TP : Introduction à GLPK et optimisation dans les graphes

N. Dupin

Université d'Angers, année universitaire 2022-2023

Consignes générales

Ce TP est à rendre individuellement ou en binôme, et donnera la note de contrôle continue (CC). Tous les créneaux de TD/TP du cours sont consacrées à ce projet, une part de travail autonome est également attendue. Le sujet de projet est rédigé pour que vous soyez assez autonomes pour avancer, avec plusieurs parties assez indépendantes pour vous puissiez avancer sur plusieurs fronts/questions si vous bloquez sur un point. Les questions/réponses se font uniquement sur les séances de TP. Pas de question individuelle par mail, après le dernier créneau de TP, il est trop tard pour poser des questions. Il vous est dont conseillé de profiter des séances de TP pour des questions sur les points les plus critiques qui peuvent vous bloquer dont l'implémentation, réaliser par vous même le travail où vous serez le plus autonomes, comme pour lancer un code qui fonctionne, avancer le travail d'analyse et de présentation des résultats.

Le rendu de ce TP projet est attendu sur Moodle pour le 31 mars, avec un rapport glpk-NOM1(-NOM2).pdf, où le code GLPK complété est inséré aux réponses aux questions. Pour la partie 5, "De la PLNE exacte aux matheuristiques" le code est à fournir dans une archive séparée, les réponses aux questions se trouvant alors dans le rapport.

1 Installations

On utilisera lors de ce TP, le solveur libre (sous licence LGPL) d'optimisation linéaire GLPK qui peut facilement être installé sur vos machines personnelles. L'installation de CBC est mentionnée et peut aussi être réalisée, mais n'est pas indispensable aux séances de TP. L'installation de CBC est intéressante pour comparer les performances de CBC et GLPK (on observera que le solveur de CBC est bien plus performant). Dans les TP, on utilisera les facilités de modélisation avec GLPK, qui pourront aussi permettre de lancer une résolution plus efficace avec CBC, ou avec Cplex, disponible à l'université.

Si vous avez le choix, il vaut mieux installer GLPK sous Linux et Mac car l'installation et l'utilisation de GLPK sont plus simples que sous Windows.

1.1 Installation de GLPK sous Linux et Mac

Installez GLPK sous Linux en tapant `sudo apt-get install glpk-utils` dans le terminal. GLPK est aussi disponible dans le gestionnaire de paquets. Sous Mac, reportez-vous à `http://arnab-deka.com/posts/2010/02/installing-glpk-on-a-mac`

Pour vérifier que GLPK est installé sur votre poste, tapez la commande `glpsol -h`. Elle doit renvoyer les différentes options de GLPK.

1.2 Installation de GLPK sous Windows

Sous Windows, commencez par télécharger la version Windows de GLPK: `https://sourceforge.net/projects/winglpk/`
GLPK s'utilise en ligne de commande (outil cmd sous Windows), similairement à l'utilisation sous Linux/Mac. Cela nécessite de modifier la variable d'environnement `path`, pour que glpsol soit reconnu depuis la ligne de commande. Vous trouverez les instructions très détaillées ci-dessous:
`http://www.osemosys.org/uploads/1/8/5/0/18504136/glpk_installation_guide_for_windows10_-_201702.pdf`
Ce tutoriel existe sous version vidéo (à partir de 4 minutes pour GLPK): `https://www.youtube.com/watch?v=wcGr0hxmEX0`

Pour vérifier que GLPK est installé sur votre poste, tapez la commande `glpsol -h`. Elle doit renvoyer les différentes options de GLPK.

1.3 Installation de CBC

CBC, de la suite COIN-OR , est au autre solveur libre de PL/PLNE. L'installation de CBC n'est pas nécessaire pour le déroulement des TP, son installation est intéressante pour comparer les performances de CBC et GLPK.
Pour installer CBC sous Linux Debian/Ubuntu, tapez: `sudo apt-get install coinor-cbc` dans le terminal. Sous Fedora, CBC est disponible dans le package `coin-or-Cbc`. Pour une installation sous Windows ou Mac, vous trouverez les instructions ici: `https://github.com/coin-or/Cbc`.

2 Le modeleur GNU MathProg de GLPK

Dans la suite, on utilisera le modeleur GNU MathProg fourni avec GLPK. Les alternatives sont OPL (fourni avec Cplex), Zimpl (de la suite SCIP), CMPL, de la suite COIN-OR ... Si les principes des modeleurs sont identiques, il n'y a pas de syntaxe unifiée pour les modeleurs, contrairement aux fichiers .lp et .mps. Ce cours permettra de se familiariser avec ces principes en utilisant GLPK. Si vous avez un autre solveur à votre disposition dans un autre contexte, vous devrez pouvoir vous adapter facilement à de nouvelles syntaxes.

Sur ce TP, le code de base est fourni pour le sac à dos, vous pourrez tout coder en réutilisant les fonctionnalités présentés dans les codes fournis. Pour plus de détails sur le fonctionnement et les différentes syntaxes accessibles de GNU MathProg, vous pouvez vous à:
`http://lim.univ-reunion.fr/staff/fred/Enseignement/Optim/doc/GLPK/CoursGLPK.pdf`

2.1 Code fourni

Le code fourni à ce TP comprend un dossier `knapsack`, où le problème basique du sac à dos est implémenté Plus précisément, on trouvera les fichiers suivants que l'on regardera en premier:

- un fichier `knapsack.mod`, modèle GLPK, qui correspond au PLNE du sac à dos binaire.
- un fichier `knapsackRCO.mod`, modèle GLPK, qui correspond à un PL: la relaxation continue de `knapsack.mod`, le sac à dos continu.
- un fichier `knapsack.dat`, fichier de données GLPK, expliqué dessous.
- un sous-dossier `Data`, avec plusieurs jeux de données au même format que `knapsack.dat`.

Le fichiers `knapsack.dat` est assez intuitif: on a 5 objets, respectivement de poids 2, 6, 1, 7 et 8 et de coûts 12, 15, 5, 16 et 5 17. Le sac à dos supporte un poids de 20:

```
data;
param nbItems := 5;
param capacity := 20;
param costItem :=
1 12
2 15
3 5
4 16
5 17;
param weightItem :=
1 2
2 6
3 1
4 7
5 8;
end;
```

`knapsack.dat` correspond à toutes les caractéristiques d'une instance d'un problème de sac à dos, c'est-à-dire les **paramètres** d'un problème de sac à dos.

Les fichiers `knapsack.mod` et `knapsackRCO.mod` sont très similaires. La première partie déclare les paramètres `nbItems` et `capacity`, ainsi que deux vecteurs de taille `nbItems`: `costItem` et `weightItem`.

```
param nbItems ; #number of items
param capacity; #knapsack capacity

set Items :=1..nbItems;
param costItem{i in Items}; #cost of items
param weightItem{i in Items}; #weight of items
```

Ces lignes ne spécifient pas de valeur pour les paramètres, mais définissent en fonction de ces données, la taille des éléments attendus quand on a des tableaux ou des matrices. Ce seront les valeurs trouvées dans le .dat qui permettront de construire le PL/PLNE complet. On remarquera que les noms des variables doivent coïncider dans le .mod et le .dat, c'est ce qui permet de faire la correspondance.

