Sorbonne Université

UE de Projet d'Ingénierie

MU5IN556

Projet d'ingénierie : Gestion des voeux d'UE

Auteur :
Ingrid theret

Encadrant :
Antoine genitrini

26 février 2024

Table des matières

Introduction

Pour ce projet, on a travaillé sur une application de gestion des voeux des étudiants afin de garantir une charge équilibrée au sein de chaque UE tout en se basant autant que possible sur les voeux des étudiants. Celle-ci présentait des bugs et manquaient quelques fonctionnalités.
Par ailleurs, elle repose sur le solveur de contrainte fourni par Gurobi qui nécessite une licence payante afin de le faire fonctionner. On souhaite donc réécrire la partie logicielle des gestion des voeux, en se basant sur le coeur existant qui permet de faire fonctionner ce solveur.

1 Application de base

L'application de base a été réalisé par Adan Bougherara et Vivien Demeulenaere dans le cadre de l'UE projet de première année de Master Androïde à Sorbonne Université. Ce qui est décrit ci-dessous résume leur rapport qui a été fourni afin de comprendre l'objectif du projet présent et de comprendre le fonctionnement de l'application.

1.1 Fonctionnement de l'application

Pour le lancement de l'application, il est nécessaire de posséder `python3` avec le module `Tkinter` pour l'interface graphique ainsi qu'une licence Gurobi valide pour le module `gurobipy`. On tape la commande `python gestionnaire_de_voeux.py` qui permet l'ouverture de la fenêtre.

Sur la figure 1, on a décrit simplement comment se décompose le programme avec également des indications sur les fichiers qui seront modifiés ou créés au cours de ce projet :

- `gui.py` est le fichier permettant d'obtenir l'interface graphique, il permet d'associer à chaque menu/bouton une action et d'afficher les résultats. Il s'agit du fichier principal où les modifications seront nombreuses
- `traitement.py` lit les fichiers ligne par ligne et renvoi un dictionnaire d'ue qui sont considérés comme des objets créées par `ue.py` et une liste d'étudiants, également des objets créés par `etudiant.py`
- `parametres.py` contient les paramètres (texte à afficher, langue utilisé,...).
- `outils.py` pour des fonctions qui peuvent être réutilisée dans n'importe quel fichier, contient principalement des fonctions de lecture et d'écriture de fichiers.
- `erreurs.py` pour la création de nouvelles erreurs
- `solveur.py` contient le solveur de contrainte de Gurobi auquel on peut ajouter de nouvelles contraintes spécifiques listés dans `contraintesPersonnalisees.py`.

Les trois autres fichiers ont été créé lors de ce projet et permettent de faire la corrections de bugs (`verification.py`) et d'avoir un nouveau solveur de contraintes (`solveur_glpk.py` avec `bouchon.py` qui sert aux tests)

