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

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.

Dans la suite nous noterons n le nombre de sommets du graphe.

3 Graphes eulériens

3.1 Analyse mathématique

Un graphe eulérien est un graphe contenant un cycle eulérien, c'est-à-dire un chemin parcourant toutes les arêtes du graphe une et une seule fois, en revenant au sommet de départ ; ce problème est donc celui de la goudronneuse qui doit passer sur toutes les rues sans pouvoir repasser dessus une seconde fois. Un théorème fondamental garantit qu'un graphe connexe est eulérien si et seulement si chacun de ses sommets est associé à un nombre pair d'arêtes.

Un graphe semi-eulérien, quant à lui, contient une chaîne eulérienne : celle-ci passe également par toutes les arêtes du graphe une seule et unique fois, mais ne retourne pas au point de départ. Le théorème précédent se généralise alors aux graphes semi-eulériens : un graphe connexe et semi-eulérien si et seulement tous ses sommets sauf deux sont associés à un nombre pair d'arêtes. Dans ce cas, la chaîne eulérienne aura pour départ l'un des deux sommets associés à un nombre impair d'arêtes et pour point d'arrivée le deuxième.

3.2 Méthode de résolution

Afin de trouver une chaîne ou un cycle eulérien dans un graphe, nous avons implémenté deux méthodes : une méthode qui teste toutes les possibilités, et une autre plus intelligente et moins coûteuse.

3.2.1 Matrices latines

La première méthode est inspirée des matrices latines. Chaque coefficient de la matrice sera un ensemble de chemins, un chemin étant lui-même une liste de sommets. La matrice latine de notre graphe sera la matrice M dont chaque coefficient $m_{i,j}$ vaudra :

— l'ensemble vide si le nœud i n'est pas relié au nœud j dans le graphe ;

— un ensemble contenant pour unique élément le chemin $[N_i, N_j]$ si les nœuds i et j sont reliés (où N_k représente le nœud k).

Nous définissions ensuite un produit sur les coefficients d'une telle matrice. Le produit de deux chemins sera :

— nul si le dernier nœud du premier chemin n'est pas le premier nœud du deuxième ;

— la concaténation des deux chemins sinon.

Le produit de deux ensembles de chemins sera l'ensemble contenant les produits de chaque couple de nœuds.

Pour tout k entier naturel, le coefficient (i, j) de la matrice M^k représentera l'ensemble des chemins de longueur k reliant les nœuds i et j.

Puisque un chemin eulérien passe une unique fois par chaque arête, il suffira de calculer la matrice latine élevée à cette puissance pour trouver sur sa diagonale l'ensemble des cycles possibles. En éliminant à chaque produit les chemins qui passent plusieurs fois par la même arête, on trouve l'ensemble des cycles eulériens.

La complexité de cette algorithme est exponentielle, calculer la puissance de la matrice latine revient en fait à calculer chaque chemin possible dans le graphe, et tester s'il est un cycle eulérien ou non.

3.2.2 Algorithme d'Euler

La deuxième méthode, basée sur l'algorithme d'Euler est nettement plus efficace. Une fonction récursive cherche un cycle eulérien d'un sous-graphe de notre graphe de départ, puis s'appelle récursivement sur chacun des sommets parcourus par ce chemin, dans le graphe où l'on a supprimé les arêtes déjà parcourues. En reconstruisant ces cycles astucieusement, on parvient à trouve un cycle eulérien de complexité linéaire en le nombre d'arêtes du graphe.

3.3 Algorithmes

3.3.1 Méthode de la matrice latine

```
Entrée : un graphe
Sortie : la liste des cycles eulériens dans le cas d'un graphe eulérien
         la liste des chemins eulériens dans le cas d'un graphe semi-eulérien
         la liste vide sinon

Construire la matrice latine du graphe :
    construire une matrice à n lignes et n colonnes
    remplir la matrice de listes vides
    pour chaque nœud du graphe:
        pour chaque arête sortant de ce nœud:
            ajouter la liste [noeud de départ, noeud d'arrivée] à la case de la
                matrice correspondante

n = "le nombre d'arêtes total du graphe"

calculer la puissance (n-1)ième de la matrice

pour chaque coefficient de la matrice ainsi calculée:
    si le coefficient n'est pas nul:
```

```
        concaténer ce coefficient à la variable de retour
```