Une fois les paramètres définis, on définit la variable d'un objet, pour indiquer s'il est sélectionné. Ces variables sont indexées par `Items`, la syntaxe est la suivante.
Pour `knapsack.mod`, on déclare des variables binaires:

```
var x{Items} binary;
```

Pour `knapsackRCO.mod`, on a des variables à valeurs continues dans $[0, 1]$:

```
var x{Items}, >=0, <=1;
```

On remarquera que ces différences correspondent dans les .lp à `Bounds` ou `Binaries`.

La suite du modèle correspond à l'écriture de l'objectif et des contraintes linéaires (= la section objectifs et contraintes d'un .lp). L'objectif et les contraintes sont communs au modèle PLNE et sa relaxation linéaire. Ici, il n'y a qu'une seule contrainte, appelée `capacityConstraint`:

```
maximize obj :sum {i in Items} costItem[i]*x[i] ;
subject to
capacityConstraint : sum{i in Items} weightItem[i]*x[i] <= capacity ;
```

La fin du .mod appelle la résolution par `solve`, pour signifier que l'écriture des contraintes est terminée. Après le `solve`, on peut avoir accès aux valeurs des variables dans la solution optimale (ou la meilleure solution trouvée). Ces lignes permettent de formater des résultats de sortie plus simplement que la sortie standard qu'on avait déjà observé avec `-o` ou `--output`. Ici, on affiche juste les objets sélectionnés:

```
solve;
printf "The continuous knapsack contains the items:\n";
printf {i in Items : x[i] > 0} " item %i in quantity %f \n", i, x[i];
printf "\n";
end;
```

Le dossier `knapsack` comprend également deux fichiers modèles GLPK: `knapsackFull-variation.mod` et `knapsackFull.mod`. Dans ces deux cas, on inclut les données dans le modèle, pour n'avoir qu'un seul fichier. Ce n'est pas l'utilisation la plus habituelle d'un modeleur, tout l'intérêt d'un modeleur est de séparer la modélisation mathématique en PL/PLNE dans le .mod, et les données dans le .dat, pour pouvoir appeler les différents calculs en ne changeant que le .dat. `knapsackFull.mod` correspond au modèle PLNE `knapsack.mod` instancié avec `knapsack.dat`. `knapsackFull-variation.mod` correspond au fichier donné dans la documentation de GLPK, avec une syntaxe légèrement différente (et plus compliquée). Dans la suite des TP, une syntaxe simple comme dans `knapsack.mod` sera largement suffisante.

On trouvera également dans codeFourni/:

- le fichier `sudoku.mod`, fourni avec GLPK, permet une autre illustration. Les solveurs de PLNE ne sont pas forcément les plus adaptés et efficaces à de tels problèmes, qui relèvent plus de la Programmation Par Contrainte (PPC). Pour du sudoku 9×9, cela ne pose pas de souci , même pour GLPK. `sudoku.mod` comprend le modèle GLPK et une instance. Pour résoudre: `glpsol -m sudoku.mod`
- le dossier `techRouting`, correspond à un problème de tournée de techniciens, dont vous trouvez la description et le modèle PLNE à l'adresse `https://hal.archives-ouvertes.fr/hal-02304958/document`. Un modèle .mod et un .dat sont fournis, pour vous illustrer la syntaxe, sur un PLNE avec plusieurs jeux de contraintes et de variables, indexées sur plusieurs indices. Ce dossier permet aussi d'illustrer que le modeleurs GLPK permet d'écrire assez facilement des problèmes, au plus près des équations mathématiques, ce qui est un atout pour modéliser des problèmes industriels d'optimisation.

2.2 Lancer des calculs avec le modeleur

Pour lancer un modèle GLPK, on utilise le préfixe -m ou –model, ce qui donne des commandes du type:

```
glpsol -m knapsackFull.mod
```

```
glpsol --model knapsackFull-variation.mod
```

Pour lancer un modèle GLPK sur une instance de données .dat, on utilise de plus le préfixe -d ou –data, ce qui donne des commandes du type:

```
glpsol --model knapsack.mod --data knapsack.dat
```

```
glpsol -m knapsack.mod -d Data/kp-100-1.dat
```

Les fichiers de données `kp-NB-*.dat` comprennent des sac à dos de taille NB. A ce stade, vous pouvez observer que les modèles se résolvent facilement par GLPK pour $NB \leqslant 10000$. Pour $NB = 100000$, la résolution commence à être plus difficile pour GLPK. Dans ce cas, on peut limiter le temps de calcul à 60s, et afficher la meilleure solution trouvée par:

```
glpsol -m knapsack.mod -d Data/kp-100000-1.dat --tmlim 60
```

On peut également limiter l'espace mémoire utilisé, avec l'option `--memlim` en donnant une limite mémoire exprimée en Mo.

En pratique, GLPK est bien plus lent que Cplex, Gurobi et CBC pour des calculs de grande taille. Dans ce cas, on peut avoir intérêt à utiliser le modeleur uniquement pour générer un fichier .lp que d'autres solveurs de PL/PLNE pourront résoudre. Pour générer un fichier .lp sans résoudre, on utilise l'option `--check`, et `--wlp`:

```
glpsol --check -m knapsack.mod -d Data/kp-100000-1.dat --wlp kp-100000-1.lp
```

Appliqué sur des petits jeux de données, la génération de fichier .lp peut permettre de déboguer des équations des modèles .mod.

Ensuite, il est à noter que l'option `--nomip` permet de résoudre uniquement la relaxation continue d'un problème, ce qui induit le même calcul que `knapsackRCO.mod`:

```
glpsol --nomip -m knapsack.mod -d knapsack.dat
```

N.B: dans ce cas, il faut veiller à ce que les scripts de post-traitement aient toujours un sens en relaxation continue. Pour `knapsackRCO.mod`, on affiche l'état des variables fractionnaires, ce cas n'est pas inclus dans le modèle binaire.

Pour affiche plus d'options de GLPK, utilisez la commande `glpsol --h`. On notera que certaines options font apparaître des paramètres de résolution PLNE. On trouvera par exemple des stratégies de branchements, des stratégies sur l'ordre de parcours des noeuds de l'arbre de Branch & Bound, qui sont des questions que vous vous êtes sans doute posées pour votre implémentation maison sur un TP précédent:

```
--first           branch on first integer variable
--last            branch on last integer variable
--mostf           branch on most fractional variable
--drtom           branch using heuristic by Driebeck and Tomlin (default)
```

```
  --pcost           branch using hybrid pseudocost heuristic (may be
                    useful for hard instances)
  --dfs             backtrack using depth first search
  --bfs             backtrack using breadth first search
  --bestp           backtrack using the best projection heuristic
  --bestb           backtrack using node with best local bound (default)
```

Cela montre que GLPK a une implémentation assez générique du Branch & Bound, où différentes stratégies peuvent être considérées. L'implémentation C++ du TP précédent explique les options `--bestb` et `--bfs` ou `--dfs`.

