Introduction

L'algorithme que Cplex utilise pour résoudre un programme en variables mixtes entières est un algorithme de branch-and-cut. Cet algorithme explore un arbre des relaxations linéaires du problème initial. Plus précisément, à chaque étape de son algorithme, Cplex sélectionne un noeud de l'arbre, résout la relaxation continue du problème à ce noeud, tente de générer des plans coupants pour couper cette solution courante, fait appel à une heuristique pour tenter de déterminer une solution entière proche de la solution continue courante, sélectionne une variable de branchement (qui a une valeur fractionnaire dans la solution courante alors qu'elle devrait être entière), et finalement crée deux noeuds résultant des arrondis inférieurs et supérieurs de la valeur de la variable de branchement.

L'API C++ de Cplex fournit un mécanisme de callback pour permettre à l'utilisateur d'intervenir pendant chacune des étapes de l'algorithme de branch-and-cut. Si l'utilisateur code une fonction en utilisant ce mécanisme, Cplex invoquera cette fonction pendant le branch-and-cut.

Dans le cadre de ce dernier TP (Dont vous devriez rendre le projet), nous souhaitons ajouter des inégalités valides dans les noeuds du branch-and-cut, afin de renforcer la valeur de la relaxation continue. Pour cela nous utiliserons la macro `ILOUSERCUTCALLBACKI (nomCallback, type1, nom1, ..., typeI,nomI)`. Cette dernière crée deux choses :

- Un objet de la classe `UserCutCallbackI` nommé nomCallback
- Un constructeur `nomCallback(env, nom1, ..., nom7)` qui crée une instance de cette classe, où `env` est l'environnement courant. L'objet peut ensuite être utilisé via la méthode `use` de la classe `IloClpex`.

A l'intérieur du bloc de la macro il faut utiliser les routine de l'API C++ pour obtenir par exemple la valeur des variables (`IloNum a = getValue(x[i]);`) au noeud courant. Ensuite, il faut invoquer une fonction utilisateur qui permet d'identifier les inégalités valides violées que l'on souhaite ajouter au modèle. Ces inégalités valides sont ensuite ajoutées globalement via la méthode `add(cut)` . Il est possible d'ajouter des inégalités valides uniquement localement (c'est-à-dire uniquement dans le sous-arbre du noeud courant) via la méthode `addLocal(cut)`. Voici un exemple d'utilisation :

```
// 0 <= I <= 7
// ILOUSERCUTCALLBACKI (nomCallback, type1, nom1, ..., typeI,nomI)
ILOUSERCUTCALLBACK3(CtCallback, IloExprArray, lhs, IloNumArray, rhs, IloNum, eps) {
	IloInt n = lhs.getSize();
	for (IloInt i = 0; i < n; i++) {
		if (rhs[i] < IloInfinity && getValue(lhs[i]) > rhs[i] + eps) {
			IloRange cut;
			try {
				cut = (lhs[i] <= rhs[i]);
				add(cut).end();
			} catch (...) {
				cut.end();
				throw;
			}
		}
	}
}
// fonction utilisateur qui prépare les arguments pour la macro
void makeArguments(const IloNumVarArray vars, IloExprArray lhs, IloNumArray rhs) {
	lhs.add(vars[0] - vars[1]);
	rhs.add(0.0);
	lhs.add(vars[1] - vars[2]);
	rhs.add(0.0);
```

```
}
int main(int argc, char** argv) {
	IloEnv env;
	try {
		IloModel m(env);
		IloCplex cplex(env);
		IloObjective obj;
		IloNumVarArray var(env);
		IloRangeArray con(env);
// Création du modèle
		...
// création du callback
		IloExprArray lhs(env);
		IloNumArray rhs(env);
		makeArguments(var, lhs, rhs);
		cplex.use(CtCallback(env, lhs, rhs, cplex.getParam(IloCplex::EpRHS)));
		cplex.setParam(IloCplex::MIPInterval, 1000);
		cplex.setParam(IloCplex::MIPSearch, IloCplex::Traditional);
		env.out() << "solving model ...\n";
		cplex.solve();
		env.out() << "solution status is " << cplex.getStatus() << endl;
		env.out() << "solution value is " << cplex.getObjValue() << endl;
	} catch (IloException& ex) {
		cerr << "Error: " << ex << endl;
	}
	env.end();
	return 0;
}
```

