Projet "Résolution de problèmes d'optimisation"
Recherche de contre-exemple

Master Informatique - 1ère année

année 2022-2023

1 Contexte

Les graphes sont l'un des universaux pour modéliser de nombreux phénomènes tel que la propagation d'information dans un réseau de personnes, la propagation de maladie, l'interaction entre espèces, la relation d'interdépendance entre entreprises, etc. Les propriétés fondamentales de ces graphes, souvent appelés réseaux lorsqu'une information est échangée entre les nœuds, sont nombreuses et donnent des propriétés diverses aux systèmes modélisés. Par exemple, le nombre de couleurs minimales pour colorier un graphe tel que deux nœuds voisins n'est pas la même couleur a des applications en affectations de ressources ; Le rayon spectral d'un graphe (*i.e.* la plus grande valeur propre de la matrice d'adjacente d'un graphe) est relié à la vitesse de diffusion d'information dans un réseau ; Ou encore la taille du plus grand couplage (maximum matching number) a des conséquences pour les problèmes de répartition de ressources.

De nombreuses propositions reliants des propriétés fondamentales du graphe ont été proposés ou démontrées. Nous nous proposons dans ce projet de montrer qu'une proposition conjecture est fausse en exhibant un contre-exemple, c'est-à-dire en trouvant un graphe qui ne vérifie pas la proposition.

Soit G_n un graphe simple non orienté à n noeuds, et soit f la fonction définie par :

$$f(G_n) := \lambda_{G_n} + \mu_{G_n} - \sqrt{n-1} - 1 \ ,$$

où λ_{G_n} est le rayon spectral de G_n et μ_{G_n} est la taille du couplage maximum de G_n (maximal matching number). Soit la proposition conjecture [1] suivante :

$$\boxed{f(G_n) \geqslant 0 \quad \text{pour tout graphe simple } G_n \text{ connexe.}}$$

Cette conjecture est fausse pour certaines tailles de graphe [2]. Pour le démontrer, nous allons minimiser la fonction f sur l'ensemble des graphes. Si pour un graphe G_n connexe, $f(G_n) < 0$ alors la conjecture est fausse.

2 Techniques d'optimisation combinatoire

Dans cette section, nous vous proposons de mettre au point des algorithmes d'optimisation combinatoire pour résoudre le problème d'optimisation.

L'ensemble des arêtes d'un graphe non orienté avec n nœuds peuvent être représentés de différentes manières (voir figure 1): une liste de couple de nœuds, une matrice d'adjacence, etc. Comme la matrice d'adjacence est symétrique, nous représentons la partie supérieure de la matrice par un vecteur de booléens de dimension $\frac{n(n-1)}{2}$ pour coder l'ensemble des arêtes d'une graphe. L'arête entre les nœuds i et j a pour indice $\frac{(2n-1-i)i}{2} + j - i - 1$ dans le vecteur. Le code python `solution.py` et `graphProblem.py` contiennent les classes qui permettent de définir une solution potentielle (un graphe) et la fonction d'évaluation f. Le code `main_basic.py` donne des exemples d'utilisation de ces classes.

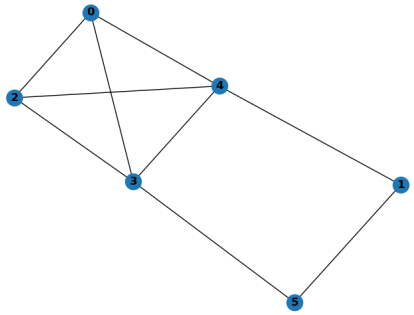
Pour tester expérimentalement vos algorithmes, vous pouvez considérez deux tailles de problème : l'une petite avec $n = 6$ et l'autre plus grande pour $n = 19$ où il est connu qu'il existe des contre-exemples. À noter que nous ne savons pas s'il existe des contre-exemples pour $n = 18$.

Questions :

2.1 Définition du problème :

2.1.a Ecrire formellement le problème d'optimisation dans ce cas.

2.2.b Quelles sont les caractéristiques de ce problème d'optimisation ?



$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 1: Exemple de graphe à 6 nœuds, et la matrice d'adjacence correspondante.

2.2 Iniitialisation :

2.2.a Proposer une méthode d'initialisation aléatoire d'un graphe.