De multiples paramètres rentrent en ligne de compte pour expliquer les différences entre GLPK et CBC ou Cplex. Dans la suite, on va jouer sur des paramètres de GLPK que vous connaissez moins, les coupes et les heuristiques, avec les options `--cuts`, `--cover`, `--clique` et `--fpump`:

- `--fpump` active une heuristique pour trouver des solutions initiales au problème: Feasibility Pump. Avoir au plus tôt de meilleures solutions peut permettre d'élaguer plus tôt des noeuds de l'arbre de Branch&Bound
- `--cuts` active les coupes. Les coupes permettent d'améliorer les qualités de relaxation continue par l'ajout de contrainte vérifiée par toutes les solutions entières, et qui rendent infaisables certaines solutions continues. De tels mécanismes sont illustrés sur la dernière partie de ce TP. L'effet de coupes est de réduire le nombre de noeuds explorés par B&B pour converger, en améliorant les qualités des bornes supérieures, on peut activer plus tôt les critères d'arrêt d'exploration. En contre-partie, la génération de coupes nécessite un temps supplémentaire, et rallonge les temps de calculs de bornes de relaxation continue dans les itérations de simplexe, en augmentant la taille des matrices dans l'algorithme du simplexe.
- `--cover` active uniquement les coupes dites de couvertures, spécifiques à des structures de sac à dos.
- `--clique` active uniquement les coupes dites de cliques, pour des relations d'incompatibilités induites entre variables.

Les définitions des coupes et leur illustration est fournie en annexe. Pour analyser les différences impliquées par ces différents paramètres, il peut être nécessaire e savoir lire ces fichiers de logs des solveurs, l'annexe en donne également un aperçu.

3 Optimisation dans les graphes

On considère des graphes non orientés $G = (V, E)$ avec les notations standards:

- V désigne les sommets,
- $E \subset V^2$ désigne les arêtes, couples de sommets reliés dans le graphe
- Pour tout sommet $v \in V$, $\delta(v)$ désigne l'ensemble des voisins de v.
- Pour tout sommet $v \in V$, $N_v = |\delta(v)|$ désigne le degré du sommet v, le nombre de ses voisins.

Code fourni

Le code fourni comprend tout d'abord un fichier `graph0.dat` qui fixe le format d'un graphe. `graph[i][j]` indiquera que i et j sont reliés si et seulement si `graph[i][j]`=1. Comme les instances sont construites avec des graphes non orientés, on a toujours `graph[i][j]`=`graph[j][i]`.

```
data;
param nbVertex:=5;
param graph:  1 2 3 4 5 :=
 1 0 1 0 1 0
 2 1 0 1 1 1
 3 0 1 0 1 0
 4 1 1 1 0 1
 5 0 1 0 1 0;
end;
```

Le code fourni comprend également un modèle GLPK un fichier `maxClique.mod`, selon le problème de clique maximale du CM5, suivant un format d'entrée correspondant au format défini par les graphes dans ce TP, tel qu'écrit dans `graph0.dat`. On pourra lancer un calcul par:

```
glpsol -m maxClique.mod -d graph0.dat
```

Le code fourni comprend dans le dossier `Data` des instances sur le même format que `graph0.dat`. Les instances sont nommées sur le format graph-n-a ou graphCompl-n-a. graphCompl-n-a est construit comme étant le graphe complémentaire de graph-n-a, une arrête est présente dans graphCompl-n-a si et seulement si elle n'est pas présente dans graph-n-a.

- $n \in \{20, 30, 50, 70, 80, 100\}$ correspond au nombre de noeuds de l'instance
- $a \in [1; 4]$ correspondent à 4 grands graphes de base

graph-n-a est leur troncature en ne considérant que les noeuds indexés de 1 à n. Les graphes diffèrent par leur densité, le pourcentage d'arêtes présentes:

- $a = 1$ et $a = 2$ correspondent à des graphes graph-n-a très denses, densité de 90%. Les graphes graphCompl-n-a sont alors très épars, densité de 10%.
- graph-n-3 est dense, densité de 70%. Les graphes graphCompl-n-3 sont alors épars, densité de 30%.
- $a = 4$ correspond à une densité d'arêtes de 50% pour graph-n-4 comme graphCompl-n-4.

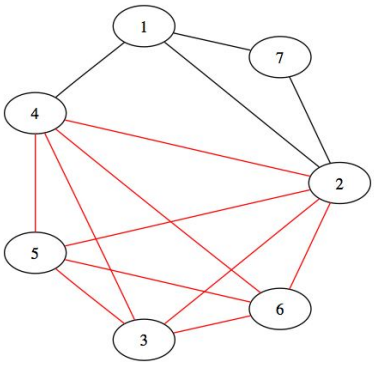
On observera que suivant les modèles PLNE, les tailles critiques où la résolution PLNE est réalisée dans l'ordre de la seconde est variable, ce sera un élément à analyser.

Enfin, le code fourni comprend un dossier `Data2`, avec des fichiers de données graphColored-n-a, correspondant aux instances graph-n-a avec des données supplémentaires pour le problème de coloration des sommets.

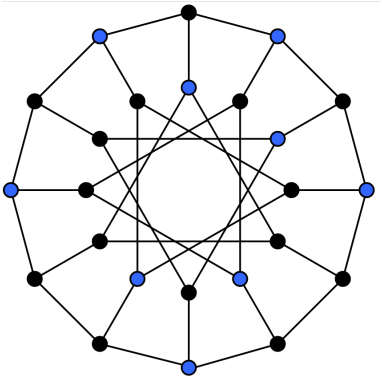
4 Problèmes Stables/cliques max, résolution PLNE

4.1 Problème de stables/cliques max dans un graphe

On rappelle les définitions des problèmes de recherche de cliques et de stables maximaux, illustrées ci dessous:



Clique max: sélectionner un ensemble de sommets de cardinal maximal, tq tous les sommets sélectionnés sont reliés par une arête.



Stable max: sélectionner un ensemble de sommets de cardinal maximal, tq tous les sommets sélectionnés ne sont pas reliés par une arête.

4.2 Modélisation PLNE des problèmes Stables/cliques max dans un graphe

Ces deux problèmes se modélisent de manière similaire en PLNE. On utilise des variables binaires $z_v \in \{0, 1\}$ pour tout $v \in V$, où $z_v = 1$ si le sommet v est considéré dans le stable (ou la clique). Pour maximiser le cardinal d'un sous ensemble de V, on maximise l'expression linéaire $\sum_{v \in V} z_v$.

