Rapport du projet de théorie des graphes

Maxence Ahlouche Maxime Arthaud Korantin Auguste
Martin Carton Thomas Forgione Thomas Wagner

7 octobre 2013

Table des matières

Listings

1 Présentation de l'équipe

Cette équipe a été menée par Korantin Auguste, assisté de son Responsable Qualité Martin Carton. Les autres membres de l'équipe sont Thomas Wagner, Thomas Forgione, Maxime Arthaud, et Maxence Ahlouche. Tous les membres de l'équipe ont été présents à chacune des séances lors de cette UA.

2 Modélisation mathématique

Nous avons choisi de représenter nos graphes comme une liste de sommets, chacun ayant une liste d'arêtes.

3 Analyse mathématique

3.1 Graphes eulériens

3.2 Graphes hamiltonien

Un graphe hamiltonien est un graphe est un graphe sur lequel on peut trouver un cycle passant par tout les sommets une et une seule fois.

Le problème de savoir si un graphe admet un cycle, ou même un chemin hamiltonien est NP-complet, de même que de trouver un tel cycle ou chemin s'il y en a.

Il existe cependant des conditions suffisantes pour lesquelles on peut affirmer qu'un graphe est hamiltonien ou non.

3.3 Problème du postier chinois

3.4 Problème voyageur de commerce

On s'intéresse ici à passer par tout les points d'un ensemble une et une seule fois en minimisant la distance totale du cycle.

On peut modéliser ce problème par un graphe complet, dont les arêtes ont un coup qui correspond à la distance entre chaque point, on cherche alors le cycle hamiltonien de coût minimal. On sait qu'un tel cycle existe car le graphe est complet.

Cependant trouver un tel graphe est un problème NP-difficile, il n'existe donc pas d'algorithme efficaces pour trouver ce cycle.

Il existe cependant plusieurs heuristique pour trouver un cycle dans ce graphe.

Un heuristique simple consiste à partir d'un sommet au hasard du graphe et d'aller au sommet le plus proche sur lequel on est pas encore passer (puis à retourner au sommet de départ pour boucler le cycle). Cet algorithme n'offre cependant aucune garantie de résultat, il existe même des graphes pour lesquels il donne le pire cycle.

Il existe aussi des algorithmes non-constructifs comme le 2-opt, qui essaye d'améliorer un cycle donné en échangeant des sommets.

4 Méthode de résolution

4.1 Graphes eulériens

4.2 Graphes hamiltonien

4.3 Problème du postier chinois

4.4 Problème voyageur de commerce

5 Algorithmes

5.1 Graphes eulériens

5.2 Graphes hamiltonien

5.3 Problème du postier chinois

5.4 Problème voyageur de commerce

6 Conclusion

7 Tests

7.1 Problème voyageur de commerce

Nous avons lancé cet algorithme sur plusieurs "grands" graphes [1], les résultats sont présentés dans la table 1 [2].

Fichier de test	Résultat optimum	Plus proche voisin	Plus proche voisin + 2-opt	Plus proche voisin amélioré + 2-opt
berlin52.tsp	7542	8981/19.1%	8060/6.7%	7810/3.6%
bier127.tsp	118282	137297/16.7%	125669/6.2%	N/A
d657.tsp	48912	62176/27.1%	N/A	N/A
fl1577.tsp	22249	N/A	N/A	N/A
u724.tsp	41910	55344, 32.1%	N/A	N/A

TABLE 1 – Résultats pour TSP

On remarque que bien qu'il ne fournisse aucune garantie, l'algorithme du plus proche voisin donne des résultats plutôt bons.

1. Trouvés sur `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`.
2. N/A indique que l'algorithme est trop long ou cause une erreur à cause de la taille du graphe, pour chaque méthode de résolution sont données les longueurs des chemins trouvés et l'erreur relative avec le résultat optimum.

8 Annexe

Listing 1 – Classe pour représenter un graphe

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-

class Edge:
    def __init__(self, origin, dest, cost=1):
        self.origin = origin
        self.dest = dest
        self.cost = cost

    def other_side(self, node):
        return self.origin if node is self.dest else self.dest

    def __repr__(self):
        return "Edge(%s, %s, %s)" % (self.origin.data, self.dest.data, self.cost)

class Node:
    def __init__(self, data):
        self.edges_out = set() # liste des arêtes qui sortent du nœud pour
            pointer sur d'autres
        self.data = data # must be unique

    def __hash__(self):
        return hash(self.data)

    def degree(self):
        return len(self.edges_out)

    def __repr__(self):
        return "Node(%s, [%s])" % (self.data, ', '.join(map(repr,
            self.edges_out)))