3.3.2 Produit matriciel

```
Entrée : A et B deux matrices latines
Sortie : le produit de ces deux matrices

construire la matrice de retour à n lignes et n colonnes
initialiser chaque coefficient de cette matrice à la liste vide

pour chaque coefficient de la matrice de retour:
    pour k allant de 1 jusqu'à n:
        calculer les chemins produits entre a(i,k) et b(k,j)
        ajouter au coefficient de la matrice ces chemins
```

3.3.3 Produit entre listes de chemins (coefficients de matrices latines)

```
Entrée : liste_1 et liste_2 deux listes de chemins
Sortie : une liste de chemins

créer une liste de chemins vide (liste de retour)
pour i dans liste_1:
    pour j dans liste_2:
        construire le chemin résultant de la concaténation de i et j (en enlevant
            le nœud présent deux fois)
        construire un ensemble de chemins vide
        pour k allant de 1 à la longueur du chemin construit:
            construire le chemin élémentaire menant du nœud k au nœud k+1
            si ce chemin n'est pas dans l'ensemble:
                ajouter ce chemin dans l'ensemble
            sinon:
                rendre le chemin nul
                sortir de la boucle

        si le chemin n'est pas nul:
                concaténer le chemin trouvé à la liste de retour
retourner la liste de retour
```

4 Graphes hamiltonien

4.1 Analyse mathématique

Un graphe (semi-)hamiltonien est un graphe sur lequel on peut trouver un cycle(/chemin) passant par tout les sommets une et une seule fois. Ce problème est donc celui de l'enfant qui souhaite visiter de manière unique toutes les salles d'un musée.

Le problème de savoir si un graphe est (semi-)hamiltonien est NP-complet, de même que de trouver un 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.

Par exemple un graphe complet est forcement hamiltonien (utile dans le cas du voyageur de commerce, voir section 6), il existe aussi des conditions sur les degrès des sommets (théorème de Dirac, d'Ore, etc.).

4.2 Méthode de résolution

Pour tester si un graphe est hamiltonien, nous avons utilisé les théorèmes de Dirac et Pósa qui donnent des conditions nécessaires, si ces conditions ne sont pas vérifiées, comme il n'y a aucun théorème qui permette d'affirmer qu'un graphe n'est pas semi-hamiltonien, on recherche un chemin hamiltonien dans ce graphe.

Pour rechercher un chemin hamiltonien dans un graphe, nous avons écrit un algorithme qui recherche parmi tout les chemins possibles. Sa complexité dans le pire des cas est donc très mauvaise : $O(n!)$. Comme on peut s'arêter dès qu'on a trouvé un chemin sans devoir tester tout les autres chemins possibles, la complexité moyenne sera inférieure.

Nous avons écrit une version améliorée de cet algorithme qui essaye d'éviter les culs de sac.

4.3 Algorithmes

4.3.1 Tests de semi-hamiltoniannité

```
Entrée : un graphe
Sortie : un booléen indiquant si le graphe est semi−hamiltonien ou non

si le graphe suit les conditions du théorème de Dirac ou du théorème de Pósa:
    retourner Vrai
sinon:
    chercher un chemin hamiltonien
    retourner Vrai si on en a trouvé un, Faux sinon
```

4.3.2 Recherche de chemin hamiltonien

```
Entrée : un graphe graph
         un point de départ optionnel node_from
         un ensemble (éventuellement vide) de nœuds déjà parcouru nodes_done
Sortie : un chemin hamiltonien sous la forme d'une liste ordonnée de points, ou
    None s'il n'en existe pas

Si la fonction a été appelée sans node_from:
    node_from = "un nœud de graph"

ajouter node_from à nodes_dones

si cardinal(node_from) == ordre(graph):
    retourner [node_from]

pour chaque arête dans le graphe:
    autre = "le point opposé à node_from par rapport à cette arête"
    si autre dans nodes_done:
        passer à la prochaine arête
```

```
    appeler la fonction récursivement avec graphe, node_from et nodes_dones comme
        paramètre
    si la liste retournée est non-vide:
        y ajouter node_from au début et la retourner

retourner None (si on arrive ici, aucun chemin n'est bon)
```

