TP Optimisation – session 4 – Algorithmes d'approximation

1er janvier 2025

Objectif d'apprentissage : cette séance vise à concevoir des instances pathologiques pour des algorithmes d'approximation vus en cours. On combinera deux approches :
— la réflexion issue de l'analyse de l'algorithme et de ses limites ;
— l'exploration empirique basée sur de la génération aléatoire.
C'est en combinant ces deux approches que l'on maximise les chances de trouver la pire instance pathologique.

> Les deux premières sections sont essentielles.

1 Couverture gloutonne

Le professeur Sourire propose l'heuristique suivante pour résoudre le problème de la couverture de sommets. On choisit de façon répétée un sommet de plus haut degré, et on supprime toutes ses arêtes incidentes. On s'arrête lorsqu'il n'y a plus d'arêtes. On souhaite construire une instance dont le facteur d'approximation soit le plus large possible.

Pour cela, on commence par créer un graphe quelconque qui facilitera l'implémentation de l'heuristique en fournissant la structure sur laquelle on travaille :

```
import networkx as nx
G = nx.Graph()
G.add_nodes_from([1, 2, 3, 4])
print(G.nodes())
G.add_edges_from([(1, 2), (2, 3), (1, 3), (3, 4)])
print(G.edges())
print(G[3])
```

On va maintenant implémenter deux méthodes :
— `verifier_couverture` qui retourne vrai si une couverture donnée est valide pour un graphe donné ;
— `heuristique_Sourire` qui génère une couverture pour un graphe.
Vous aurez peut-être besoin de la méthode `copy` pour copier le graphe.

```
def verifier_couverture(G, C):
    GG = G.copy()
    GG.remove_nodes_from(C)
    return nx.number_of_edges(GG) == 0

def heuristique_Sourire(G):
    GG = G.copy()
    C = []
    while nx.number_of_edges(GG) != 0:
        node = max(GG.nodes, key=lambda v: len(GG[v]))
```

```
        C.append(node)
        GG.remove_node(node)
    return C

print(verifier_couverture(G, [1, 2, 4]))
print(heuristique_Sourire(G))
```

On va désormais construire une instance pour laquelle l'heuristique du professeur Sourire est arbitrairement mauvaise. L'instance aura une structure de graphe biparti avec n sommets à gauche. Ensuite, pour tout $2 \leq i \leq n$: on rajoute $k = \lfloor n/i \rfloor$ sommets à droite de degré i, tous connecté aux sommets de gauche qui ont le plus petit degré.

On peut vérifier que l'ensemble des sommets de gauche constitue une couverture et que l'heuristique retourne une couverture de taille supérieure avec un ratio qui augmente avec n.

```
def construire_instance(n):
    G = nx.Graph()
    G.add_nodes_from(range(n))
    for i in range(2, n + 1):
        k = n // i
        for _ in range(k):
          right = len(G)
          G.add_node(right)
          for _ in range(i):
              left = min(range(n), key=lambda i: len(G[i]))
              G.add_edge(left, right)
    return G

n = 10
G = construire_instance(n)
c_sourire = heuristique_Sourire(G)
print(verifier_couverture(G, range(n)))
print(verifier_couverture(G, c_sourire))
print(f"{len(c_sourire)} {n}")
```

Pour quelle valeur de n obtient-on un facteur qui dépasse 2 ?

Pour $n = 16$:

```
for i in range(3, 20):
    G = construire_instance(i)
    c_sourire = heuristique_Sourire(G)
    print(f"{i} {len(c_sourire) / i}")
```

2 Tournée-VC-Approchée

L'approche précédente consiste à concevoir spécifiquement une instance qui exploite une faille de l'heuristique. Une autre approche consiste à générer aléatoirement des instances pour étudier le comportement.

Nous allons maintenant générer des graphes complets pondérés pour le problème du voyageur de commerce afin déterminer s'il existe une instance dont le facteur d'approximation dépasse le 3/2 qui a été trouvé en TD.

Il faudra calculer l'optimal pour évaluer la qualité de ce que retourne l'algorithme d'approximation. Cette recherche ne sera donc valable que pour des petites instances (4 à 8 sommets).

On va préparer la rechercher en codant 3 parties : la construction d'un graphe ; le calcul de la tournée optimale ; l'implémentation de l'algorithme d'approximation.

