Faculté des Sciences et Ingénierie - Sorbonne université

Master Informatique parcours - DAC / ANDROIDE

COMPLEX - Complexité, algorithmes randomisés et approchés

Rapport de projet

Algorithme de Karger

Réalisé par :
BENHADDAD Sabrina - M1 DAC
SLIMANI Nour Ismahane - M1 ANDROIDE

Décembre 2022

Table des matières

Table des figures

Introduction

En théorie des graphes, une coupe d'un graphe est une partition des sommets en deux sous-ensembles disjoints.
On appelle cardinal d'une coupe l'ensemble des arêtes ayant une extrémité dans chaque sous-ensemble de la partition. Une coupe minimum d'un graphe est donc, une coupe contenant un nombre minimal d'arêtes.

Il peut y avoir pour un même graphe, plusieurs coupes minimums, c'est-à-dire plusieurs coupes différentes mais de même cardinal : le cardinal minimum.
Le problème de la coupe minimum consiste alors, étant donné un graphe, à déterminer une coupe minimum.
Pour ce faire, l'une des solutions étudiées en cours est l'algorithme de Karger.

L'algorithme de Karger est un algorithme probabiliste qui permet de déterminer une coupe minimum d'un graphe non orienté à partir d'une méthode de contraction d'arête.
Cette procédure consiste, étant donné un graphe G et une arête aléatoire $e = (u, v)$, à fusionner les sommets u et v pour former un unique sommet uv relié à la fois aux voisins de u et aux voisins de v (en évitant les boucles).
L'algorithme consiste simplement à répéter l'opération de contraction plusieurs fois sur le graphe jusqu'à n'avoir que deux sommets. Les deux sommets représentent la partition de sommets, et les arêtes entre eux deux sont les arêtes de la coupe.

Notre travail consiste donc à évaluer la complexité de l'algorithme de Karger ainsi que de ses améliorations à travers trois parties distinctes que voici :

— Implémentation et analyse de l'algorithme de Karger sur deux structures de données de graphes

— Amplification du succès de l'algorithme en étudiant la complexité et probabilité de succès de l'algorithme de Karger itéré

— Étude de l'algorithme de Karger-Stein

L'objectif étant de répondre à toutes les questions de l'énoncé du projet à travers ce rapport ainsi que le code joint.

Chapitre 1

Exercice 1 - Algorithme de Karger

Commençons par rappeler tout d'abord l'algorithme de Karger vu en cours :

Algorithme 1 : Algorithme de Karger

Entrée : $G = (V, E)$ un graphe non orienté avec $n = \#V$ et $m = \#E$
Sortie : S une coupe de G
tant que $\#V > 2$ **faire**
  $e \xleftarrow{\text{🎲}} E$
  $G \leftarrow G/e$
$\{v_1, v_2\} \leftarrow V$ ▷ Il reste deux sommets dans V
retourner $S = \{$ sommets qui « apparaissent » dans v_1 $\}$

Au cours de l'exécution de cet algorithme, le graphe G donné en entrée peut être transformé en un multi-graphe.
Nous rappelons qu'un multi-graphe par définition, est un graphe doté d'une ou plusieurs arêtes multiples ou de boucles.
Il est donc nécessaire de pouvoir exécuter cet algorithme sur un multi-graphe dans notre implémentation.

Question 1.A

Commençons par programmer une structure de donnée permettant de représenter un multi-graphe par sa matrice d'adjacence.
A noter qu'une matrice d'adjacence n'est rien d'autres qu'une matrice carrée dont l'élément non diagonal a_{ij} est le nombre d'arêtes liant le sommet i au sommet j.

Pour ce faire, nous avons implémenté la classe **matriceadj** qui dispose comme attributs : une taille n, une matrice (ici la matrice d'adjacence de taille $n * n$ et enfin un ensemble de sommets représenté par une liste de listes (nous avons choisi ce type de données pour pouvoir représenter la contraction des sommets par la suite).
A noter que nous avons également implémenter la méthode ajout_arete permettant de remplir la matrice d'adjacence.

```
class matriceadj :
    #Nous allons initialiser la matrice
    def __init__(self,taille):
        self.taille = taille
        self.adjMatrice = np.zeros((taille,taille))
        self.ens_sommets = [] #à cause de la contrac
        for i in range (taille):
            self.ens_sommets.append([i])
            #On programme une méthode permettant d'a
    def ajout_arete(self,u,v):
        self.adjMatrice[u,v] +=1
        self.adjMatrice[v,u] += 1
```

FIGURE 1.1 – Implémentation de la structure de donnée représentant une matrice d'adjacence

En ce qui concerne la fonction de contraction, cette dernière prend tout d'abord en entrée un multi-graphe $G = (V, E)$ non orienté représenté ici par sa matrice d'adjacence, et retourne en sortie une coupe de G en suivant les étapes ci-dessous :

- Choisir aléatoirement une arête (u, v), nous fixerons pour l'instant l'arête à contracter (le tirage aléatoire sera effectué au cours des questions suivantes).
- Supprimer l'arête choisie en modifiant la matrice d'adjacence.
- Supprimer les boucles (elles ne sont d'aucune utilité dans notre algorithme).
- Contracter l'ensemble des sommets
- Retourner la matrice d'adjacence ainsi que l'ensemble des sommets associés à la coupe du graphe.

```
def contraction(self,u,v):
    self.adjMatrice[v] += self.adjMatrice[u]
    self.adjMatrice[:,v] += self.adjMatrice[:,u]
    self.adjMatrice[v,v] = 0 #On supprime les boucles
    self.adjMatrice = np.delete(self.adjMatrice,u,0)
    self.adjMatrice = np.delete(self.adjMatrice,u,1)
    #Passons au merge des sommets
    merge_sommets = self.ens_sommets[v]
    for s in merge_sommets :
        self.ens_sommets[u].append(s)
    self.ens_sommets.pop(v)
    return self.adjMatrice, self.ens_sommets
```

FIGURE 1.2 – Implémentation de la fonction de contraction

Question 1.B

Nous avons vu dans le cours, que l'opération de contraction d'arête prend un temps $O(n)$ en utilisant une représentation du multi-graphe G par matrice d'adjacence.
Testons de ce fait, notre fonction sur différentes familles de graphe, que voici :

Cycles de taille n

Un cycle de taille n en théorie des graphes, est une chaîne dont l'extrémité initiale est égale à l'extrémité finale (pour simplifier nos implémentations, nous nous intéresserons aux cycles élémentaires à n sommets)

Graphes complets à n sommets

Un graphe complet est un graphe simple dont tous les sommets sont adjacents deux à deux, c'est-à-dire que tout couple de sommets disjoints est relié par une arête.