Exercice 1. Génération de coupes

Une entreprise possède 15 sites qu'elle désire interconnecter par un réseaux de communication. Il est possible de créer des liens directs entre certaines paires de sites mais pas toutes. Le graphe si dessous montre les paires de sites pouvant être connectés directement. Sur chacun de ces liens il est possible d'installer plusieurs câbles sachant que chaque câble offrira une capacité C (en Gb/s) au lien. Chaque site v a besoin d'avoir un débit de 1 Gb/s vers chaque autre site w. Nous supposons que chaque câble installé coutera la même chose à l'entreprise. Cette dernière veut donc minimiser le nombre des câbles installés tout en respectant les demandes.

Question 1. Est il possible de prévoir la solution pour un C très grand ?

Question 2. Ecrire le PLNE qui permet de résoudre ce problème. Utiliser la technologie concert pour le résoudre avec Cplex et donner la solution pour chaque capacité C dans l'ensemble $\{1, 5, 10, 15, 20, 40, 60, 100\}$.

Question 3. Imaginer un algorithme combinatoire capable de résoudre le cas $C = 1$. Décrire ce dernier en quelques lignes.

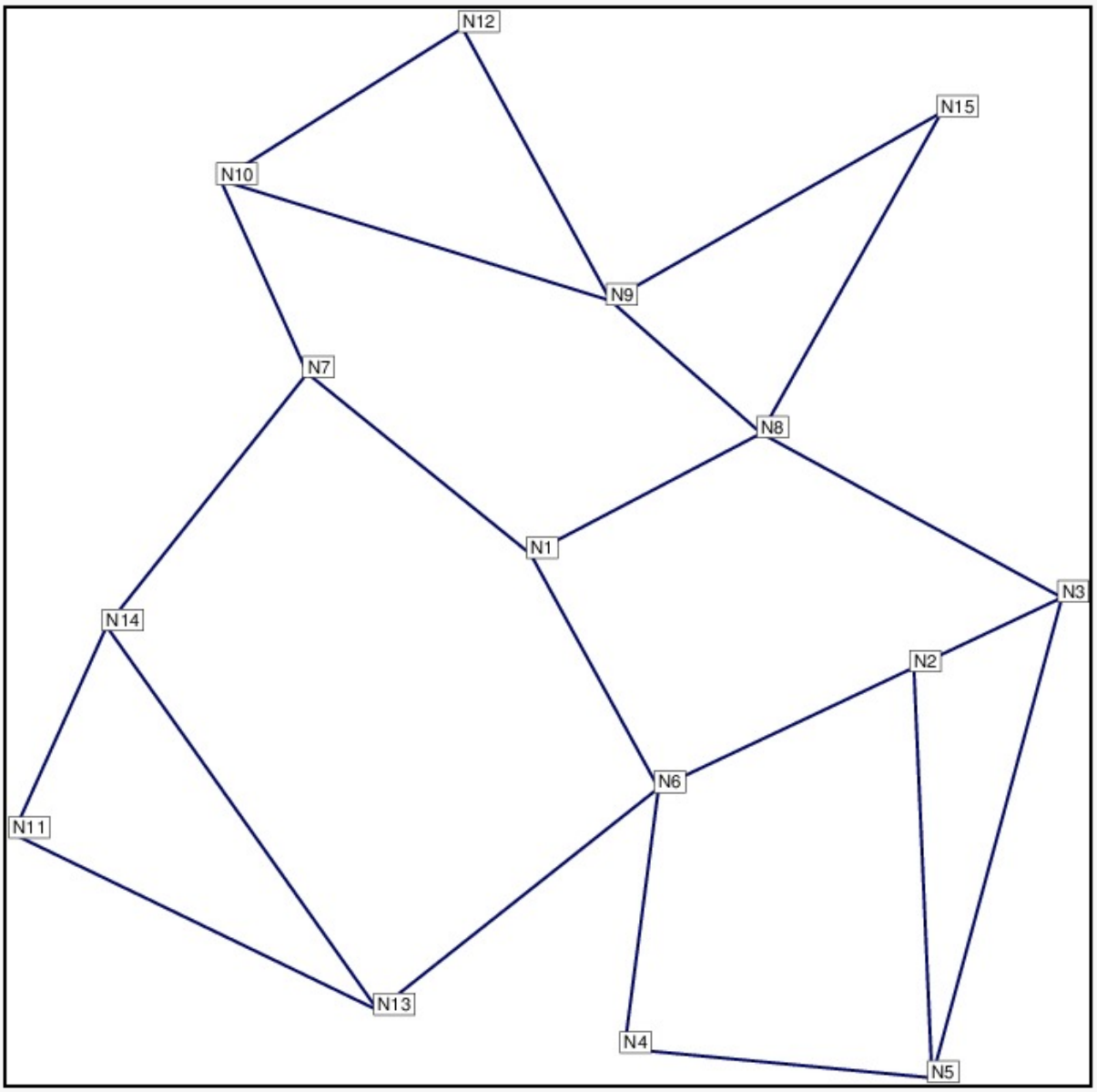
Question 4. Est ce qu'il est possible d'avoir une solution faisable dont la valeur objective est 13 ? Déduire une inégalité valide. A l'aide des `UserCutCallbackI`, ordonner à Cplex de rajouter cette inégalité (de manière locale) à chaque fois qu'elle est violée.