Les contraintes sur les variables z_v définissant que le sous-ensemble est un stable s'écrivent:

$$\forall e = (v_1, v_2) \in E, z_{v_1} + z_{v_2} \leqslant 1 \tag{1}$$

Les contraintes sur les variables z_v définissant que le sous-ensemble est une clique s'écrivent:

$$\forall e = (v_1, v_2) \notin E, z_{v_1} + z_{v_2} \leqslant 1 \tag{2}$$

Cela donne les modélisations PLNE suivantes, pour les problèmes de recherche de stables maximaux et de cliques maximales dans un graphe:

$$\begin{array}{lll} \text{Stable Max:} & \max\limits_{z \in \{0,1\}^{|V|}} & \sum\limits_{v \in V} z_v \\ s.c: & \forall e = (v_1, v_2) \in E, & z_{v_1} + z_{v_2} \leqslant 1 \end{array} \tag{3}$$

$$\begin{array}{lll} \text{Clique Max:} & \max\limits_{z \in \{0,1\}^{|V|}} & \sum\limits_{v \in V} z_v \\ s.c: & \forall e = (v_1, v_2) \notin E, & z_{v_1} + z_{v_2} \leqslant 1 \end{array} \tag{4}$$

4.3 Résolution par GLPK

Le modèle fourni `maxClique.mod` implémente sous GLPK et selon la formulation PLNE, le problème de clique maximale (4).

```
param nbVertex, integer;
set VERTICES :=1..nbVertex;
param graph{i1 in VERTICES, i2 in VERTICES};

var x{i in VERTICES}, binary;

maximize obj :
 sum{i in VERTICES} x[i];

s.t. C_clique {i1 in VERTICES, i2 in VERTICES : i1<i2 && graph[i1,i2]<0.5}:
x[i1]+x[i2] <= 1;

solve;
end;
```

Pour cela on utilise `graph[i1,i2]<0.5` pour tester si $i1$ et $i2$ sont reliés dans le graphe. Les contraintes sur les variables z_v définissant que le sous-ensemble est une clique peuvent se formuler comme une contrainte:

$$\forall v_1, v_2 \in V, \qquad \text{graph[v1,v2]} = 0 \implies z_{v_1} + z_{v_2} \leqslant 1 \tag{5}$$

Question 1 : Identifier pour quelles valeurs de n les 48 instances considérées permettent d'être résolues à l'optimalité en moins de 10 secondes avec `maxClique.mod`, et sur quelles valeurs de n cela devient plus difficile. On reportera les valeurs prouvées optimales, les temps de calculs et nombres de noeuds de l'arbre B&B nécessaires à la convergence de GLPK pour les deux valeurs de n critiques suivant l'analyse ci dessus. La densité du graphe a-t-elle un impact sur les capacités de résolution par GLPK?

Question 2 : Implémenter dans un fichier `maxStable.mod` le modèle PLNE définit par la formulation (3), et reprendre les questions de la question 1.

La recherche d'une clique maximale dans un graphe est équivalente à la recherche d'un stable maximal dans le graphe complémentaire. Des résultats des questions 1 et 2 doivent concorder sur les valeurs des optimums. Vérifiez que c'est bien le cas, dans le cas contraire, vous avez une erreur à déboguer.

Dans la suite, on considérera uniquement le problème de recherche de stable maximal.

Question 3 : Calculer les valeurs des relaxations continues dans le modèle `maxStable.mod` pour les graphes assez denses. Y a t'il une règle générale? Pouvez-vous justifier que les valeurs de relaxation continues seront toujours supérieures à une valeur facile à calculer?

Sur les problèmes de sac à dos, on avait vu que les coupes de couvertures peuvent améliorer les qualités des relaxations continues. Pour le problème max-stable, les *coupes de clique* jouent un rôle similaire. L'annexe fourni précise ces coupes de manière générale, on regarde ici uniquement les cliques de taille 3.

Si on a trois sommets, $v_1, v_2, v_3 \in V$ tous reliés entre eux deux à deux, i.e tels que $(v_1, v_2) \in E$, $(v_1, v_3) \in E$ et $(v_2, v_3) \in E$, on a la contrainte, $z_{v_1} + z_{v_2} + z_{v_3} \leqslant 1$, un seul de ces trois éléments peut être sélectionné. Il en résulte un nouveau PLNE où on considère ces contraintes additionnelles, qui ne changent rien à la réalisabilité des points entiers, mais peut potentiellement couper des solutions fractionnaires.

$$\begin{array}{lll} \text{Stable Max:} & \max\limits_{z\in\{0,1\}^{|V|}} & \sum\limits_{v\in V} z_v \\ s.c: & \forall e=(v_1,v_2)\in E, & z_{v_1}+z_{v_2}\leqslant 1 \\ & \forall (v_1,v_2)\in E, v_3\in V^3, & \text{graph[v2,v3]} = \text{graph[v1,v3]} = 1 \Longrightarrow z_{v_1}+z_{v_2}+z_{v_3}\leqslant 1 \end{array} \tag{6}$$

Question 4 : Implémenter dans un fichier `maxStableV2.mod` le modèle PLNE défini par la formulation (6). On comparera sur les relaxations continues des formulations (3) et (6) les bornes obtenues et les temps de calculs pour le calcul des bornes continues.

N.B: on ne testera pas l'implication de ces coupes sur la résolution entière par Branch&Bound. On pourrait la voir sur des graphes très très denses. De manière générale, on peut expérimenter avec le paramètre GLPK `--clique` pour spécifiquement les coupes de cliques pour une génération dynamique de coupes de cliques. Avec le paramètre `--cuts`, on peut analyser l'intérêt d'en ajouter d'autres en plus des coupes de clique. A noter que par défaut, GLPK ne génère aucune coupe, contrairement aux autres solveurs

Question 5 : Relancer les expériences de la question 2 sur le modèle `maxStableV2.mod` en ajoutant la génération de toutes les coupes, et d'uniquement les coupes de cliques. On reportera les valeurs prouvées optimales, les temps de calculs et nombres de noeuds de l'arbre B&B nécessaires à la convergence de GLPK. Quel est l'impact des différentes coupes?

La question précédente illustre l'impact des coupes dans l'algorithme B&B. Il est à noter qu'avoir au plus tôt une bonne solution réalisable (idéalement l'optimum sans savoir que la solution est effectivement optimale) permet d'activer plus tôt le critère d'élagage et donc d'explorer moins de noeuds pour converger à l'optimalité. De nombreux solveurs proposent des fonctionnalités de démarrage à chaud ("warmstart") permettant de donner au solveur en entrée une solution réalisable. Sinon, des heuristiques existent pour construire une solution réalisable au tout début de l'algorithme B&B, comme "feasibility pump", activée avec GLPK avec le paramètre `--fpump`. Des heuristiques sont aussi implémentées pour pouvoir améliorer de telles solutions tout au long de la recherche B&B, ce qui a un impact sur les temps de calculs pour al résolution à l'optimalité.