4.4 Tests

Nos algorithmes fonctionnent biens sur de petits graphes, mais ils sont beaucoup trop lents pour être utilisés sur de grands graphes pour lesquels il y a “peu” d’arêtes : jusqu’à 30 sommets et 46 arêtes, l’algorithme trouve une solution en moins d’une seconde. Pour 40 sommets et 63 arêtes, il faut déjà une minute. Pour 100 sommets et 150 arêtes, l’algorithme prend tellement de temps que l’avons arrêté après quelques heures.

Par contre, pour des graphes ayant beaucoup d’arêtes (graphes “presque complets”), l’algorithme reste rapide.

Nous avons pu constater que la deuxième version de notre algorithme ne sert à rien : il n’y a aucune amélioration des performances.

5 Problème du postier chinois

5.1 Analyse mathématique

Le problème du postier chinois consiste à trouver le plus court chemin dans un graphe connexe passant au moins une fois par chaque arête, et revenant à son point de départ ; ce problème est donc celui du facteur qui souhaite réaliser une tournée la plus rapide possible en passant par toutes les rues et retournant à la poste.

Ce problème peut être réduit à la recherche d’un couplage parfait de coût minimum, il peut donc être résolu en temps polynomial dans le cas général.

5.2 Méthode de résolution

Tout d’abord, si le graphe est eulérien, il suffit d’appliquer l’algorithme d’Euler pour avoir le chemin voulu.

Sinon, la méthode de résolution consiste à transformer le graphe en graphe eulérien :

- on crée d’abord le graphe partiel contenant uniquement les sommets de degré impair ;
- on transforme ensuite ce graphe en clique : pour chaque couple de sommets non reliés entre eux, on crée une arête les rejoignant, de poids égal au coût le plus faible possible pour rejoindre ces sommets dans le graphe inital (ceci se calcul facilement avec l’algorithme de Dijkstra) ;
- on cherche le couplage parfait de coût minimum : c’est à dire l’ensemble d’arêtes disjointes couvrant tous les sommets du graphe, dont la somme des poids doit est le plus faible possible. Pour cela, on peut utiliser des algorithmes comme celui

d'Edmonds, mais dans notre implémentation, nous avons utilisé la bruteforce, par manque de temps ;
— pour chaque arête de cet ensemble, on double le chemin le plus court reliant les nœuds reliés par cette arête dans le graphe initial ;
— on obtient alors un graphe eulérien sur lequel on applique l'algorithme d'Euler.

5.3 Algorithmes

5.3.1 Algorithme de Dijkstra

On aura besoin de l'algorithme de Dijkstra, pour retrouver le chemin le plus court entre 2 sommets :

```
Entrée : (s1, s2) 2 sommets
Précondition : il existe un chemin entre s1 et s2
Sortie : (coût, chemin) avec chemin le plus court chemin entre s1 et s2, et coût
le coût associé

pour chaque sommet dans le graphe:
    sommet.parcouru = infini
    sommet.précédent = 0

s1.parcouru = 0
sommets_non_visites = ensemble des sommets du graphe

tant que sommets_non_visites est non vide:
    s = le sommet de sommets_non_visites avec s.parcouru minimum
    supprimer s de sommets_non_visites

    pour chaque sommet s2 dans les fils de s:
        si s2.parcouru > s.parcouru + poids de l'arc entre s et s2:
            s2.parcouru = s.parcouru + poids de l'arc entre s et s2
            s2.précédent = s
            ajouter s2 dans sommets_non_visites

chemin = vide
s = s2

tant que s != s1
    chemin = s + chemin
    s = s.précédent

chemin = s1 + chemin
retourner (s2.parcouru, chemin)
```