Graphes bipartis complets à $n = 2k$ sommets

Un graphe biparti complet est un graphe biparti dont chaque sommet du premier ensemble est relié à tous les sommets du second ensemble.
Si le premier ensemble U est de cardinal m et le second ensemble V est de cardinal n, le graphe biparti complet est noté $K_{m,n}$.

```
def mat_cycle(taille):#La fonction prend en ar
    matrix = matriceadj(taille)#Pour initialis
    matrix.ajout_arete(0,taille-1)
    for i in range(taille-1):
        matrix.ajout_arete(i,i+1)
    return matrix

#Graphe complet a n sommets
def mat_complet(taille):#La fonction prend en
    matrix = matriceadj(taille)#Pour initialis
    #Comme tous les sommets du graphe sont rel
    for i in range(taille):
        for j in range(taille):
            if j!= i and i <j:#pour ajouter qu
                matrix.ajout_arete(i,j)
    return matrix

#Graphe biparti complet n = 2k
def mat_bicomplet(taille):
    if (taille % 2 == 0):#La fonction prend en
        matrix = matriceadj(taille)#Pour initi
        for i in range(taille //2):
            for j in range(taille//2,taille):
                matrix.ajout_arete(i,j)
        return matrix
```

FIGURE 1.3 – Implémentation des familles de graphes

Une fois les familles de graphes implémentées, passons à l'objectif de cette question à savoir : l'analyse de la complexité expérimentale de notre fonction de contraction.

Pour ce faire, nous allons pour les trois familles de graphes ci-dessus, exécuter la fonction de contraction pour des arêtes données et sur des données croissantes (tailles =

$[50, 100, 150, 200, 250, 300, 350, 400, 450, 500]$) puis, mesurer le temps d'exécution.
L'évolution des temps de calcul en fonction de la taille des données est représenté comme suit :

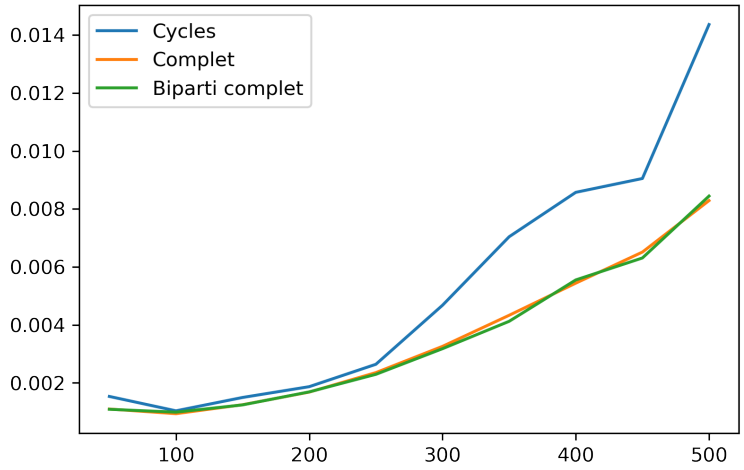

FIGURE 1.4 – Complexité expérimentale pour la fonction de contraction

Nous pouvons aisément constater que la fonction de contraction a une tendance d'exécution en temps linéaire.
Cela s'explique assez facilement vu l'implémentation de notre algorithme qui a clairement une complexité en $O(n)$ (présence de boucles simples avec une taille polynomiale de l'entrée).

Question 1.C

Une fois que la première opération de contraction a eu lieu dans l'algorithme de Karger, le graphe G est possiblement devenu un multi-graphe avec plusieurs arêtes entre deux sommets.
Dans notre analyse théorique de la complexité de l'algorithme de Karger, nous avons supposé que l'arête e de l'opération de sélection aléatoire dans la boucle **tant que** est tirée uniformément aléatoirement parmi toutes les arêtes possibles.
Lorsque le multi-graphe G est représenté par sa matrice d'adjacence, la méthode du tirage aléatoire est en réalité relativement simple.
En effet étant donné que nous travaillons avec des matrices d'adjacence, pour tirer une arête aléatoirement, nous allons tout simplement tirer aléatoirement deux sommets formant une arête (c'est à dire dont la valeur $a_{ij} = 1$ ou plus dans notre matrice d'adjacence).
Puisqu'en effet chaque arête (u, v) est représentée dans une matrice d'adjacence par l'intersection d'une ligne u et d'une colonne v tel que a_{ij} représente le degré de multiplicité de l'arête en question.

A noter que la fonction de ce tirage aléatoire prend en entrée la matrice d'adjacence associée au multi-graphe G et retourne en sortie deux sommets u et v formant une arête (n'étant pas une boucle).

Cette implémentation est définie comme suit :

```
def tirage_arete(self):
    u = rd.randint(0,len(self.adjMatrice)-1)
    v = rd.randint(0,len(self.adjMatrice)-1)
    while(u==v or self.adjMatrice[u,v]==0): #Nous
        u = rd.randint(0,len(self.adjMatrice)-1)
        v = rd.randint(0,len(self.adjMatrice)-1)
    return u,v
```

FIGURE 1.5 – Implémentation de la fonction du tirage aléatoire

Question 1.D

En utilisant les fonctions implémentées au cours des questions précédentes, nous pouvons terminer une première implémentation de l'algorithme de Karger en utilisant une représentation par matrice d'adjacence, que voici :

```
def karger(self):
    u,v = self.tirage_arete()
    print("\nL'arête tirée est entre les sommets:",u,v)
    m,s = self.contraction(u,v)
    print("Les sommets du graphe apres la contraction :",s)
    print("La matrice d'adjacence apres la contraction :\n",m)
    while(len(m)>2):
        self.adjMatrice = m
        self.ens_sommets = s
        u,v = self.tirage_arete()
        print("\nL'arête tirée est entre les sommets :",u,v)
        m,s = self.contraction(u,v)
        print("La matrice d'adjacence apres la contraction :\n",m)
        print("Les sommets du graphe apres la contraction :",s)
    return m,s
```

FIGURE 1.6 – Implémentation de l'algorithme de Karger (représentation du graphe par matrice d'adjacence)

Analyser la complexité expérimentale obtenue pour les familles de graphes étudiées

Nous ajouterons dans cette partie une nouvelle famille de graphes : les graphes aléatoires.