La partie suivante propose d'implémenter des heuristiques sur le problème de stable maximum, en utilisant des optimisation locales par résolutions PLNE à l'optimalité. De tels algorithmes peuvent être utilisées comme démarrage à chaud d'une résolution PLNE exacte.

5 De la PLNE exacte aux matheuristiques

Cette partie illustre le cours sur les heuristiques d'optimisation combinatoire, en utilisant un solveur de PLNE tel que GLPK pour des optimisations locales. De telles hybridations entre programmation mathématique et (méta-)heuristiques sont nommées "matheuristiques".
On prendra comme cas d'application le problème de stable maximum.

5.1 Matheuristiques, un aperçu général

La partie précédente et la partie suivante sensibilisent à l'utilisation efficace de solveurs PLNE, en faisant le lien avec la théorie de PLNE. Une telle étape permet d'avoir une résolution exacte à l'optimalité en temps plus court, et de pouvoir résoudre à l'optimalité des instances plus grandes. Cependant, les temps de calculs explosent quand les tailles d'instances augmentent, la résolution exacte peut être inefficace lorsque

l'on souhaite résoudre des instances de grande taille. Au contraire, les approches heuristiques telles que la recherche locale passent bien à l'échelle, mais ne fournissent pas de garantie d'optimalité en général.

Dans ce TP, on utilisera une résolution PLNE à l'optimalité comme opérateur de recherche locale ou de construction gloutonne. Techniquement, cela fera itérer des calculs de solveurs PLNE, dont les résultats des optimisations locales seront utilisées pour les calculs suivants. La section suivante précise des techniques de programmation pour itérer des résolutions PLNE.

Il est à noter que de telles approches matheuristiques sont utilisables pour des problèmes industriels complexes, vous pouvez en voir un aperçu sur l'article suivant:

N. Dupin, E-G Talbi, Matheuristics to optimize refueling and maintenance planning of nuclear power plants. Journal of Heuristics, 27, pp 63-105,2021. `https://doi.org/10.1007/s10732-020-09450-0`

version preprint: `https://arxiv.org/pdf/1812.08598.pdf`

5.2 Itérer des calculs PLNE, python-MIP

Cette section suivante précise des techniques de programmation pour itérer des résolutions PLNE.

On pourrait utiliser GLPK comme précédemment avec le modeleur, en itérant avec un langage de script ou de programmation, par exemple qui lancera des calculs par un appel au terminal système et récupère les sorties intéressantes pour la suite.

Une utilisation de GLPK en API serait plus adaptée, pour éviter ces écritures et lectures de fichiers textes pour les entrée-sorties. Les liens suivants vous aideront à utliser les API Java et C/C++ de GLPK si besoin:
`https://glpk-java.sourceforge.net/`
`https://en.wikibooks.org/wiki/GLPK/Using_the_GLPK_callable_library`

On vous conseille d'utiliser python-MIP, pour avoir les mêmes avantages de concision du modeleur GLPK avec python, et l'intégration dans un langage de programmation comme pour l'utilisation en API, avec une utilisation plus simple que l'API de GLPK. Julia JuMP peut aussi être utilisée, avec des avantages sur python-MIP pour une utilisation en thèse de RO, avec le désavantage que la langage Julia est moins connu que Python, et a des spécificités quand on le découvre.

Julia est un langage plus récent que Python, conçu notamment pour une meilleure efficacité des opérations élémentaires telles que les boucles et pour la programmation parallèle et distribuée. Cela implique des petites spécificités dans le langage Julia, qui reste un langage dont la concision est proche de Python. Julia a une bibliothèque de modélisation générique de PL/PLNE, JuMP, qui permet d'appeler les principaux solveurs libres et commerciaux, et donne accès à des fonctionnalités avancées génériques, quand elles sont implémentées par les solveurs. Par ailleurs, il faut noter que JuMP a des extensions de la PLNE au sommet de l'état de l'art actuel, développées par des laboratoires académiques.

Sous Python, la bibliothèque Python-MIP (de Coin-OR, `https://python-mip.readthedocs.io/en/latest/intro.html`) offre un autre compromis: avoir une bibliothèque de modélisation simple, pouvant appeler uniquement CBC ou Gurobi, c'est-à-dire parmi les meilleurs solveurs libres et commerciaux, en ayant la possibilité d'appeler des fonctionnalités avancées communes aux deux solveurs. PuLP est une autre biblioyhèque générique de modélisation de PL/PLNE, permettant d'appeler tous els solverus principaux, mais avec peu de fonctionalités avancées et une efficacité bien moindre L'efficacité des temps de chargement des modèles avec Python-MIP est aussi améliorée par rapport à PuLP, et de nombreux bénéfices sont apportés par une compilation par Pypy plutôt que par l'interpréteur standard de Python. Ces points sont très importants pour une utilisation avancée de la PLNE.

L'installation de Python-MIP est assez facile, et installe automatiquement une version de CBC:
`https://python-mip.readthedocs.io/en/latest/install.html`
`https://www.python-mip.com/`
`https://python-mip.readthedocs.io/en/latest/examples.html`

La documentation est disponible aux liens suivants:
- `https://python-mip.readthedocs.io/en/latest/quickstart.html`
- `https://python-mip.readthedocs.io/en/latest/classes.html`

5.3 Import de données DIMACS

Pour tester vos heuristiques sur des instances de grande taille, où les résultats sotn de grande qualité comme faisant partie des benchmarks d'instances des approches de la littérature, on utilisera les instances DIMACS du problème de clique maximum, dont les meilleures valeurs sont rapportées ci dessous:
`https://iridia.ulb.ac.be/~fmascia/maximum_clique/DIMACS-benchmark`

Ces instances peuvent se télécharger au format ASCII au lien suivant:
`http://iridia.ulb.ac.be/~fmascia/files/DIMACS_subset_ascii.tar.bz2`

Pour valider vos heuristiques sur le problème de stable maximum, il faudra considérer le graphe complémentaire de es instances. Pour cela, il vous faut réaliser un lecteur de tels fichiers, et prendre le graphe complémentaire.

Question 6 Dans le langage de programmation de votre choix (qui sera utilisé par la suite pour des appels PLNE et pour les heuristiques), créer votre structure de graphe non orienté sur lequel un des heuristiques de stable maximums seront appelées par la suite, et l'import d'une instance DIMACS en considérant le graphe complémentaire. question: parseur DIMACS.

N.B: si vous choisissez C/C++, un code peut vous être fourni pour cette question pour vous permettre de gagner du temps. L'appel à un solveur PLNE depuis C/C++ peut être moins direct à coder pour la suite.