5.3.2 Algorithme du postier chinois

```
Entrée : g (Graphe)
Précondition : g non orienté et connexe
Sortie : le cycle le plus court permettant de visiter toutes les arêtes de g

# Création du graphe partiel
graphe_partiel = graphe vide

pour chaque sommet de g:
    si le sommet est de degré impair:
```

```
        créer le sommet dans graphe_partiel

pour chaque arête de g:
    si ses 2 sommets sont dans graphe_partiel:
        créer la même arête dans graphe_partiel

# Transformation en clique
pour chaque couple de sommet (s1, s2) dans graphe_partiel:
    s'il n'y a pas d'arête reliant s1 et s2:
        (cout, chemin) = dijkstra(s1, s2)
        créer l'arête reliant s1 et s2 dans graphe_partiel, de coût cout

# Recherche du couplage parfait de coût minimum : méthode bruteforce
fonction aux(arêtes, sommets_visites, cout):
    si sommets_visites contient tous les sommets de graphe_partiel:
        retourner (arêtes, cout)
    sinon:
        meilleur_couplage = Vide
        meilleur_cout = 0

        pour chaque arête de graphe_partiel:
            si les 2 sommets de l'arête ne sont pas dans sommets_visites:
                arêtes_copie = copie de arêtes
                sommets_visites_copie = copie de sommets_visites

                ajouter arête dans arêtes_copie
                ajouter les 2 sommets de arête dans sommets_visites_copie
                couplage, cout = aux(arêtes_copie, sommets_visites_copie, cout +
                    cout de arête)

                si meilleur_couplage = Vide ou meilleur_cout > cout:
                    meilleur_couplage = couplage
                    meilleur_cout = cout

        retourner (meilleur_couplage, meilleur_cout)

couplage, cout = aux(ensemble vide, ensemble vide, 0)

# On double les arêtes dans couplage
pour chaque arête dans couplage:
    (s1, s2) = sommets reliés par arête dans g
    (cout, chemin) = dijkstra(s1, s2)
    pour chaque arête dans chemin:
        doubler arête dans g

retourner le cycle eulérien de g
```

6 Problème voyageur de commerce

6.1 Analyse mathématique

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. Ce problème est donc celui de la fraiseuse qui doit percer des trous dans une plaque le plus rapidement possible. Il pourrait aussi servir à résoudre le problème du car de touristes.

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

6.2 Méthode de résolution

Bien que la résolution exact de ce problème soit NP-complet, il existe des méthodes approchées de résolution.

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 est en $O(n)$ et donc rapide. Mais il n'offre cependant aucune garantie de résultat, il existe même des graphes pour lesquels il donne le pire cycle.

Nous avons aussi écrit une version améliorée de cet algorithme qui plutôt que d'aller systématiquement vers le voisin le plus proche essaye les deux voisins les plus proches, la complexité de l'algorithme serait alors exponentielle, nous avons donc limité ce choix au début du cycle construit, puis l'algorithme choisit toujours le voisin le plus proche. La longueur du chemin à partir de laquelle on repart vers le voisin le plus proche, ou le nombre de voisins proches à essayer peuvent être choisit.

Il existe aussi des algorithmes non-constructifs comme le 2-opt, qui essaye d'améliorer un cycle donné en échangeant des sommets. Sa compléxité est en $O(n^2)$, mais comme l'ont montrés nos tests, l'appliquer une seule fois donne parfois une amélioration négligeable.

6.3 Algorithmes

6.3.1 Plus proche voisin

```
Entrée : g (Graphe complet)
Sortie : (coût, cycle) où cycle est un cycle hamiltonien construit selon la
    méthode du plus proche voisin et coût son coût associé sous forme de liste de
    points
coût = 0
cycle = ["un point de g au hasard"]

tant qu'il reste des points:
    # On ajoute au cycle le point suivant
    plus_proche = "point de g sur lequel on est pas encore passé le plus proche
        du dernier point du cycle"

    coût += "coût de plus_proche au dernier point du cycle"
    cycle = cycle :: plus_proche

# On ferme le cycle
coût += "coût du dernier au premier point de cycle"
cycle = cycle :: "premier point de chemin"

retourner (coût, cycle)
```