Il s'agit tout simplement de graphes générés aléatoirement à n sommets où une arête est ajoutée à E l'ensemble des arêtes du graphe avec une probabilité p. Nous l'avons implémentée comme ceci :

```
def mat_alea(size, p):
    mat = matriceadj(size)
    for i in range(size):
        for j in range(size):
            if(i<j):#pour ajouter qu'une se
                _p = np.random.uniform(0,1)
                if(_p < p):
                    mat.ajout_arete(i,j)
    return mat
```

FIGURE 1.7 – Implémentation d'une famille de graphes aléatoires

Une fois l'ensemble des familles des graphes implémentées et définies, nous pouvons à présent passer à l'analyse de la complexité expérimentale pour chacune d'entre elles.

En d'autres termes, nous allons pour chaque famille de graphes :

— Exécuter l'algorithme de Karger sur des données de plus en plus grandes (dans notre cas $n = [50, 60, 70, 80, 100, 110, 120, 130, 140, 150]$)

— Mesurer les temps d'exécution

— Construire la courbe d'évolution de ces temps de calcul en fonction de la taille des données

Nous nous retrouvons ainsi avec les résultats suivants :

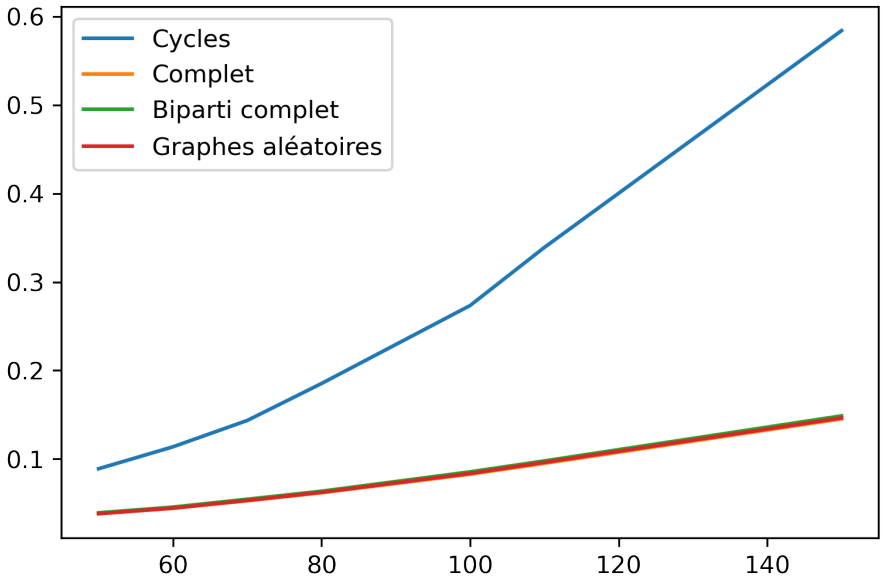

FIGURE 1.8 – Complexité expérimentale obtenue pour l'algorithme de Karger

Nous pouvons remarquer que plus la taille des données augmente, plus l'algorithme a une temps d'exécution à tendance quadratique.
Nous pouvons l'observer plus particulièrement dans le cas des cycles de taille k dont la courbe s'approche fortement de la fonction quadratique.
A noter qu'il est possible d'obtenir une meilleure observation de la tendance quadratique de l'algorithme en utilisant des tailles de données beaucoup plus grandes que celles utilisées. (Par soucis d'ordinateur, nous n'avons pas pu étudier l'algorithme pour des tailles très grandes contrairement à l'analyse menée en question 1.b).

Comparer avec l'analyse théorique vue en cours :

Comme nous l'avons pu voir en cours, l'algorithme de Karger effectue exactement $n-2$ répétitions de la boucle **tant que** tel que chaque opération de contraction peut être réalisée en $O(n)$ opérations si le graphe est stocké par matrice d'adjacence.
Ce qui veut dire entre autres, que la complexité issue de l'analyse théorique de l'algorithme de Karger est de $O(n^2)$ opérations sur la matrice d'adjacence.
En ce qui concerne notre algorithme implémenté, nous pouvons voir que ce dernier effectue lui aussi, exactement $n-2$ répétitions de la boucle **tant que** avec un temps d'exécution en $O(n)$ de la fonction de contraction (démontrée en question 1.B).
A noter que les écarts entre le temps théorique d'exécution et le temps pratique de l'algorithme peut effectivement provenir du choix aléatoire des arêtes.
Le temps d'exécution de la fonction tirage_arete dépend effectivement fortement du tirage aléatoire réalisé, il peut être au pire des cas en $O(n)$ (si l'arête tirée est un boucle ou une arête qui n'existe pas dans le graphe, l'algorithme rentre dans la boucle **tant que** et n'en sort que si les deux conditions sont vérifiées).

Ainsi d'après les résultats obtenus, nous pouvons constater que notre implémentation de l'algorithme de Karger par matrice d'adjacence est relativement bonne.

Question 1.E

L'objectif de tout ce qui suit, est de proposer une implémentation de l'algorithme de Karger en utilisant une autre représentation des multi-graphes, à savoir les **listes d'adjacences**.

Rappel :

Une liste d'adjacence est une structure de données utilisée pour représenter un graphe. Dans le cas des graphes non-orientées traités dans ce projet, il s'agit tout simplement de la liste des voisins de chaque sommet.