    def cost_to(self, other):
        return self.edge_to(other).cost

    def edge_to(self, other):
        for edge in self.edges_out:
            if edge.other_side(self) == other:
                return edge
        raise RuntimeError("Martin fait trop de C (en vrai, le graphe est pas
            complet)")

class Graph:
    def __init__(self, path=None):
        self.nodes = [] # liste des noeuds du graphe
        self.name = ""

        if path:
            self.name = path.split('/')[-1]

            nodes_added = dict()
            with open(path, 'r') as f:
                nb_v, nb_e, oriented = map(int, f.readline().split(' '))
                self.oriented = oriented == 1

                for i in range(nb_v):
                    data = int(f.readline())
                    n = Node(data)
```

```
                    self.nodes.append(n)
                    nodes_added[data] = n

                for i in range(nb_e):
                    #(orig, dest, cost) = map(int, (f.readline()+" 1").split('
                        ')[:3])
                    line = f.readline()
                    try:
                        orig, dest, cost = map(int, line.split(' '))
                    except(ValueError):
                        orig, dest = map(int, line.split(' '))
                        cost = 1

                    n_orig = nodes_added[orig]
                    n_dest = nodes_added[dest]
                    edge = Edge(n_orig, n_dest, cost)
                    n_orig.edges_out.add(edge)
                    if not self.oriented:
                        n_dest.edges_out.add(edge)

    def __repr__(self):
        return 'Graph(\n%s\n)' % ',\n'.join(map(repr, self.nodes))

    def order(self):
        return len(self.nodes)

    def copy(self):
        g = Graph()
        g.oriented = self.oriented

        for node in self.nodes:
            g.nodes.append(Node(node.data))

        g_nodes = set(g.nodes)
        while g_nodes:
            node_g = g_nodes.pop()
            node = filter(lambda n: n.data == node_g.data, self.nodes)[0]
            for edge in node.edges_out:
                other_side_g = filter(lambda n: n.data ==
                    edge.other_side(node).data, g.nodes)[0]
                if self.oriented:
                    node_g.edges_out.add(Edge(node_g, other_side_g, edge.cost))
                elif other_side_g in g_nodes:
                    edge_g = Edge(node_g, other_side_g, edge.cost)
                    node_g.edges_out.add(edge_g)
                    other_side_g.edges_out.add(edge_g)

        return g
```

Listing 2 – Algorithmes relatifs au graphes eulériens

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-

from connected import *

def get_odd_vertices(graph):
    """
    Returns true if the graph is eulerian or semi-eulerian
    """
    if not graph.oriented:
        nb_odd_deg = 0
```

```
        odd_list = []
        for n in graph.nodes:
            if len(n.edges_out) % 2 != 0:
                odd_list.append(n)
        return odd_list
    else:
        raise NotImplementedError()

def is_eulerian(graph):
    nb_odd_vertices=len(get_odd_vertices(graph))
    return nb_odd_vertices == 0 and is_connected(graph)

def is_semi_eulerian(graph):
    nb_odd_vertices = len(get_odd_vertices(graph))
    return nb_odd_vertices == 2 and is_connected(graph)

def eulerian_path_euler(graph):
    def aux(node, visited_edges):
        result = [node]
        final_result = [node]

        while True:
            edges = filter(lambda e: e not in visited_edges, node.edges_out)
            if not edges:
                break
            else:
                edge = edges[0]
                node = edge.other_side(node)
                result.append(node)
                visited_edges.add(edge)

        for node in result[1:]:
            cycle = aux(node, visited_edges)
            final_result += cycle

        return final_result

    if not graph.is_connected():
        return None

    odd_vertices = get_odd_vertices(graph)
    if len(odd_vertices) == 0:
        return aux(graph.nodes[0], set())
    elif len(odd_vertices) == 2:
        return aux(odd_vertices[0], set())
    else:
        return None

# no multigraph nor reflexive edge
# equivalent to bruteforce
def eulerian_path_lat_mat(graph):
    def gen_lat_mat(graph):
        nb_n = len(graph.nodes)
        lat_mat = [[None for i in range(nb_n) ] for j in range(nb_n) ]
        for n in graph.nodes:
            for e in n.edges_out:
                n2 = e.other_side(n)
                lat_mat[n.data-1][n2.data-1] = [[n.data, n2.data]]
        return lat_mat

    def lat_mat_mul(a, b):
        nb_n = len(a)
```

```
        result = [[None for i in range(nb_n) ] for j in range(nb_n) ]
        for i in range(nb_n):
            for j in range(nb_n):
                # for each cell
                result[i][j] = []
                for k in range(nb_n):
                    # "multiplication"
                    cell_a = a[i][k]
                    cell_b = b[k][j]
                    if cell_a is not None and cell_b is not None:
                        for l in cell_a: # for each path in cell_a
                            for m in cell_b: # for each path in cell_b
                                path = l[:]
                                path.extend(m[1:])
                                edges = set()
                                for n in range(len(path)-1):
                                    edge = (path[n], path[n+1])
                                    edge_rev = (path[n+1], path[n])
                                    if edge not in edges:
                                        edges.add(edge)
                                        edges.add(edge_rev)
                                    else:
                                        path = None
                                        break
                                if path is not None:
                                    result[i][j].append(path)
                if result[i][j] == []:
                    result[i][j] = None
        return result

    def lat_mat_pow(lat_mat, n):
        result = lat_mat_mul(lat_mat, lat_mat)
        for i in range(n-2):
            result = lat_mat_mul(lat_mat, result)

        return result


    nb_a = 0
    for n in graph.nodes:
        nb_a += len(n.edges_out)
    nb_a /=2
    a = gen_lat_mat(graph)
    b = lat_mat_pow(a, nb_a)
    for row in b:
        for cell in row:
            print cell

    return None
```

