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

21 juillet 2015

Introduction
Dans le cadre du dernier travail, une étude de performance d'exécution de différents algorithmes pour un même type de problème était à faire. Chaque équipe devait choisir un problème et au moins trois algorithmes pour résoudre le problème choisi. L'équipe a choisi le problème du calcul de l'arbre sous-tendant de coût minimal. Les trois algorithmes sont : Borůvska, Krustal et Prim. Les trois algorithmes ont été codé en Python avec le logiciel PyCharm.

Dans ce rapport, il sera présenté les outils de travail pour partager le code et facilité l'avancement du travail, les algorithmes choisis, la conception du générateur d'échantillon pour faire les test et les résultats de no tests avec les hypothèses les algorithmes.

Outil de travail
Pour ce travail, le langage choisit est le Python, version 2.7 . C'était un langage que certain connaissait plus que d'autre, mais simple et rapide à comprendre. De plus, il semblait plus simple de trouver les algorithmes choisis codés afin de ce concentrer plus sur les tests et les résultats.

Pour programmer en Python, le programme de programmation choisit est PyCharm. PyCharm donne accès à un outil Git pratique et facile à utiliser contrairement à d'autre programme comme Visual Studio pour C++. GitHub fut utilisé avant tout pour le partage et faciliter l'accès au code, la distribution de tâches et permettre de garder les un historique des changement au cas où il faudrait revenir en arrière.

Nos algorithmes
Le choix des algorithmes fut en fonction de ce qui a été vu en classe et des algorithmes proposés dans l'énoncé du travail. Les algorithmes choisis pour ce travail sont Borůvska, Krustal et Prim. Ce sont trois algorithmes de stratégie gloutonne, mais avec des complexités différentes que nous présenterons dans nos hypothèses.

Le seul algorithme dont le code a été trouvé sur Internet, c'est celui de Kruskal. Quant à Borůvka et Prim, aucun code acceptable a été trouvé, donc c'était préférable de les écrire pour ne pas perdre trop de temps sur cet tâche. Dans ce cas-ci, un code raisonnable a été déterminé par le fait qu'il ne demande pas de modification majeur et qu'il est construit de manière optimal.

Classe DisjointSet
La classe *DisjointSet* a été créé pour les algoritmes Borůvska et Krustal, qui est tiré du code de l'algorithme de Kruskal. Celui-ci permet de faire des ensembles de sommets, trouver un sommet et relier 2 sommets ensembles. Cette partie de code était avant tout utilisée pour l'algorithme de Kruskal. Celle-ci a été mise dans une classe pour avoir la possibilité, lorsque possible, de l'utiliser dans d'autre algorithme.

Générateur d'échantillon
Le générateur de graphe a été construit selon la documentation trouvé sur Internet : *https ://en.wikipedia.org/wiki/Me*
Dans le code du générateur de graphe, la fonction « *random* » de Python est utilisée. Contrairement à la fonction de C++, celui-ci est efficace.

Expérience de travail

Il sera présenté dans cette section les hypothèses puis les résultats de des tests. Il sera expliqué la méthodologie pour obtenir les résultats. Il sera considérer n le nombre de sommets et m le nombre d'arêtes.

Hypothèses

D'abord, la complexité des algorithme devraient être : $O(m \log n)$ pour Borůvka, $O(m \log m)$ pour Kruskal et $O(m \log n)$ pour Prim. Il est à croire qu'étant de même complexité, Borůvka et Prim auront un temps de calcul similaire. Pour ce qui est de Kruskal, il devrait être le plus performant des trois algorithmes par sa différence $\log m$, car il y aura au plus $\frac{n(n-1)}{2}$ arêtes.

Méthodologie

Pour faire le temps de calcul pour un graphe de n sommets, une moyenne du temps est fait sur une boucle qui exécute 100 fois sur une même taille d'échantillon. Pour ce qui est de l'échantillonnage, il est fait par pas de 10^i, pour $i = 1, 2, ..., k$. Le k est déterminé selon la rapidité d'exécution de l'ordinateur, choisit dans le laboratoire informatique.

Pour recueillir les données sur le temps d'exécution versus le nombre de sommet, la librairie *matplotlib* a été importé. Cette librairie permet de faire un graphique des données recueillies. Cela évite de devoir les compiler dans un fichier pour faire un graphique avec un logiciel comme Excel.

Résultats

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 random

# 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 = 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 = random.randint(0, n-1)
        v = 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 = 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)
```

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 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', type=int, default=10000, help="Max data size")
    parser.add_argument('--step_size', '-s', type=int, default=100, help="Data size step increment")
    parser.add_argument('--nb_test', '-t', type=int, default=1000, help="Number of test at each data size")
    parser.add_argument('--step_number', '-sn', type=int, help="Number of step to go from 1 to n" +
                                                               " in each batch of test. Override data step size")
    args = parser.parse_args()

    return args


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

    s_size = args.step_size

    if args.step_number is not None:
        s_size = args.n / args.step_number

    step_sizes = range(s_size, args.n+s_size, s_size)
    timed = np.zeros((nb_algo, len(step_sizes)))

    fct = [boruvka, kruskal, prim]
    # TODO: iterate on the algo
    for algo_idx in range(nb_algo):
        random.setstate(random_state)

      # TODO: iterate on the step_size
        for sz_idx in range(len(step_sizes)):
            start_time = timeit.default_timer()
            for test_iter in xrange(0, args.nb_test):
                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

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

        for test_iter in xrange(0, args.nb_test):
```

```
                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 /= step_sizes

    save_graph(timed, step_sizes)

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