Pour la construction du graphe, on utilisera la méthode suivante : on part d'un graphe complet avec une pondération unitaire sur chaque sommet. On sélectionne alors une arête au hasard pour augmenter sa pondération d'une unité si cela ne viole pas les inégalités triangulaires.

Le code suivant génère un graphe complet avec une pondération unitaire :

```
G = nx.complete_graph(n)
for e in G.edges():
    G[e[0]][e[1]]["weight"] = 1
```

Vous choisir une arête aléatoire, vous pourrez utiliser l'approche suivante :

```
edge = random.sample(list(G.edges()), 1)[0]
```

```
import random

def inegalite_triangulaire(G):
    for i in G:
        for j in G:
            for k in G:
                if i == j or j == k or i == k:
                    continue
                if G[i][j]["weight"] > G[i][k]["weight"] + \
                        G[k][j]["weight"]:
                    return False
    return True

def construire_graphe(n, M):
    G = nx.complete_graph(n)
    for e in G.edges():
        G[e[0]][e[1]]["weight"] = 1
    while M != 0:
        edge = random.sample(list(G.edges()), 1)[0]
        G[edge[0]][edge[1]]["weight"] += 1
        if not inegalite_triangulaire(G):
            G[edge[0]][edge[1]]["weight"] -= 1
        else:
            M -= 1
    return G
```

On peut désormais calculer la tournée optimale. Il faudra d'abord écrire le code d'une fonction pour évaluation le coût d'une tournée. Ensuite, on évaluera toutes les tournées et on choisira la meilleure. Pour générer toutes les tournées, on peut s'appuyer sur la bibliothèque `itertools` :

```
import itertools
itertools.permutations(G)
```

```python
def eval_tournee(G, t):
    dist = 0
    for i in range(len(t)):
        dist += G[t[i - 1]][t[i]]["weight"]
    return dist

def tournee_VC_optimal(G):
    return min(itertools.permutations(G), key=lambda t: eval_tournee(G, t))
```

Il ne reste plus qu'à implémenter Tournée-VC-Approchée.

```python
def tournee_VC_approchee(G):
    nodes = list(G)
    t = [nodes.pop()]
    while nodes:
        i, j = min(((i, j) for i in range(len(t))
                           for j in range(len(nodes))),
                           key=lambda x: G[t[x[0]]][nodes[x[1]]]["weight"])
        t = t[:i + 1] + [nodes[j]] + t[i + 1:]
        nodes = nodes[:j] + nodes[j + 1:]
    return t
```

On peut commencer la recherche empirique. Il faut tâtonner en gérant les paramètres numériques suivant :

— la taille du graphe ;
— le nombre de graphes générés ;
— la quantité de poids rajoutée lors de la génération aléatoire.

Quelle est l'instance la plus petite qui dépasse 3/2 ?

```python
for n in range(4, 7):
    for M in range(1, 10):
        ratio = []
        for i in range(100):
            G = construire_graphe(n, M)
            opt = tournee_VC_optimal(G)
            approx = tournee_VC_approchee(G)
            O = eval_tournee(G, opt)
            A = eval_tournee(G, approx)
            ratio.append(A / O)
        print(n, M, max(ratio))
```

On obtient 8/5 avec 5 sommets et 3 poids à 2. On a un poids à 2 sur les arêtes $(0, 1)$, $(1, 2)$ et $(2, 3)$. Comme on commence par le seul sommet qui n'a que des arrêtes de poids unitaire (il est le plus proche de tous les autres sommets), on rajoute les autres sommets les uns après les autres. La tournée optimale est $(0, 2, 4, 1, 3)$ alors que Tournée-VC-Approchée trouve $[4, 3, 2, 1, 0]$.

On trouve aussi 5/3 avec 6 sommets et 4 poids incrémentés et même 12/7 avec 7 sommets et 5 poids incrémentés. En effet, le principe se généralise à n sommets et on obtient le facteur $2\frac{n-1}{n}$.

3 Couverture-Ensemble-Glouton

On peut procéder de même pour le problème de la couverture d'ensemble afin de trouver une instance qui atteint un facteur d'approximation supérieur d'au moins 3 pour Couverture-Ensemble-Glouton.