UNIVERSITÉ DE SHERBROOKE

IFT-436

ALGORITHME 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

22 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 Krustal et de Prim – ont été retenu pour l'étude. Ceux-ci ont été codés et exécutés avec Python.

Dans ce rapport seront présenté les algorithmes choisis, incluant les hypothèses concernant leur complexité, 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.

Outil 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 afin de se concentrer plus sur les tests et les résultats. Hélas, la réécriture de 2 des 3 algorithmes a été nécessaire.

Pour programmer en Python, l'environnement de développement choisi a été PyCharm 4.5.3. Dans PyCharm est inclus 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 algorithmes de calcul de l'arbre sous-tendant à coût minimal présentée en classe, ainsi que par un 3e algorithme inclus dans la liste des propositions dans l'énoncé du travail. Les algorithmes développés par Borůvka, Krustal et Prim sont tous de stratégie gloutonne, mais ne partagent pas 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 d'un dépôt GitHub de l'utilisateur « israelst », qui indique l'avoir adapté du livre « Algorithms » de Dasgupta, Papadimitriou et Vazurani. 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. Une classe représentant un ensemble d'ensembles disjoints a été créée pour unifier le fonctionnement des algorithmes de Borůvka et de Krustal. Le code a été tiré des sources originales de l'algorithme de Kruskal trouvé sur internet.

Hypothèses
Tout d'abord, la complexité des algorithmes devrait être les suivantes pour un m signifiant le nombre d'arêtes du graphe et n son 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 un temps d'exécution 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

La dernière affirmation est vraie, car le générateur utilisé pour générer les 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 égales à 10 fois le nombre de noeuds.

Générateur de nombres aléatoires

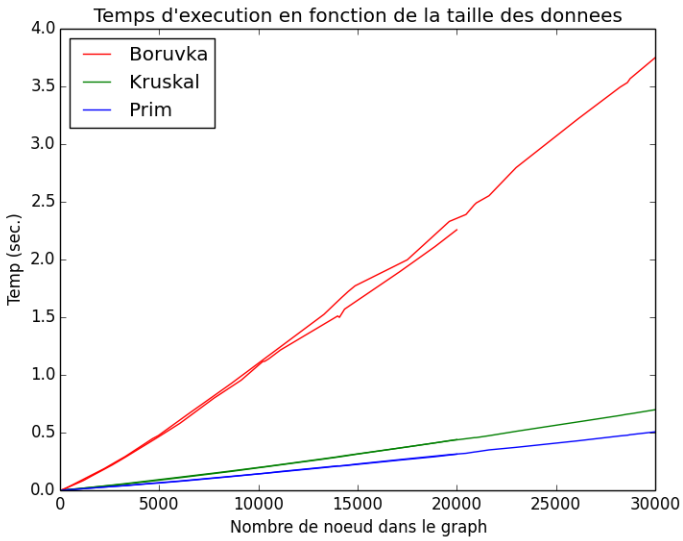
Le générateur de nombres aléatoires générique de Python est basé sur une implémentation en C du *Marsenne Twister*[1], un algorithme avec une période de $2^{19937}-1$ fortement testé. Dans le code du générateur de graphe, la fonction *randint* de Python est utilisée, qui est une application du *random* de Python qui utilise le *Marsenne Twister*.

Méthodologie

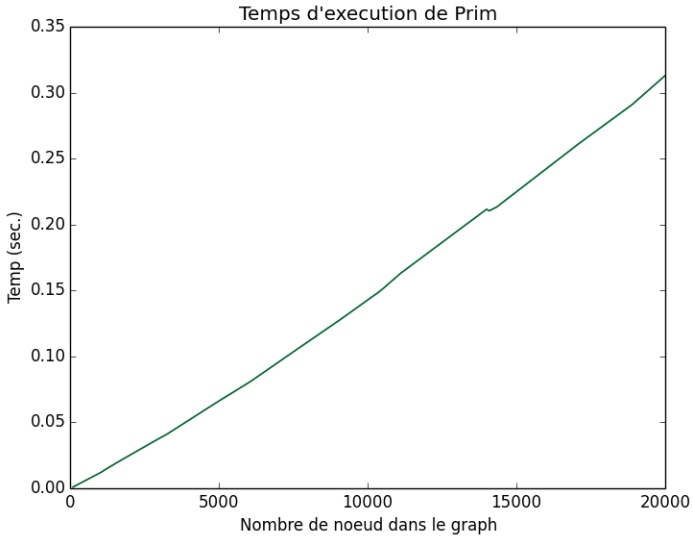
Pour exécuter les tests, un script a été créé. Celui-ci demande quatre paramètres : la taille minimum et max d'un graphe, le nombre d'échantillons à créer et le nombre de fois pour l'exécution d'une même taille. Cela permet de rouler les tests plus facilement et avec n'importe laquelle des consoles.