5.4 Matheuristiques gloutonnes déterministes

N.B: pour cette partie, coder a minima les heuristiques gloutonnes sans appel à un PLNE (paramètre $N = 1$)

Des heuristiques gloutonnes s'écrivent naturellement pour le problème de stable maximum. Dans une première famille d'algorithmes glouton, on se donne un ordre de parcours de sommets, et on construit une solution partielle en ajoutant des sommets tant qu'on le peut, ie tant que le sommet candidat n'est relié à aucun des sommets précédemment sélectionnés. L'heuristique gloutonne ne remet pas en question les choix précédemment effectués.

Comme ordre de parcours, il peut être naturel d'itérer suivant les degrés croissants: un sommet peut avoir a priori plus de chance d'être dans un grand stable si son degré est faible. Un tri de complexité $O(V \log V)$ est alors utilisé.

Si une telle stratégie gloutonne peut être toujours optimale sur des problèmes précis (ex: arbre couvrant de poids minimum/maximum ou sac à dos continu), cela n'est pas le cas sur le problème de stable maximum. Une mauvaise anticipation peut être une source d'inefficacité: prendre un sommet peut interdire une solution ou deux autres sommets suivants peuvent être ajoutés.

Pour analyser de tels effets, on fera des optimisations locales plus grandes: on considère les N premiers sommets possiblement à ajouter dans la liste des sommets sélectionnés, en retirant ceux qui ont au moins un voisin dans les sommets sélectionnés, et on utilise une optimisation PLNE pour en sélectionner un nombre maximum. Ces optimisations locales itérées considèrent au plus N sommets, N peut être calibré pour être le plus grand possible en permettant une résolution PLNE rapide (de l'ordre de la seconde ou suivant le temps de calcul autorisé)

Question 7 Implémenter les heuristiques avec et sans PLNE et analyser les résultats sur les instances DIMACS. On montrera les qualités de solution, les temps de calculs, et pour le nombre d'appels au solveur PLNE (que l'on comparera avec le nombre théorique maximal V/N)

Des versions dites "adaptatives" de ces heuristiques peuvent être réalisées, le critère du degré le plus bas évolue lorsque l'on ajoute des sommets dans la liste des sommets sélectionnés, où que des sommets doivent être retirés car voisin d'au moins un sommet de la liste. Lorsque des sommets sont sélectionnés, cela met à jour la liste des candidats restants, et le critère du plus petit degré devrait se faire sur le degré dans le graphe résiduel en ne considérant que les candidats restants, ie en enlevant les sommets sélectionnés et leurs voisins.

Question 8 Coder les versions adaptatives des algorithmes gloutons de la question 7. On prendra soin de mettre à jour efficacement les degrés résiduels, un recalcul "from scratch" des degrés n'est pas efficace, et on précisera la méthode de recalcul des degrés.

5.5 Matheuristiques gloutonnes randomisées

N.B: la question 9 moins importante, peut être sautée, ou faite rapidement.

Sur les heuristiques des questions 7 et 8, le critère de parcours par degré croissant n'est pas toujours pertinent. on peut même construire des instances de stable maximums avec des solutions optimales évitant tous les sommets de degrés les plus faibles. Une autre faiblesse de ces heuristiques précédentes est que si on peut avoir rapidement une solution, le processus est déterministe. Avec une heuristique randomisée, on pourrait lancer plusieurs heuristiques et avoir des résultats différents, pour prendre la meilleure solution sur plusieurs telles heuristiques itérées.

Les heuristiques des questions précédentes avec un ordre de parcours faisant intervenir des tirages aléatoires. On peut par exemple utiliser des ordres aléatoires de parcours. Pour créer une permutation aléatoire, on peut tirer aléatoirement des grands entiers pour chaque sommet du graphe, et les trier selon la valeur aléatoire tirée.

On peut aussi mixer de tels critères aléatoires avec le critère des degrés. Si $N_{\min}, N_{\max}$ sont les degrés minimaux et maximaux des sommets du graphe, on peut tirer pour chaque sommet v un entier dans $r_v = \{0, \dots \frac{1}{2}(N_{\max} - N_{\min})$ et considérer comme ordre de parcours glouton l'ordre $r_v + N_v$ croissant.

Pour valider une heuristique randomisée, on effectue un nombre significatif de tirage (ici 30), la dispersion des résultat importe autant voir plus que le meilleur résultat trouvé. On reporte la meilleure solution trouvée sur les runs, la valeur moyenne, et on peut fournir des indicateurs statistiques comme l'écart type ou des quartiles pour montrer la dispersion.

Question 9 Coder des versions randomisées des heuristiques gloutonnes précédentes et montrer la dispersion des résultats sur 30 runs indépendants.

5.6 Matheuristiques de recherche locale taboue

Dans cette partie, on s'intéresse à des approches de recherche locale, partant d'une solution existante, et modifiant localement cette solution pour explorer plusieurs optimums locaux. Un point clé est la définition du voisinage, on utilisera dans la suite des voisinages $\mathcal{N}(x)$, qui à un stable x, associe tous les stables obtenus en ajoutant des sommets à la solution courante x. On pourra restreindre ce voisinage à un type de sommets: $N_1(x)$ définissant l'ensemble des stables issus de x en ajoutant au plus un sommet, $N_k(x)$ définissant l'ensemble des stables issus de x en ajoutant au plus k sommets, $N_V(x) = \mathcal{N}(x)$.

On peut initialiser la recherche locale en utilisant une des heuristiques précédentes, ou on peut partir d'une sélection nulle de sommets.

La recherche locale la plus simple, Hill climbing (HC) itère des modifications locales, en explorant le voisinage de la solution courante et en prenant une des meilleures solutions du voisinage, et converge vers un maximum local (écrite ici en maximisant une fonction f):

- Paramètres: solution initiale $x_i \in X$, on se donne un voisinage $\mathcal{N}$
- Initialisation: $x = x_i$, $f = f(x)$
- TANT QUE (critère d'arrêt temps ou nombre d'itération atteint)
- On parcourt tout le voisinage $\mathcal{N}(x)$ en calculant les $f(x_0)$
- Existe t'il une solution $x_0 \in X$ telle que $f(x_0) > f(x)$?
- Si NON, on sort de la boucle TANT QUE
- Si OUI, on actualise $x = \text{argmax}_{x_0 \in \mathcal{N}(x)} f(x_0)$ et $f = f(x)$.
- FIN TANT QUE
- On renvoie $x, f(x)$

Une telle heuristique peut être codée facilement avec le voisinage $\mathcal{N}_1$, l'exploration du voisinage étant alors en $O(V)$. Avec des voisinages $\mathcal{N}_k$, on peut utiliser un calcul de PLNE pour trouver la meilleure solution du voisinage. Comme variante d'Heuristique Hill Climbing, on n'explore pas tout le voisinage, on peut s'arrêter à la première solution améliorante, ou prendre la meilleure solution trouvée sur un critère d'arrêt défini si une solution améliorante a été trouvée (ex: une fois que n solutions améliorantes ont été trouvées, ou sur n explorations de solutions voisines.) On notera que les heuristiques gloutonnes précédentes sont proches de tels algorithmes Hill Climbing, en partant d'une solution sans sommet, et convergent alors vers un maximum local du voisinage $\mathcal{N}$.