6.3.2 2-opt

```
Entrée : un cycle hamiltonien (liste de sommets) et son coût
Sortie : un cycle hamiltonien et son coût inférieur ou égal au coup d'entrée

pour chaque couple de points (a, b) dans le cycle:
    nouveau_coût = coût
                   − "coût de a à son successeur dans le cycle"
                   − "coût de b à son successeur dans le cycle"
                   + "coût de a à b"
                   + "coût du successeur de a et au successeur de b dans le
                       cycle"

    si nouveau_coût < coût:
        coût = nouveau_coût
        cycle = cycle crée en échangeant a et b dans cycle

retourner (coût, cycle)
```

6.4 Tests

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	Plus proche voisin amélioré + 2-opt
berlin52.tsp	7542	8981/19.1%	8060/6.7%	7972/5.7%	7810/3.6%
bier127.tsp	118282	137297/16.7%	125669/6.2%	127857/8.1%	122072/3.2%
d657.tsp	48912	62176/27.1%	N/A	N/A	N/A
u724.tsp	41910	55344, 32.1%	N/A	N/A	N/A
fl1577.tsp	22249	N/A	N/A	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.

L'application du 2-opt sur les résultats donnés par la méthode du plus proche voisin donne des résultats assez intéressants : ils sont très proches du résultat optimum. Malheuresement, cet algorithme est très coûteux en temps.

L'algorithme du 2-opt a été appliqué en boucle tant qu'il améliorait le résultat pour `berlin52.tsp` et `bier127.tsp`, mais pour le fichier `d657.tsp`, il était beaucoup trop long pour pouvoir faire ça, cependant, après 50 itérations (4h de calcul), on obtient un chemin de coût 58754, soit une erreur relative de 20.1%. L'algorithme est d'autant plus long que le cycle est déjà bon.

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.

7 Conclusion

Le python n'est évidemment pas adapté blabla

8 Annexes

Listings

Listing 1 – Classes pour représenter un graphe

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

class Edge:
    """
        Class that represents an edge as an origin vertice and a destination
        vertice and a cost.
    """
    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:
    """
        Represents a vertex as a set of edges.
    """
    def __init__(self, data):
        self.edges_out = set() # set of nodes that go out of that vertex
        self.data = data # an unique identifient for that vertex

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

    def degree(self):
        """
            Returns the degree of that vertex, that is, the number of edges
            that connect to it."
        """
        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):
        """
            Returns the cost to go from that node to the node other.
        """
        return self.edge_to(other).cost

    def edge_to(self, other):
        """
            Returns the edge from that vertex to the vertex other.
            Throw a RuntimeError if there is no such edge.
        """
        for edge in self.edges_out:
            if edge.other_side(self) == other:
                return edge
        raise RuntimeError("Not complete graph")

class Graph:
    """
        Represents a graph as a list of node.
    """
    def __init__(self, path=None):
        self.nodes = [] # nodes of the graph

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

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

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

def read_gph(path):
    """
        Construct a Graph from a file of the form:
            4 2 0
            1
            2
            3
            4
            1 2
            3 4
    """
    nodes_added = dict()
    g = Graph(path)

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

        for i in range(nb_v):
            data = int(f.readline())
            n = Node(data)
            g.nodes.append(n)
            nodes_added[data] = n

        for i in range(nb_e):
```

```
            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 g.oriented:
                n_dest.edges_out.add(edge)

    return g
```

Listing 2 – Codes relatifs à la connexité

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

def is_connected(graph):
    """
        Returns whether a graph is connected or not.
    """
    def visit(node, visited):
        visited.add(node)
        for e in node.edges_out:
            n = e.other_side(node)
            if n not in visited:
                visit(n, visited)

    if graph.order() <= 1: # useless cases
        return True

    if not graph.oriented:
        visited = set()
        x = next(iter(graph.nodes))
        visit(x, visited)
        return len(visited) == graph.order()
    else:
        raise NotImplementedError()
```