Son implémentation est en fait, un dictionnaire défini comme suit :

```
class listadj:
    def __init__(self,Nodes,is_directed=False):
        self.nodes=Nodes
        self.adj_list={}
        self.is_directed=is_directed

        for node in self.nodes:
            self.adj_list[node]=[]

    def add_edge(self,v,e):
        self.adj_list[v].append(e)
        if not self.is_directed:
            self.adj_list[e].append(v)
```

FIGURE 1.9 – Implémentation de la structure de donnée représentant une liste d'adjacence

En ce qui concerne la fonction de contraction, le principe est exactement le même que précédemment, la seule différence étant la structure de donnée manipulée.

La liste d'adjacence étant représentée sous forme de dictionnaire et non pas sous forme de matrice, nous allons suivre les étapes suivantes afin de réussir l'opération de contraction :

— Concaténer la liste des voisins des sommets u et v

— Supprimer les sommets u et v de la liste résultante (l'objectif étant de supprimer les boucles)

— Supprimer le sommet v

— Remplacer toutes les occurrences de v par u

Ce qui nous donne la fonction suivante :

```
def changer_sommet(adjlist,node,v,u):
    adjlist[node]= [u if n == v else n for n in adjlist[node] if n != node]
#Fonction qui réalise la contraction
def contractlist(adjlist,u,v):
    arete_de_v = [n for n in adjlist[v] if n!= u]
    adjlist[u] = [n for n in adjlist[u] if n!= v]
    adjlist[u] += arete_de_v
    adjlist.pop(v)
    for node in list(set(arete_de_v)):
        changer_sommet(adjlist, node, v, u)
    return adjlist
```

FIGURE 1.10 – Fonction de contraction pour les listes d'adjacence

Question 1.F

Étant donné que nous manipulons des listes d'adjacence qui sont des dictionnaires, nous allons commencer ,pour tirer aléatoirement une arête (u, v), par tirer aléatoirement un sommet u parmi les clés du dictionnaire.
Après avoir tiré aléatoirement un sommet u , nous allons aléatoirement choisir un de ses voisins parmi donc les valeurs de la clé u (on utilisera les fonctions de bibliothèque *random* comme pour la question 1.C)

Question 1.G

À l'aide des questions précédentes, il est désormais facile de définir l'algorithme de Karger en utilisant une représentation des graphes par liste d'adjacence.
Ce dernier est ainsi composé d'une boucle **tant que** qui exécute ,tant que le nombre de sommets restants dans le graphe est supérieur à 2, le tirage aléatoire d'une arête ainsi que la fonction de contraction.
La sortie de cette autre version de Karger n'est rien d'autres qu'un dictionnaire possédant 2 clés (représentant le nombre de sommets finaux dans le graphe) tel que chaque clé dispose d'un certain nombre identique de voisins. Il s'agit bien évidemment de la coupe minimum retournée.

En analysant la complexité théorique de l'algorithme implémenté nous pouvons clairement voir que l'algorithme s'exécute au moins en $O(n^2)$.
En effet étant donné que :

— La taille de l'entrée de l'algorithme est polynomiale (il s'agit de la liste d'adjacence

— La fonction de contraction s'exécute en $O(n)$

— Karger s'exécute $(n-2)$ fois

Alors, nous pouvons déduire que la complexité théorique de l'algorithme de Karger est au moins quadratique.
A noter que l'utilisation d'une liste d'adjacence améliore la complexité spatiale de notre algorithme.
La liste d'adjacence disposant en effet d'une complexité spatial polynomiale $O(|S| + |A|)$ contrairement à la matrice d'adjacence qui dispose d'une complexité spatiale en $O(n^2)$

Question 1.H

Analyse de la complexité expérimentale obtenue avec les familles de graphes représentées par liste d'adjacence

Pour répondre à cette question, nous allons dans un premier temps adapter la définition des familles de graphes définies plus haut, pour les représenter cette fois-ci par leur liste d'adjacence.

Nous allons par la suite, tester notre nouvelle implémentation de Karger sur ces familles de graphes avec des tailles croissantes pour calculer et évaluer le son temps d'exécution (ce sont les mêmes étapes que la question 1.D).
Nous obtenons ainsi, les résultats suivants :

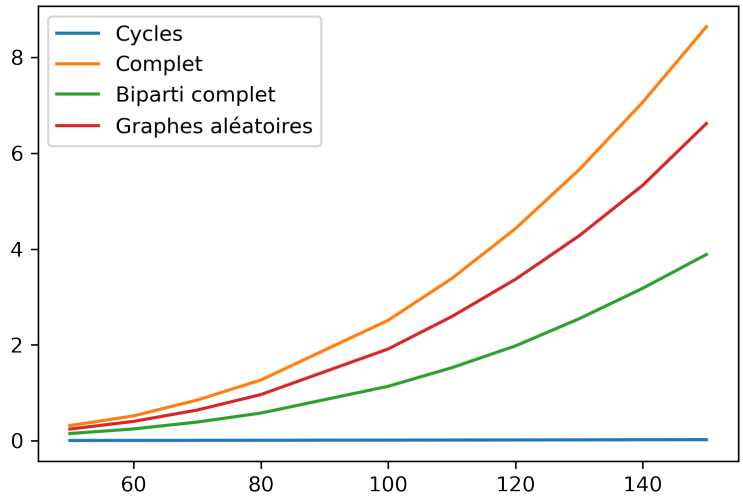

FIGURE 1.11 – Complexité expérimentale obtenue pour la nouvelle implémentation de Karger

Ainsi comme nous pouvons le voir, l'exécution de l'algorithme de Karger pour des graphes représentés par leur liste d'adjacence a une tendance assez quadratique, nous pouvons notamment le remarquer dans le cas des graphes complets par exemple.
Nous aurions pu toutefois disposer de meilleurs résultats et de meilleures courbes avec des données plus grandes.

Ainsi nous pouvons voir que notre implémentation de l'algorithme de Karger est relativement bonne et dispose d'une complexité assez conforme par rapport aux résultats théoriques obtenus en cours.

Chapitre 2

Exercice 2 - Amplification du succès de l'algorithme

Comme nous l'avons vu en cours, l'algorithme de Karger est un algorithme probabiliste de type Monte-Carlo et peut retourner une coupe du graphe qui n'est pas minimum. Dans tout ce qui suit, nous utiliserons la représentation de l'algorithme de Karger par sa matrice d'adjacence.