2.2.b Si votre méthode comporte des paramètres, étudier l'influence des paramètres sur la qualité de la solution initialisée.

2.3 Voisinage et optima locaux :

2.3.a Définir un voisinage élémentaire possible entre deux graphes.

2.3.b Coder une recherche locale de type Hill-Climber. Mesurer et analyser la qualité des optima locaux obtenus, le nombre d'évaluation, le temps et le nombre de pas améliorants pour les obtenir à partir d'une solution aléatoire.

2.4 Métaheuristiques :

2.4.a Coder une métaheuristique de type Iterated Local Search (ILS) en proposant un opérateur de perturbation possible. Mesurer et analyser les performances de l'algorithme en fonction de ses paramètres.

2.4.b Proposer des voisinages plus larges que le voisinage élémentaire précédant. Coder une métaheuristique de type Variable Neighborhood Search (VNS). Comparer et analyser les performances des algorithmes ILS et VNS.

3 Techniques d'optimisation numérique et apprentissage

Dans cette section, nous vous proposons de mettre au point un algorithme d'optimisation numérique pour résoudre le problème de génération de contre exemple.

Le principe consiste à coder la génération des arêtes du graphe par une loi aléatoire. Un vecteur $p \in \mathbb{R}^{(n-1)n/2}$ définit la probabilité d'existence de chaque arête : le graphe contient l'arête i selon la loi de Bernouilli de paramètre p_i, *i.e.* l'arête i a la probabilité p_i d'exister dans le graphe. De nombreux algorithmes peuvent définir le vecteur p. Nous vous proposons de définir ce vecteur à l'aide d'un réseau de neurones N_w parametrisé par un vecteur $w \in \mathbb{R}^d$ prenant en entrée deux vecteurs, chacun de dimension $n(n-1)/2$ correspondant au nombre d'arêtes et ayant comme sortie un nombre réel $z \in [0, 1]$ pour définir chaque probabilité p_i. Le premier vecteur d'entrée correspond aux arêtes du graphe en cours de construction jusque là, et le second e^i donne l'information de l'arête i considérée pour l'ajout ($e^i_k = 1$ ssi $k = i$, $e^i_k = 0$ sinon). La procédure suivante **generate_graph** décrit le processus de génération à partir du réseau de neurones. Itérativement, une arête est ajoutée l'une après l'autre en fonction des arêtes déjà présentes. Vous trouverez aussi le code pour définir une solution dans `solutionNN.py` et les fonctions d'évaluation possibles dans `nn_graphGenerator.py` qui peuvent vous aider, ainsi que dans l'article original [2].

```
def generate_graph(N_w, n):
        nb_aretes = int(n*(n-1)/2)
        g = [0 for i in range(nb_aretes)]
        for i in range(nb_aretes):
                e = [0 for i in range(nb_aretes)]
                e[i] = 1
                p = N_w(g, e)
```

```
            if rand() < p:
                g[i] = 1
    return g
```

Le principe de l'algorithme que vous devez mettre au point consiste à apprendre le réseau de neurone qui génère des graphes de plus petite valeur de f.

Questions :

3.1 Définition du problème :

3.1.a Ecrire formellement le problème d'optimisation dans ce cas.

3.2.b Quelles sont les caractéristiques de ce problème d'optimisation ?

3.2 Voisinage :

3.2.a Comment naviguer dans l'ensemble des solutions ?

3.3 Algorithme :

3.3.a Proposer et coder un algorithme d'optimisation adapté à cette définition du problème.

3.3.b Tracer l'évolution de la valeur de votre solution en fonction du nombre de solutions évaluées et du temps de calcul.

3.3.c Analyser et comparer les performances de votre algorithme avec les algorithmes de la section précédante.

3.3.d Est-il possible de définir un algorithme d'optimisation qui combine les algorithmes des sections 2 et 3 ?

References

[1] Gilles Caporossi and Pierre Hansen. Variable neighborhood search for extremal graphs: 1 the autographix system. *Discrete Mathematics*, 212(1-2):29–44, 2000.

[2] Adam Zsolt Wagner. Constructions in combinatorics via neural networks. *arXiv preprint arXiv:2104.14516*, 2021.