Listing 3 – Codes relatifs au graphes eulériens

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

from connected import *

def get_odd_vertices(graph):
    """
        Returns the number of nodes with odd number of vertices.
    """
    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):
    """
        Returns whether the graph is eulerian or not.
    """
    nb_odd_vertices=len(get_odd_vertices(graph))
    return nb_odd_vertices == 0 and is_connected(graph)

def is_semi_eulerian(graph):
    """
        Returns whether the graph is semi-eulerian but not eulerian or not.
    """
    nb_odd_vertices = len(get_odd_vertices(graph))
    return nb_odd_vertices == 2 and is_connected(graph)

def eulerian_path_euler(graph):
    """
        Returns an eulerian path og the graph or None if no such path exists.
    """
    def aux(node, visited_edges):
        result = [node]
        final_result = [node]

        while True:
            edges = [e for e in node.edges_out if e not in visited_edges]
            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 is_connected(graph):
        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 path_list_mul(list1, list2):
        result = []
        for i in list1:
            for j in list2:
                path = i[:]
                path.extend(j[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.append(path)
        return result;

    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:
                      result[i][j].extend(path_list_mul(cell_a,cell_b))
                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 4 – Codes relatifs au graphes hamiltonien

```
#!/usr/bin/python2
```

```python
# -*- coding: utf-8 -*-
import graphs

def dirac_test(graph):
    """
        Dirac's theorem.
    """
    return len(graph.nodes) < 3 and min(node.degree() for node in graph.nodes) >=
        graph.order() / 2

def posa_test(graph):
    """
        Pósa's theorem.
    """
    n = len(graph.nodes)
    if n < 3: return False

    k = int((n+1)/2)
    degrees = [0 for i in range(0, k)]
    for node in graph.nodes:
        for d in range(0, node.degree()+1):
            if d < k:
                degrees[d] += 1

    for i in range(0, k):
        if degrees[i] > i:
            return False

    return True

def is_semi_hamiltonian(graph):
    """
        Returns whether a graph is hamilonian or not.
        The graph must be a simple graph and non-oriented.
    """
    # rapid tests, only sufficient
    if dirac_test(graph) or posa_test(graph):
        return True

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

def hamiltonian_path(graph, node_from=None, nodes_done=frozenset()):
    """
        Return a hamiltinian path if one exists or None.
        This is brute force.
        The graph must be non-oriented.
    """
    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()):
    """
        Return a hamiltinian path if one exists or None.
        This is slitghly more efficient than brute force as it tries to stop
            sonner.
        The graph must be non-oriented.
    """
    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):
    """
        Create a graph from a file of the form:
            [3 useless lines]
            DIMENSION : 1000
            [2 useless lines]
            node1 node2
            node3 node4
            ...
            -1
        These graphs are found here
            http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/
    """
    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(path)
        g.nodes = nodes
        g.oriented = False
        return g
```

Listing 5 – Codes relatifs postier chinois

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

def dijkstra_min_cost(origin, dest):
    '''
    Retourne le coût minimum pour aller de origin vers dest.
    précondition: le graphe est connexe
    '''
    if origin is dest:
        return 0

    reachable_nodes = [(e.cost, e.other_side(origin)) for e in origin.edges_out]
    reachable_nodes.sort(key=lambda n: n[0])

    while True:
        cost, node = reachable_nodes.pop(0)
        if node is dest:
            return cost

        for edge in node.edges_out:
            node_reachable = edge.other_side(node)
            reachable_nodes.append((cost + edge.cost, node_reachable))

        reachable_nodes.sort(key=lambda n: n[0])

def dijkstra_min_cost_path(origin, dest):
    '''
    Retourne le chemin (liste d'arêtes) de coût minimum pour aller de origin vers
        dest.
    précondition: le graphe est connexe
    '''
    if origin is dest:
        return []

    reachable_nodes = [(e.cost, [e], e.other_side(origin)) for e in
        origin.edges_out]
    reachable_nodes.sort(key=lambda n: n[0])

    while True:
        cost, path, node = reachable_nodes.pop(0)
        if node is dest:
            return path
```