Listing 3 – Algorithmes relatifs au graphes hamiltonien

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
import graphs

def is_hamiltonian(graph):
    # rapid test, only sufficient, not necessary
    min_degree = min(node.degree() for node in graph.nodes)
    if min_degree >= graph.order() / 2:
        return True
```

```
    # general test, complexity sucks
    return hamiltonian_path2(graph) != None

def hamiltonian_path(graph, node_from=None, nodes_done=frozenset()):
    if node_from is None:
        node_from = graph.nodes[0]

    nodes_done = nodes_done | frozenset((node_from,))

    if len(nodes_done) == graph.order():
        return [node_from]

    for edge in node_from.edges_out:
        other = edge.other_side(node_from)
        if other in nodes_done:
            continue
        path = hamiltonian_path(graph, other, nodes_done)
        if path:
            return [node_from] + path

    return None

def hamiltonian_path2(graph, node_from=None, nodes_done=frozenset()):
    if node_from is None:
        node_from = graph.nodes[0]

    nodes_done = nodes_done | frozenset((node_from,))

    if len(nodes_done) == graph.order():
        return [node_from]

    poss = filter(lambda x: x.other_side(node_from).degree() == 1,
        node_from.edges_out)

    if len(poss) == 1:
        path = hamiltonian_path(graph, poss[0].other_side(node_from), nodes_done)
        return [node_from] + path if path else None
    elif len(poss) > 1:
        return None

    for edge in node_from.edges_out:
        other = edge.other_side(node_from)
        if other in nodes_done:
            continue
        path = hamiltonian_path2(graph, other, nodes_done)
        if path:
            return [node_from] + path

    return None

def read_hcp(path):
    with open(path, 'r') as f:
        for i in range(1,4): f.readline()

        line = f.readline()
        nb_nodes = int(line.split()[2])

        f.readline()
        f.readline()

        nodes = []
        for i in range(0, nb_nodes):
```

```
            nodes.append(graphs.Node(i))

        line = f.readline()
        while line != "-1\n":
            a, b = map(int, line.split())
            edge = graphs.Edge(nodes[a-1], nodes[b-1])
            nodes[a-1].edges_out.add(edge)
            nodes[b-1].edges_out.add(edge)
            line = f.readline()