Question 2.A

Étude expérimentale de la probabilité de succès de l'algorithme de Karger implémenté précédemment sur les familles de graphes définies dans l'exercice 1 :

Dans cette question, nous allons considéré trois familles de graphes uniquement car nous connaissons la taille de leur coupe minimum, que voici :

— Un graphe complet de taille 5 dont on sait que la taille de sa coupe minimum est 4.

— Un graphe biparti complet $K_{3,3}$ dont on sait que la taille de sa coupe minimum est le minimum des cardinaux des deux sous-ensembles, ici 3.

Nous n'étudierons pas les cycles de taille n car quelque soit l'arête tirée la taille de sa coupe minimum restera toujours identique (elle est égale à 2). Ainsi que les graphes aléatoires car nous ne connaissons pas la taille de leur coupe minimum (vue leur nature aléatoire).
A noter que dans cette expérimentation nous allons volontairement fixer la taille des graphes étudiés (nous devons effectivement connaître la taille de leur coupe minimum) puis, exécuter Karger un certain nombre de fois sur le même graphe afin de calculer la probabilité de succès.
Ainsi en testant notre algorithme sur les instances de graphes précisées plus haut, nous obtenons une probabilité de succès pour le K_5 de 0.48 et une probabilité de succès pour le biparti complet $K_{3,3}$ est de 0.41

Pour aller plus loin, nous avons souhaité tester cette probabilité de succès pour plusieurs tailles de graphes bipartis complets, étant donné que nous connaissons la taille de leur coupe minimum.
Nous obtenons ainsi les résultats suivants :

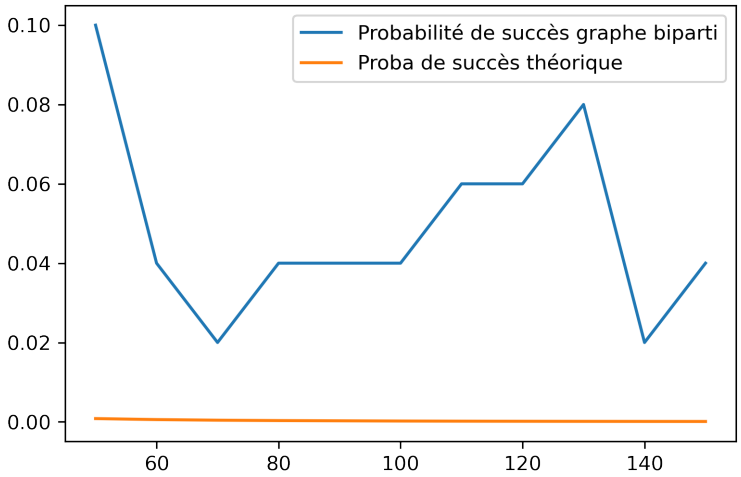

FIGURE 2.1 – Étude expérimentale de la probabilité de succès de Karger sur des graphes bipartis complets

Comparaison avec l'analyse théorique :

Comme vu en cours, la probabilité de succès de l'algorithme de Karger doit toujours être $\geq \frac{2}{n(n-1)}$
Dans notre expérimentation, nous avons volontairement affiché dans le graphe ci-dessus cette borne inférieure (en orange).
Ainsi à l'aide de l'ensemble des résultats obtenus, nous pouvons déduire que quelque soit la taille des graphes implémentées, la probabilité de succès est toujours supérieure à la borne étudiée en cours, nos résultats sont donc cohérents.

Question 2.B

Nous avons vu que pour augmenter la probabilité de succès, il est utile de répéter plusieurs fois l'algorithme de Karger. Nous obtenons ainsi l'algorithme de Karger itéré qui prend en entrée un entier T et exécute l'algorithme de l'exercice 1 (Karger) $T \geq 1$ fois.
L'objectif de cette question est d'adapter donc les implémentations de l'exercice 1 pour obtenir l'algorithme de Karger itéré que voici :

Algorithme 2 : Algorithme de Karger itéré

Entrée : $G = (V, E)$ un graphe non orienté avec $n = \#V$ et $m = \#E$, T
Sortie : S une coupe de G
$m^* \leftarrow +\infty$
pour i *de* 1 *à* T **faire**
  tant que $\#V > 2$ **faire**
    $e \xleftarrow{\text{⚄⚅}} E$
    $G \leftarrow G/e$
  $\{v_1, v_2\} \leftarrow V$ ▷ Il reste deux sommets dans V
  $S = \{$ sommets qui « apparaissent » dans v_1 $\}$
  $m \leftarrow$ cardinal de la coupe $(S, V \setminus S)$
  si $m < m^*$ **alors**
    $S^* \leftarrow S$
    $m^* \leftarrow m$
retourner S^*

FIGURE 2.2 – Algorithme de Karger itéré

Notre implémentation est définie comme ceci :

```
def karger_itere(Matrix,sommet,t):
    m_min = 1000
    for i in range(t):
        print("\n - Execution n°",i+1,"de l'algorithme de
        mat,s=karger(Matrix,sommet)
        m=mat[0][1] #Car quoiqu'il arrive la taille de la
        if (m<m_min):
            s_opt=s
            m_min=m
        mat_opt=mat
        print("Le cardinal de la coupe est m=",m)
        print("-----------------------------------------
    print("Le cardinal de la coupe minimum est : ",m_min)
    return mat_opt,s_opt
```

FIGURE 2.3 – Implémentation de l'algorithme de Karger itéré (représentation de leur graphe par matrice d'adjacence)

Question 2.C

Étude expérimentale de Karger itéré sur les familles de graphes dont on connaît la taille d'une coupe minimum (en fonction de T et du nombre de sommets du graphe).
Pour répondre à cette question nous allons procéder en deux étapes :

Analyse de la complexité expérimentale de Karger itéré

Pour ce faire nous allons tout simplement suivre les mêmes étapes que précédemment, pour obtenir ainsi les résultats suivants :