En utilisant la PLNE, on définit la taille du voisinage suivant l'efficacité de la PLNE sur de telles tailles de problèmes. Techniquement, cela fait itérer des calculs de PLNE, l'outillage précédemment décrit est adapté.

Sur le problème de stable maximum, une difficulté est que les solutions améliorantes du voisinage $\mathcal{N}_1$ ont toutes le même coût, le choix du voisin est aléatoire, la fonction objectif ne discrimine pas quel voisin choisir. Avec des voisinages $\mathcal{N}_k$, on discrimine mieux les directions prometteuses, mais de nombreuses égalités peuvent demeurer.

On utilisera dans la suite la recherche taboue pour explorer plusieurs maximums locaux, et ne pas explorer toujours les mêmes maximums locaux. Une première version de recherche taboue peut d'écrire de cette manière, avec L une liste taboue comportant différents stables, qui seront évités:

- On définit L une liste taboue qui correspond à un ensemble de points $x \in X$.
- Idée: dans Hill Climbing, on ne parcourt que $\mathcal{N}(x) - L$, on ignore les éléments de la liste taboue.
- Quand on tombe dans un minimum local $x \in X$, on actualise la liste taboue $L = L \cup \{x\}$, et on actualise x un autre point du voisinage.

N.B: la taille de la liste taboue est un paramètre important: pour ne pas trop alourdir les calculs, on fixe une taille maximale de liste taboue, au bout d'un certain temps, on s'est éloigné des premiers minimas locaux, ces tabous peuvent être inutiles. La liste taboue est alors plutôt une file FIFO, une implémentation comme une liste doublement chaînée peut être adéquate.

L'implémentation et l'encodage de la liste taboue sont des éléments importants pour la performance de l'heuristique, n'hésitez pas à échanger sur ce sujet avant de vous lancer dans une implémentation.

Une telle heuristique taboue peut être codée sans PLNE avec les voisinages $\mathcal{N}_1$. Pour utiliser efficacement des voisinages PLNE, on utilisera plusieurs adaptation de la recherche taboue. Avec des plus grands voisinages, on pourrait utiliser la PLNE, mais de manière moins efficace

Une première adaptation est d'avoir dans la liste taboue des sommets qu'on n'autorise plus à ajouter, et pas une solution complète. Lorsqu'on est dans un maximum local, il faut retirer des sommets et les ajouter dans la liste taboue. Ce choix se fait aléatoirement, et peut permettre d'ajouter alors des sommets. Si trop peu de sommets redeviennent disponibles (car non liés aux sommet restants sélectionnés), l'optimisation PLNE est de petite taille. On peut itérer l'ajout de sommets dans la liste taboue tant que le nombre de sommets à nouveau disponible n'atteint pas un seuil où la résolution PLNE est efficace. Avec le calcul PLNE sur les sommets sélectionnables, concaténé avec les sommets courants non sélectionnés, on obtient une autre solution. Un tel stable n'est pas forcément un maximum local, des éléments de la liste taboue pourraient compléter cette autre solution. On peut vérifier si des sommet de la liste taboue peuvent être ajoutés dans la nouvelle solution, un algorithme glouton ou un calcul exact PLNE permet obtenir un maximum local. Ainsi, l'algorithme permet d'explorer ma modification locale de nombreux maximums locaux.

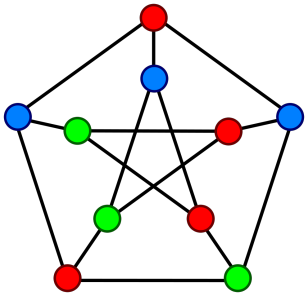
Question 10 Implémenter une ou plusieurs recherche taboues, montrer la dispersion des recherches taboues randomisées. On précisera les paramètres électionnés(nombre d'itérations, encodage et taille de liste taboue, opérateurs spécifiques pour matheuristique ...). Est ce que l'amélioration est significative en partant d'un glouton, les qualités de solutions se rapprochent elles des meilleures solutions connues?

6 Problème de coloration "Vertex Coloring Problem" (VCP)

Cette partie illustre l'impact de choix de modélisations différents sur l'efficacité de la résolution PLNE. Plusieurs variantes de formulations PLNE peuvent être écrites pour le problème de coloration "Vertex Coloring Problem" (VCP). L'impact des symétries sur l'efficacité de la résolution et uen manière de les casser sont présenées en PLNE, cela doit faire écho à des préoccupations similaires en Programmation par Contraintes (PPC).

6.1 VCP, première modélisation PLNE

On rappelle la définition du problème de coloration des sommets:
On veut colorer les sommets du graphe avec un nombre minimal de couleurs sachant que deux voisins auront une couleur différente.



On utilise dans la première formulation PLNE C, un majorant du nombre de couleurs. $C = |V|$ convient, ce qui est un majorant très pessimiste, tous les sommets sont alors colorés de de couleurs différentes, ce qui est le cas uniquement du graphe complet. Une heuristique telle que DSATUR, RLF ou Welsh-Powell, permettent une telle initialisation pour une valeur C, majorant du nombre de couleur optimal. Dans la suite, on analysera l'intérêt d'avoir une valeur de C aussi petite que possible.

On notera C_{opt} le nombre de couleur optimal, c'est la valeur optimale pour le choix d'un majorant C.

On utilise dans cette première formulation des variables binaires $z_{v,c} \in \{0,1\}$, $y_c \in \{0,1\}$ pour $v \in V$ et $c \in [\![1;C]\!]$; $z_{v,c} = 1$ ssi le sommet v est coloré avec la couleur c, $y_c = 1$ ssi la couleur c est utilisée.

En cherchant à minimiser le nombre de couleurs utilisées, on minimise $\sum_{c=1}^{C} y_c$.

Le lien entre $z_{v,c}$ et y_c, peut s'écrire $y_c = 0 \Longrightarrow z_{v,c} = 0$ pour tout v, si une couleur est non activée dans y_c, aucun sommet ne peut être coloré avec cette couleur. En PLNE, cela peut s'écrire:

$$\forall c \in [\![1;C]\!], \forall v \in V, z_{v,c} \leqslant y_c$$

La contrainte de couleurs différentes pour des voisins est assez analogue à celles de MaxStable:

$$\forall c \in [\![1;C]\!], \forall e = (v_1, v_2) \in E, z_{v_1,c} + z_{v_2,c} \leqslant 1$$

