np
import matplotlib.pyplot as plt

nb_algo = 3
algo = ['Boruvka', 'Kruskal', 'Prim']

def plot_graph(x1, y1, x2, y2, idx):
    fig, ax = plt.subplots( nrows=1, ncols=1 )
    ax.plot(y1, x1[idx])
    ax.plot(y1, x1[idx])
    plt.title("Temps_d'execution_de_" + algo[idx])
    plt.ylabel("Temp_(sec.)")
    plt.xlabel("Nombre_de_noeud_dans_le_graph")
    fig.savefig(algo[idx]+' .png')
    plt.close(fig)

def plot_bo(x1, y1, x2, y2):
    plot_graph(x1, y1, x2, y2, 0)

def plot_kr(x1, y1, x2, y2):
    plot_graph(x1, y1, x2, y2, 1)

def plot_pr(x1, y1, x2, y2):
    plot_graph(x1, y1, x2, y2, 2)

def plot_group_graph(timed, step_sizes):
    color = ['r', 'g', 'b']

    for i in range(nb_algo):
        plt.plot(step_sizes, timed[i], color[i])

def plot_total(x1, y1, x2, y2):
    plot_group_graph(x1, y1)
    plot_group_graph(x2, y2)

```

```

plt.title("Temps_d'execution_en_fonction_de_la_taille_des_donnees")
plt.ylabel("Temp_(sec.)")
plt.xlabel("Nombre_de_noeud_dans_le_graph")
plt.legend(['Boruvka', 'Kruskal', 'Prim'], loc='upper_left')

plt.savefig('total.png')

if __name__ == '__main__':
    x1 = np.load('data_100_20000_20_100.npy')
    y1 = np.load('step_100_20000_20_100.npy')
    x2 = np.load('data_100_30000_25_100.npy')
    y2 = np.load('step_100_30000_25_100.npy')

    plot_total(x1, y1, x2, y2)

    plot_bo(x1, y1, x2, y1)
    plot_kr(x1, y1, x2, y1)
    plot_pr(x1, y1, x2, y1)

```