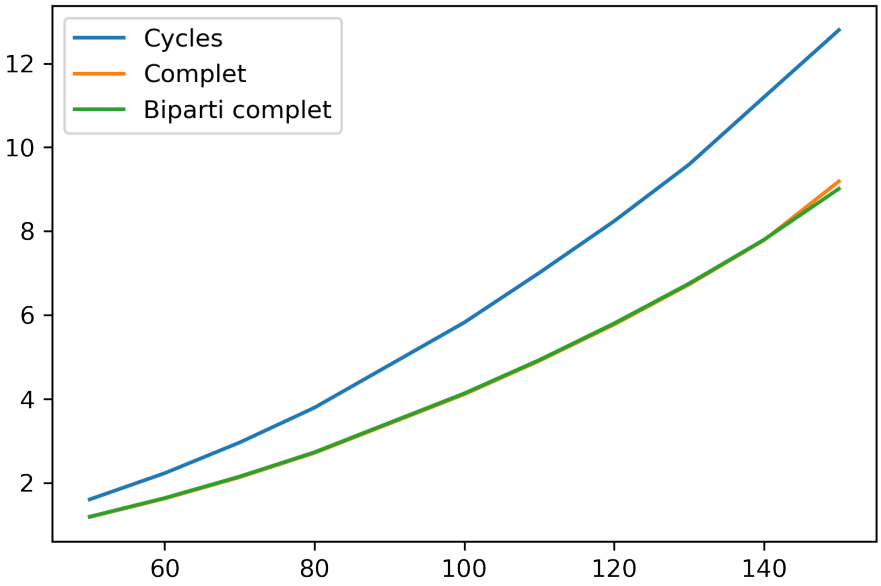

FIGURE 2.4 – Complexité expérimentale de Karger itéré

Ainsi, nous pouvons aisément voir d'après la figure ci-dessus que la complexité de notre algorithme se rapproche fortement de la complexité théorique de Karger itéré $O(n^4 log(n))$ représentée comme suit :

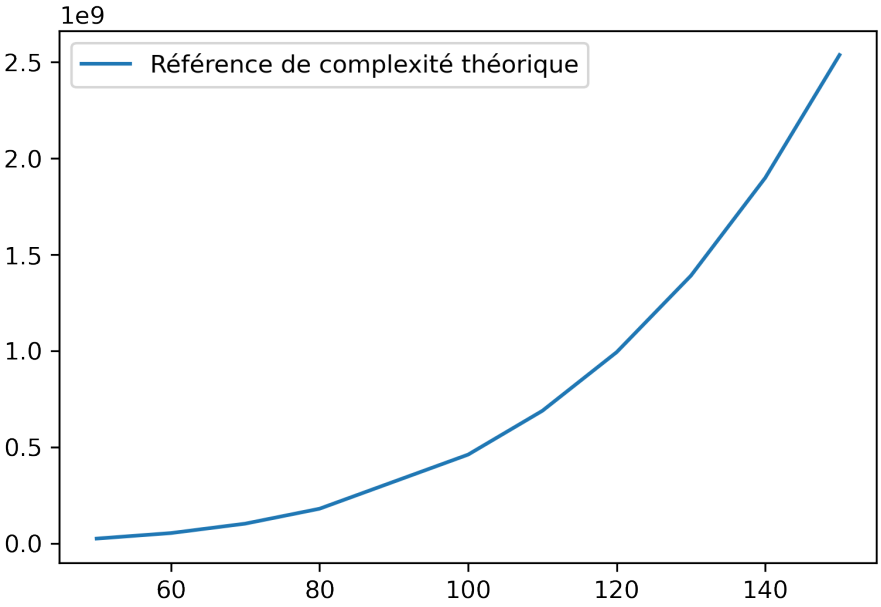

FIGURE 2.5 – Complexité théorique de Karger itéré

Analyse expérimentale de la probabilité de succès de Karger itéré

Dans cette partie nous nous intéresserons uniquement aux graphes biparti complets dont nous connaissons la taille de leur coupe minimum. Pour obtenir les résultats suivants :

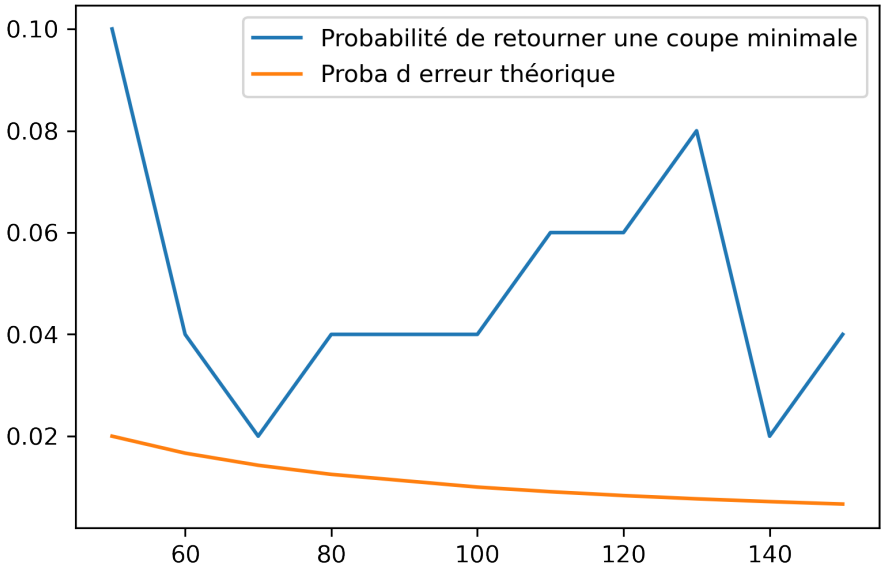

FIGURE 2.6 – Analyse expérimentale de la probabilité de succès de Karger itéré

Comme nous le montre la figure ci-dessus, on remarque clairement que la probabilité de succès de Karger itéré 50 fois est supérieure à la probabilité d'échec à savoir : la probabilité de ne pas retourner une coupe minimum qui théoriquement est $\leq \frac{1}{n}$, ce qui confirme l'analyse théorique effectuée en cours.
En effet si l'on répète T fois l'algorithme de Karger itéré, cet algorithme échoue si les T exécutions échouent ce qui arrive avec une probabilité $\leq (1 - \frac{2}{n(n-1)})^T$ qui elle même est $\leq \frac{1}{n}$.