```python
        for edge in node.edges_out:
            node_reachable = edge.other_side(node)
            reachable_nodes.append((cost + edge.cost, path + [edge],
                node_reachable))

        reachable_nodes.sort(key=lambda n: n[0])

def postier_chinois(g):
    '''
    Retourne le chemin optimal pour le problème du postier chinois, ou None s'il
        n'y a pas de chemin.
    Attention : l'algorithme peut modifier le graphe.
    '''
    if not g.is_connected():
        return None

    if g.oriented:
        raise NotImplementedError()

    # Création du graphe partiel
    pg = Graph()
    pg.oriented = False

    # Copie des nœuds de degré impaire
    for node in g.nodes:
        if len(node.edges_out) % 2 == 1:
            pg.nodes.append(Node(node.data))

    if not pg.nodes: # graphe eulérien
        return eulerian_path_euler(g)

    # Copie des arêtes
    pg_nodes = set(pg.nodes)
    while pg_nodes:
        node_pg = pg_nodes.pop()
        node = filter(lambda n: n.data == node_pg.data, g.nodes)[0]
        for edge in node.edges_out:
            other_side_pg = filter(lambda n: n.data ==
                edge.other_side(node).data, pg_nodes)
            if other_side_pg:
                other_side_pg = other_side_pg[0]
                edge_pg = Edge(node_pg, other_side_pg, edge.cost)
                node_pg.edges_out.add(edge_pg)
                other_side_pg.edges_out.add(edge_pg)

    # Transformation en clique
    pg_nodes = set(pg.nodes)
    while pg_nodes:
        node_pg = pg_nodes.pop()
        for node in pg_nodes:
            if not node_pg.exists_edge_to(node):
                # Récuperation des nœuds dans le graphe initial
                node_pg_g = filter(lambda n: n.data == node_pg.data, g.nodes)[0]
                node_g = filter(lambda n: n.data == node.data, g.nodes)[0]

                # Création de l'arête
                edge = Edge(node_pg, node, dijkstra_min_cost(node_pg_g, node_g))
                node_pg.edges_out.add(edge)
                node.edges_out.add(edge)

    # Recherche du couplage parfait de coût minimum
    edges = set() # ensemble des arêtes
```

```
    for node_pg in pg.nodes:
        edges.update(node_pg.edges_out)

    def aux(matching, nodes, cost):
        if len(nodes) == len(pg.nodes):
            return matching, cost

        best_matching, best_cost = None, 0

        for edge in edges:
            if edge.origin not in nodes and edge.dest not in nodes:
                matching_copy = matching[:]
                matching_copy.append(edge)
                nodes_copy = set(nodes)
                nodes_copy.add(edge.origin)
                nodes_copy.add(edge.dest)

                result_matching, result_cost = aux(matching_copy, nodes_copy,
                    cost + edge.cost)
                if best_matching is None or best_cost > result_cost:
                    best_matching, best_cost = result_matching, result_cost

        return best_matching, best_cost

    best_matching, best_cost = aux([], set(), 0)

    # On double les arêtes dans best_matching
    for edge_pg in best_matching:
        origin = filter(lambda n: n.data == edge_pg.origin.data, g.nodes)[0]
        dest = filter(lambda n: n.data == edge_pg.dest.data, g.nodes)[0]
        path = dijkstra_min_cost_path(origin, dest)

        for edge in path:
            new_edge = Edge(edge.origin, edge.dest, edge.cost)
            edge.origin.edges_out.add(new_edge)
            edge.dest.edges_out.add(new_edge)

    return eulerian_path_euler(g)
```

Listing 6 – Codes relatifs au TSP

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

import graphs
import heapq
import random

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
```

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

Listing 7 – Tests

```
#!/usr/bin/python2
# -*- coding: utf-8 -*-
from graphs import *
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, semi-hamiltonian]
    graphs.append([read_gph('tests/1.gph'), True, True, False, True])
    graphs.append([read_gph('tests/2.gph'), True, False, True, False])
    graphs.append([read_gph('tests/3.gph'), False, False, False, False])
    graphs.append([read_gph('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([read_gph('tests/complete.gph'), True, True, False, True])
    graphs.append([read_gph('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 semi hamiltoniannité
    test_one(graphs, is_semi_hamiltonian, 4)
```