Enfin, il faut écrire la contrainte que tous les sommets sont colorés. En l'absence de cette contrainte, la solution optimale pour minimiser le nombre de couleurs est de n'affecter aucune couleur aux sommets. Pour tout sommet v, il y a une couleur d'affectée, il y a a exactement une variable $z_{v,c}$ valant 1. En PLNE, cela se linéarise:

$$\forall v \in V, \sum_{c=1}^{C} z_{v,c} = 1$$

N.B: on pourrait aussi avoir une inégalité $\geqslant 1$, cela ne change rien à l'optimum. Si on a une solution avec plusieurs couleurs affectées à un sommet, en post-traitement, on peut en retirer, et ne garder qu'une couleur par sommet pour définir une coloration optimale. Cette transformation inégalité/égalité, peut avoir des influences en résolution PLNE, en général, on préfère les inégalités

Au final, on a la formulation PLNE suivante pour le problème VCP :

$$\begin{array}{lll} & & \min \sum_{c=1}^{C} y_c \\ s.c: & \forall v \in V, & \sum_{c=1}^{C} z_{v,c} \geqslant 1 \\ & \forall c \in [\![1;C]\!], \forall v \in V, & z_{v,c} \leqslant y_c \\ & \forall c \in [\![1;C]\!], \forall e=(v_1,v_2) \in E, & z_{v_1,c} + z_{v_2,c} \leqslant 1 \\ & \forall c \in [\![1;C]\!], \forall v \in V, & z_{v,c} \in \{0,1\} \\ & \forall c \in [\![1;C]\!], & y_c \in \{0,1\} \text{ (ou } \geqslant 0) \end{array} \tag{7}$$

Au final, on a la formulation PLNE suivante pour le problème VCP :

$$\begin{array}{lll} & & \min y \\ s.c: & \forall v \in V, & \sum_{c=1}^{C} z_{v,c} \geqslant 1 \\ & \forall c \in [\![1;C]\!], \forall v \in V, & c \times z_{v,c} \leqslant y \\ & \forall c \in [\![1;C]\!], \forall e=(v_1,v_2) \in E, & z_{v_1,c} + z_{v_2,c} \leqslant 1 \\ & \forall c \in [\![1;C]\!], \forall v \in V, & z_{v,c} \in \{0,1\} \\ & & y \geqslant 0 \end{array} \tag{8}$$

Question 11: Implémenter la formulation PLNE (7) en prenant $C = |V|$ dans le modèle GLPK `color0.mod`. A quelle taille de graphe ce modèle PLNE permet t'il de fournir des solutions prouvées optimales?

Question 12: Implémenter la formulation PLNE (8) en prenant $C = |V|$ dans le modèle GLPK `color02.mod`. Comment se comapre cette formulation par rapport à a question 11? On reportera le nombre de variables et de contraintes, les temps de caluls et le nombre de noeuds B&B pour converger. On choisira une formulation PLNE pour la suite.

6.2 Modélisation PLNE et symétries

Sur les formulations PLNE précédentes, de nombreuses solutions symétriques existent. A partir d'une solution optimale, en permutant les indices des couleurs sans changer l'affectation en sous-ensembles, on forme de nouvelles solutions optimales symétriques. Dans la suite, on réduira les symétries en rajoutant des contraintes dans les formulations PLNE précédentes. Pour éviter des permutations de couleurs, on peut fixer la couleur d' un premier sommet v, ce sommet étant à choisir. En PLNE, cela revient ajouter une contrainte $z_{v,1} = 1$. Si v a un voisin v', la couleur de v' est nécessairement différente de 1, on peut par exemple lui affecter la couleur 2 et imposer $z_{v',2} = 1$.

En fixant une ou deux couleur(s), il reste beaucoup de permutations symétriques. Pour aller plus loin dans cette logique, on cherche une grande clique dans le graphe, la plus grande qu'on puisse trouver, de cardinal K (par une méthode heuristique, ou avec un calcul exact Branch&Bound en temps restreint). Dans une clique, toutes les couleurs sont différentes on affecte les couleurs de 1 à K pour tous les sommets de la clique.

Dans le dossier `Data2`, un graphe est donné comme précédemment, et on rajoute des informations supplémentaires: la donnée d'un nombre de couleurs réalisable pour la coloration par sommet (en fait optimal), la taille maximale d'une clique dans le graphe (qui est donc un minorant de l'optimum, qui se calcule assez bien sur les tailles de graphes considérées comme vous avez pu l'observer sur la partie 1 du TP).

Question 13: Implémenter un modèle GLPK prenant en entrée des données de `Data2`, en considérant la formulation PLNE (8) en prenant $C = |V|$ comme précédemment et en affectant des couleurs sur la clique donnée en entrée. On nommera ce modèle GLPK `colorClique.mod`. L'accélération de la résolution PLNE est-elle significative?

Précision: dans `cliqueColors`, chaque indice ayant une valeur non nulle est inclus dans la clique pré-calculée, une valeur strictement positive est unique dans `cliqueColors` et correspond à la couleur que l'on va affecter au sommet considéré dans la résolution PLNE. ainsi, pour un sommet v tel que `cliqueColors`$[v] = c_v > 0$, l'affectation de couleur peut s'écrire avec une contrainte linéaire additionnelle $z_{v,c_v} = 1$ ou $z_{v,c_v} \geqslant 1$.

Question 14: Implémenter deux modèles GLPK prenant en entrée des données de `Data2`, en considérant la formulation PLNE (7) avec l'affectation de couleurs sur la clique donnée en entrée, en prenant comme valeurs de C: $C = C_{opt}$ et $C = C_{opt} + 1$. On nommera ces deux modèles GLPK `color2Copt.mod` et `color2Copt1.mod`. Comparer ces deux résolutions avec `colorClique.mod` montre l'impact d'avoir initialisé avec une bonne solution en entrée, qui peut être fournie avec une bonne heuristique. Cet impact est il important en pratique sur les données considérées?

6.3 Formulation alternative en PLNE

Cette dernière formulation présente de nombreuses symétries, notamment en permutant uniquement les couleurs sans changer aux partitions des sommets. La formulation par représnetant suivante casse naturellement des symétries.

Formulation par représentants: on utilise des variables binaires $z_{v,v'} \in \{0,1\}$, pour tout $v, v' \in V$ avec $v \leqslant v'$. $z_{v,v'} = 1$ ssi v et v' sont de la même couleur, de "représentant" v, v est l'indice minimal de sa couleur.

$$
\begin{array}{lll}
 & \min\limits_{z} \sum\limits_{v \in V} z_{v,v} & \\
s.c: & \sum\limits_{v' \leqslant v} z_{v',v} \geqslant 1 \quad (\text{ou} \quad = 1\ ) & \forall v \in V \\
 & z_{v,v_1} + z_{v,v_2} \leqslant z_{v,v} & \forall e = (v_1, v_2) \in E, \forall v \leqslant \min(v_1, v_2), \\
 & z_{v,v'} \in \{0,1\} & \forall v \leqslant v',
\end{array}
\tag{9}
$$

Question 15: Implémenter la formulation PLNE (9) dans le modèle GLPK `colorRep.mod`. A quelle taille de graphe ces modèle PLNE permettent de fournir des solutions optimales? Comment se situent les résultats par rapport aux autres formulations précédentes?