Ainsi, nous pouvons conclure à l'aide des expérimentations menées que la complexité totale de l'algorithme de Karger est en $O(n^4 log(n))$ avec une probabilité d'erreur $\leq \frac{1}{n}$

Exercice 3 - Algorithme de Karger-Stein

L'algorithme de contraction aléatoire de Karger-Stein améliore considérablement la durée d'exécution de l'algorithme de Karger en diminuant le nombre d'itérations nécessaires pour produire une coupe minimum avec une forte probabilité d'exactitude.
Le concept de base est que la probabilité de réduire la mauvaise arête (réduire une arête de la coupe minimum) augmente à mesure que le nombre d'arêtes diminue.
Ainsi, réduire le nombre d'arêtes à $1 + n/\sqrt{2}$ (avec n le nombre de sommets du graphe) et exécuter l'algorithme sur des graphes plus petits, réduit considérablement le nombre d'itérations nécessaires.

Question 3.A

Montrer que le temps d'exécution $T(n)$ de l'algorithme de Karger-Stein sur un graphe à n sommets vérifie la relation :

$$T(n) = 2T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil) + O(n^2)$$

Algorithme 4 : Algorithme de Karger-Stein KargerStein(G)

Entrée : $G = (V, E)$ un graphe non orienté avec $n = \#V$ et $m = \#E$, T
Sortie : S une coupe de G
$m^* \leftarrow +\infty$
si $\#V \leq 6$ **alors**
  retourner une coupe minimale de G ▷ par recherche exhaustive
sinon
  $t \leftarrow \lceil 1 + \#V/\sqrt{2} \rceil$ O(1)
  $G_1 \leftarrow$ ContractionPartielle(G, t) O(n²)
  $S_1 \leftarrow$ KargerStein(G_1) ; $m_1 \leftarrow$ cardinal de la coupe $(S_1, V \setminus S_1)$ T([1+n/√2])
  $G_2 \leftarrow$ ContractionPartielle(G, t) O(n²)
  $S_2 \leftarrow$ KargerStein(G_2) ; $m_2 \leftarrow$ cardinal de la coupe $(S_2, V \setminus S_2)$ T([1+n/√2])
  si $m_1 < m_2$ **alors**
    retourner S_1 O(1)
  sinon
    retourner S_2 O(1)

Figure 3.1 – Temps d'exécution de Karger-Stein

Nous effectuons la contraction partielle de G, ce qui prend $O(n^2)$
De plus, on applique à nouveau l'algorithme de Karger-Stein, cette fois sur deux graphes de taille $\lceil 1 + \frac{n}{\sqrt{2}} \rceil$ chacun, ce qui nous donne : $2T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil)$

Donc,

$$T(n) = 2T(\lceil 1 + \frac{n}{\sqrt{2}} \rceil) + O(n^2)$$

En déduire que $T(n) = O(n^2 \, log \, n)$:
On peut dire que : $T(n) = a\,T(\frac{n}{b}) + f(n)$, avec $a = 2$, $b = \sqrt{2}$ et $f(n) = O(n^2)$.
Plus précisément : $f(n) = O(n^c \, log^k \, n)$, avec $c = 2$ et k une constante $= 0$
Selon le *Master Theorem* : Si $f(n) = O(n^c \, log^k \, n)$, alors : $T(n) = O(n^c \, log^{k+1} \, n)$

Donc :

$$T(n) = O(n^2 \, log \, n)$$

Question 3.B

Montrer que la probabilité de succès $P(n)$ de l'algorithme de Karger-Stein sur un graphe à n sommets vérifie la relation :

$$P(n) \leq 1 - \left(1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Soit $\bar{P}$ la probabilité que l'algorithme de Karger-Stein échoue.
Donc, $P(n) = 1 - \bar{P}$ Soit P_1 la probabilité que l'un des graphes (G_1 , G_2) soit une coupe minimum
P_2 : la probabilité que la contraction partielle du graphe G n'ait pas touché la coupe minimum (i.e. la coupe minimum est dans G_1 ou dans G_2)
P_3 : la probabilité de succès de Karger-Stein sur l'un des graphes (G_1 , G_2)

Donc, $P_3 = P\left(\frac{n}{\sqrt{2}} + 1\right)$
$P_1 = P_2.P_3$
$P_1 = P_2.P\left(\frac{n}{\sqrt{2}} + 1\right)$

Sachant que : $P_2 \geq \frac{1}{2}$, alors :

$$P_1 \geq \frac{1}{2} P\left(\frac{n}{\sqrt{2}} + 1\right) \tag{3.1}$$

Notons $\bar{P_1}$ la probabilité d'échec de Karger-Stein sur un des graphes (G_1 , G_1)
$\bar{P_1} = 1 - P_1$

D'après (3.1) : $\bar{P_1} < 1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}} + 1\right)$

Notons P_3 la probabilité d'échec de Karger-Stein sur les deux graphes G_1 et G_2.

$$P_3 < \left(1 - \tfrac{1}{2}P\left(\tfrac{n}{\sqrt{2}} + 1\right)\right)^2$$

Sachant que P_3 est la probabilité de l'évènement contraire "Réussite de Karger-Stein", on en déduit que :

$$P(n) \leq 1 - \left(1 - \frac{1}{2}P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Question 3.D

L'objectif de cette question est d'implémenter l'algorithme de Karger-Stein. Pour ce faire, il suffit tout simplement d'adapter les fonctions définies dans les questions précédentes pour obtenir la fonction suivante :

```
from math import *
def contraction_partielle(Matrix,sommets,t):
    mat=copy.deepcopy(Matrix)
    som=copy.deepcopy(sommets)
    while(len(mat)>t):
        u,v = tirage_arête(mat)
        mat,s = contraction(mat,u,v,som)
    return mat,s

def karger_stein(Matrix,sommets,t):
    if(len(sommets)<=6):
        mat,s=karger_itere(Matrix,sommets,t)
        return mat,s
    else:
        t=ceil(1+len(sommets)/sqrt(2))
        matG1,sG1=contraction_partielle(Matrix,sommets,t)
        matS1,sS1=karger_stein(matG1,sG1,t)
        m1=matS1[0][1]
        matG2,sG2=contraction_partielle(Matrix,sommets,t)
        matS2,sS2=karger_stein(matG2,sG2,t)
        m2=matS2[0][1]
        if(m1<m2):
            return matS1,sS1
        else:
            return matS2,sS2
```

FIGURE 3.2 – Implémentation de l'algorithme de Karger-Stein

Question 3.E

Étude expérimentale des implémentations sur les familles de graphes déjà étudiées :
Pour répondre à cette question nous avons tout simplement exécuté notre algorithme de Karger-Stein sur plusieurs instances de tailles croissantes afin d'obtenir l'évaluation du temps d'exécution par rapport à la taille des données.