Question 5. En considérant un site particulier v, déduire une inégalité valide qui contraint le nombre des câbles installés sur les liens adjacents à ce site. A l'aide des `UserCutCallbackI`, ordonner à Cplex de rajouter ces inégalités (de manière globale et non locale) à chaque fois qu'une inégalité de ce type est violée.

Question 6. Considérons une bipartition arbitraire des sites $\{S, \bar{S}\}$. Soit E l'ensemble minimum des liens qui déconnecte complètement l'ensemble S de $\bar{S}$. Déduire une inégalité valide qui contraint le nombre des câbles installés sur les liens de E. Ordonner à Cplex de rajouter (de manière globale) à chaque *callback* uniquement l'inégalité la plus violée parmi les inégalités de ce type.

Question 7. (Cette question est optionnelle). On considérons cette fois les tripartitions des sites, est il possible de déduire des nouvelles inégalités valides non redondantes avec les inégalités décrites précedemment ?

Question 8. Ecrire un rapport où vous présentez la valeur ajoutée de chaque type des inégalités testées, ainsi que leurs combinaisons, sur l'ensemble des capacités $\{1, 5, 10, 15, 20, 40, 60, 100\}$.



Annexe : structure de graphe

```
struct Graph {

        int vertexCount;
    int edgeCount;
        int** adjacencyMatrix;

        Graph(int vertexCount) {
                this->vertexCount = vertexCount;
                this->edgeCount = 0;
                adjacencyMatrix = new int*[vertexCount];
                for (int i = 0; i < vertexCount; i++) {
                        adjacencyMatrix[i] = new int[vertexCount];
                        for (int j = 0; j < vertexCount; j++)
                                adjacencyMatrix[i][j] = -1;
                }
        }

        void addEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount) {
                        int edgeN = edgeCount++;
                        adjacencyMatrix[i][j] = edgeN;
                        adjacencyMatrix[j][i] = edgeN;
                }
        }

        void removeEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount) {
                        adjacencyMatrix[i][j] = -1;
                        adjacencyMatrix[j][i] = -1;
                }
        }

        bool isEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount)
                        return adjacencyMatrix[i][j] != -1;
                else
                        return false;
        }

        int getEdge(int i, int j) {
                if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount)
                        return adjacencyMatrix[i][j];
                else
                        return -1;
        }

        ~Graph() {
                for (int i = 0; i < vertexCount; i++)
                        delete[] adjacencyMatrix[i];
                delete[] adjacencyMatrix;
        }
};
```