        g = graphs.Graph()
        g.nodes = nodes
        g.oriented = False
        return g
```

Listing 4 – Algorithmes relatifs au TSP

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-

import sys
import graphs
import heapq

def nearest_neighbor(graph, node_from=None, first_node=None,
    nodes_done=frozenset()):
    """
    Nearest Neighbor algorithm, simple approximation for TSP problem.
    Requires the graph to be complete.
    Returns (cost, [nodes...]).
    """

    if node_from is None:
        node_from = graph.nodes[0]

    if first_node is None:
        first_node = node_from

    nodes_done = nodes_done | frozenset((node_from,))

    if len(nodes_done) == graph.order():
        return (node_from.cost_to(first_node), [node_from, first_node])

    heap = []
    for edge in node_from.edges_out:
        other = edge.other_side(node_from)
        if other not in nodes_done:
            heapq.heappush(heap, (edge.cost, other))

    best_cost = 0
    best_path = None
    for i in range(1): #range(2 if len(nodes_done) < 14 else 1):
        if not heap:
            break
        edge_cost, node = heapq.heappop(heap)
        cost, path_end = nearest_neighbor(graph, node, first_node, nodes_done)
        cost += edge_cost

        if cost < best_cost or best_path is None:
            best_cost = cost
            best_path = path_end
```

```
    return (best_cost, [node_from] + best_path)

def two_opt(solution):
    """
        2-opt algorithm, try to find a better solution than a given one.
    """

    best_cost, best_path = solution
    improvement_made = True

    while improvement_made:
        improvement_made = False
        for i in range(len(best_path)-1):
            for j in range(i + 1, len(best_path)-1):
                ni = best_path[i]
                nj = best_path[j]
                new_cost = best_cost - best_path[i+1].cost_to(ni) -
                    best_path[j+1].cost_to(nj) + ni.cost_to(nj) +
                    best_path[j+1].cost_to(best_path[i+1])

                if new_cost < best_cost:
                    improvement_made = True
                    best_cost = new_cost
                    best_path = best_path[:i+1] + best_path[i+1:j+1][::-1] +
                        best_path[j+1:]

    return (best_cost, best_path)

def read_tsp(path):
    """
        Create a graph from a file of the form:
            [6 useless lines]
            1 x1 y1
            2 x2 y2
            ...
        These graphs are found here
            http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/
    """
    def distance(a, b):
        return ( (a[0]-b[0])*(a[0]-b[0]) + (a[1]-b[1])*(a[1]-b[1]) ) ** 0.5

    points = []
    with open(path, 'r') as f:
        for i in range(1, 7): f.readline()

        line = f.readline()
        while line != "EOF\n":
            x, y = map(float, line.split()[1:3])
            points.append((x, y))
            line = f.readline()

    node_id = 0
    nodes = []
    for (x, y) in points:
        new_node = graphs.Node(node_id)
        node_id += 1
        for other in nodes:
            edge = graphs.Edge(other[1], new_node, distance((x, y), other[0]))
            other[1].edges_out.add(edge)
            new_node.edges_out.add(edge)

        nodes += [((x, y), new_node)]
```

```
    g = graphs.Graph()
    g.nodes = map(lambda x: x[1], nodes)
    g.oriented = False
    g.name = path.split('/')[-1]
    return g
```

Listing 5 – Tests

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
from graphs import Graph, Node, Edge
from hamiltonian import *
from connected import *
from eulerian import *
from tsp import *
import datetime

def print_ok(string):
    print "\033[92m" + string + "\033[0m"

def print_warn(string):
    print "\033[93m" + string + "\033[0m"

def print_err(string):
    print "\033[91m" + string + "\033[0m"

def test(condition, tested_fn, graph_nb, graph_name, exec_time):
    if condition:
        print_ok("OK in %s.%ss"% (exec_time.seconds, exec_time.microseconds))
    else:
        print_err("Error testing %s on graph number %s: %s" % (tested_fn,
            graph_nb, graph_name))

def test_one(graphs, fun, indice):
    fun_name = "Graph." + fun.__name__ + "()"
    print "Testing " + fun_name + "..."
    i = 0
    for g in graphs:
        i = i + 1
        start = datetime.datetime.now()
        condition = fun(g[0]) == g[indice]
        exec_time = datetime.datetime.now() - start
        test(condition, fun_name, i, g[0].name, exec_time)

if __name__ == '__main__':
    graphs = []

    # Format: [Graph, connected, eulerian, semi-eulerian, hamiltonian]
    graphs.append([Graph('tests/1.gph'), True, True, False, True])
    graphs.append([Graph('tests/2.gph'), True, False, True, False])
    graphs.append([Graph('tests/3.gph'), False, False, False, False])
    graphs.append([Graph('tests/4.gph'), True, False, False, True])
    graphs.append([read_tsp('tests/berlin52.tsp'), True, False, False, True])
    graphs.append([read_tsp('tests/d657.tsp'), True, True, False, True])
#    graphs.append([read_tsp('tests/fl1577.tsp'), False, False, False, False])
    graphs.append([read_tsp('tests/bier127.tsp'), True, True, False, True])
    graphs.append([read_tsp('tests/u724.tsp'), True, False, False, True])
    graphs.append([Graph('tests/complete.gph'), True, True, False, True])
    graphs.append([Graph('tests/complete_cost.gph'), True, True, False, True])
#    graphs.append([read_hcp('tests/alb1000.hcp'), True, True, False, True]) #todo
#    graphs.append([read_hcp('tests/alb2000.hcp'), True, True, False, True]) #todo
```

```
    #tests connexité
    test_one(graphs, is_connected, 1)

    #tests eulérianité
    test_one(graphs, is_eulerian, 2)

    # tests semi eulerianité
    test_one(graphs, is_semi_eulerian, 3)

    # tests hamiltonian
    test_one(graphs, is_hamiltonian, 4)
```