Toutefois à cause de l'erreur indiquée par le langage python suivante : *maximum recursion depth exceeded while calling a Python object* (Python ne supporte pas très bien la récursion à cause de son manque de TRE "Trail Recursion Elimination"), nous étions contraints de travailler avec les tailles suivantes : $[4, 6, 8, 10]$ pour obtenir les résultats suivants :

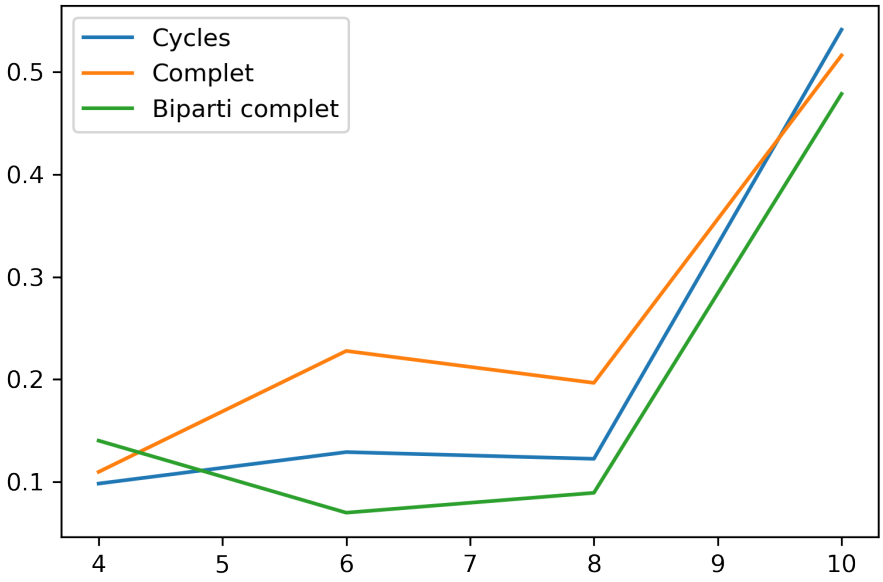

FIGURE 3.3 – Complexité expérimentale de Karger-Stein

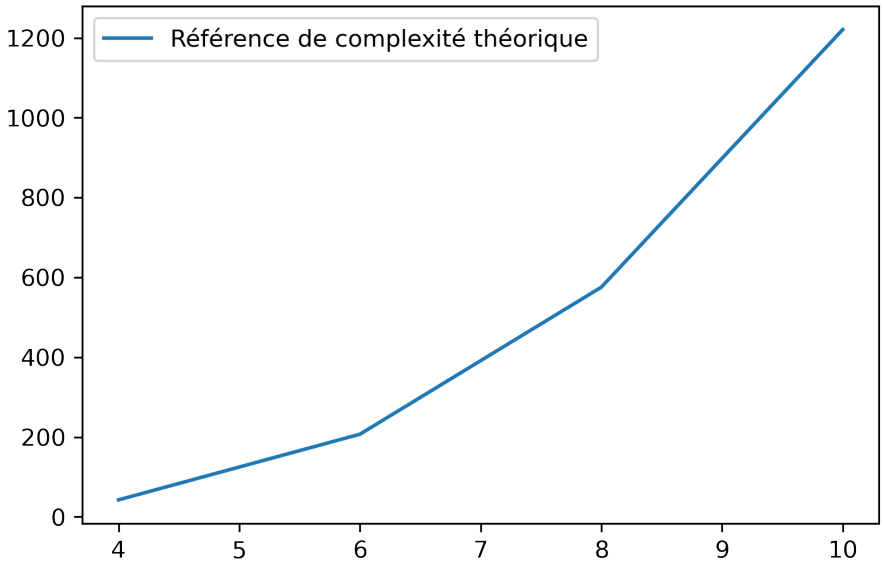

FIGURE 3.4 – Complexité théorique de Karger-Stein

Ainsi nous pouvons observer une certaine similitude entre la complexité expérimentale et la complexité théorique lorsque la taille croît (supérieure à 8).

Cela nous laisse paraître que l'algorithme implémenté de Karger-Stein correspond aux résultats théoriques menés en cours (le recours à des tailles plus grandes serait préférable

et beaucoup plus significatifs pour notre étude expérimentale).
A noter que la probabilité d'erreur dans le cas de Karger-Stein est également $\leq \frac{1}{n}$.

Nous pouvons donc conclure que l'algorithme de Karger-Stein dispose d'une complexité en $O(n^3 log^3(n))$ tel que la probabilité d'erreur est $\leq \frac{1}{n}$.

Conclusion générale

À travers ce projet, nous avons pu étudier le problème de recherche d'une coupe minimum dans un graphe non orienté grâce à l'algorithme de Karger et de ses améliorations.

Nous avons commencé par implémenter l'algorithme de Karger ainsi que ses variantes à l'aide de plusieurs représentations de graphes (listes d'adjacence et matrice d'adjacence) puis, nous avons procédé à l'étude de la complexité expérimentale de ces derniers.
En effet représentant le sujet principal de ce projet, plusieurs analyses de complexité ont été réalisées afin de comparer la complexité de nos implémentations avec les complexités théoriques étudiées en cours et ainsi, juger de la bonne qualité ou pas de nos implémentations.

Enfin, nous avons également analysé la probabilité de succès de l'algorithme de Karger itéré dans le cadre de l'amplification du succès de l'algorithme initial, puis, nous avons implémenté la version récursive de l'algorithme de Karger à savoir, l'algorithme de Karger-Stein.