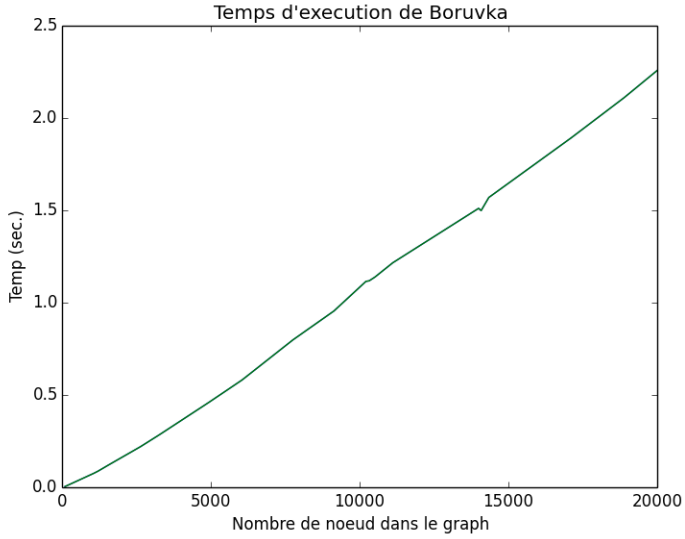
Pour recueillir les données sur le temps d'exécution versus le nombre de sommets, la librairie *matplotlib* a été importée. Cette librairie permet de tracer un graphique à partir des données recueillies, évitant ainsi la nécessité d'une compilation des résultats bruts dans un fichier pour ensuite produire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

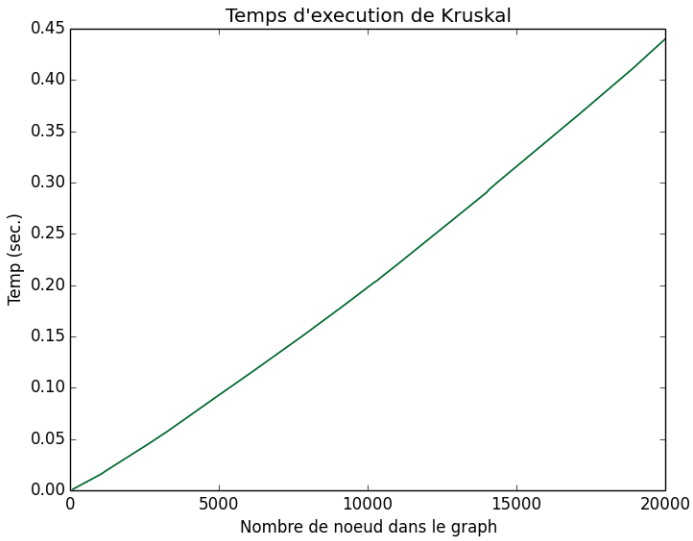
Résultats



1. https ://en.wikipedia.org/wiki/Mersenne_Twister


Temps d'execution de Prim
Temp (sec.)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0
5000
10000
15000
20000
Nombre de noeud dans le graph


Temps d'execution de Boruvka
Temp (sec.)
0.0
0.5
1.0
1.5
2.0
2.5
0
5000
10000
15000
20000
Nombre de noeud dans le graph



Conclusion
En conclusion,

Annexes

boruvka.py

```
from disjoint_set import DisjointSet

# adapted from https://en.wikipedia.org/wiki/Bor%C5%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

```
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

```
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
```

graph_generator.py

```
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)
```

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
```

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
import matplotlib.pyplot as plt

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

nb_algo = 3


def save_graph(timed, step_sizes):
    # plt.plot(step_sizes, timed[0])
    # plt.plot(step_sizes, timed[1])
    for i in range(nb_algo):
        plt.plot(step_sizes, timed[i])

    plt.title(u"Temps d'execution en  fonction de la taille des donnee")
    plt.ylabel("temp (sec.)")
    plt.legend(['Boruvka', 'Kruskal', 'Prim'], loc='upper left')

    print "step_sizes", step_sizes
    print "timed", timed
    plt.show()


def parm():
    parser = argparse.ArgumentParser(description="put someting 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]
    # TODO: iterate on the algo
    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)

    save_graph(timed, step_sizes)

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