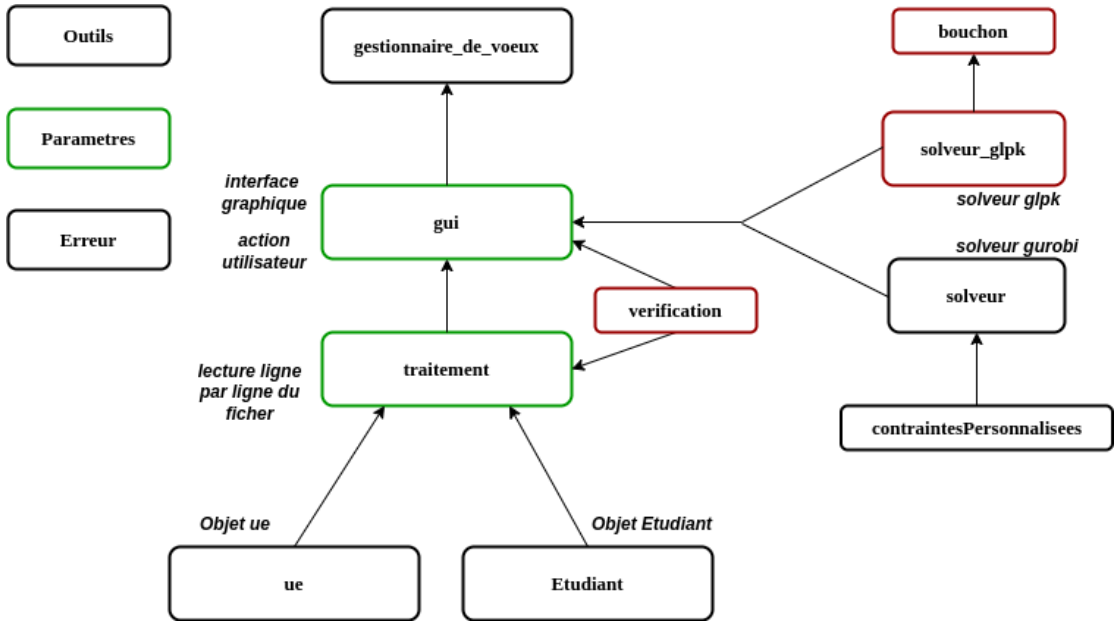

FIGURE 1 – Description simple des fichiers composant l'application avec en noir les fichiers inchangés, en vert les fichiers présent dans l'application de base et en rouge ceux qui ont été créés pour le projet (les flèches n'indique pas les import)

1.2 Modélisation du problème

On va également décrire ici le problème que le solveur de contrainte va résoudre. Comprendre ces équations permet de faciliter l'implémentation du nouveau solveur. On a donc besoin de décrire les variables à utiliser, la fonction objectif et les contraintes.

1.2.1 Problème

On a besoin de répartir les étudiants dans différentes unités d'enseignement (UE). Chaque étudiant va choisir des UE, classées en fonction de leur préférence. Chaque étudiant appartient à une spécialisation et donc les UE que chaque étudiant a et peut choisir peuvent appartenir à différentes spécialisations.

Chaque UE a un nombre de places limité. De plus, les horaires du (ou des) cours, des TDs et des TMEs d'une UE peuvent entrer en conflits avec ceux d'une autre UE.

On cherche donc à satisfaire au maximum le choix des etudiants en faisant attention au nombre de places disponibles et aux conflits d'emploi du temps.

1.2.2 Variables de décision

Les variables utilisées doivent permettre de décrire au mieux le problème. On utilisera deux variables qui seront binaires :

$$y_{ij} = \begin{cases} 1 & \text{si l'étudiant } i \text{ est inscrit dans l'UE } j \\ 0 & \text{sinon.} \end{cases} \tag{1}$$

$$x_{ijk} = \begin{cases} 1 & \text{si l'étudiant } i \text{ est inscrit dans l'horaire du cours/groupe } k \text{ de l'UE } j \\ 0 & \text{sinon.} \end{cases} \tag{2}$$

Notons que pour x_{ijk}, on va par la suite les séparer en plusieurs catégories où on aura le k qui désignera le cours ou le groupe de TD : $x_{ij(\text{cours})k}$ et $x_{ij(\text{groupe})k}$

1.2.3 Fonction objectif

On veut maximiser la satisfaction des étudiants lors des inscriptions dans chaque UE. Ainsi la fonction objectif pour les étudiant est :

$$\max \sum_i \sum_j n_j . y_{ij} \tag{3}$$

avec n_j le coefficient inversement proportionnel au choix ordonné de chaque étudiant, c'est-à-dire si un étudiant i préfère une UE a à une UE b alors $n_a > n_b$.

1.2.4 Contraintes

On dénombre 6 contraintes pour ce problème, on ajoute une nouvelle contrainte par rapport à l'application de base.

UE obligatoire : chaque étudiant doit être inscrit dans une UE obligatoire. On aura une contrainte pour chaque UE obligatoire de chaque étudiant.

$$y_{ij} = 1 \qquad \text{où } j \in \text{ UE obligatoire} \tag{4}$$

On peut aussi simplifier toutes ces équations en une seule et unique equation pour cette contrainte. On considère que n est le nombre d'équations (4) totale. Il s'agira de la seule contrainte qui peut être réduite en une unique ligne.

$$\sum_i \sum_j y_{ij} = n \qquad \text{où } j \in \text{ UE obligatoire} \tag{5}$$

Inscrit dans une UE et son groupe : un étudiant qui est inscrit dans une UE doit aussi être inscrit dans un seul des groupes de l'UE. Il ne peut pas être inscrit dans plusieurs groupe d'une même UE.

$$\forall i, \forall j,\ y_{ij} = \sum_{k} x_{ij(groupe)k} \tag{6}$$

Inscrit dans une UE et ses cours : Chaque UE peut avoir un ou plusieurs horaires de cours obligatoires. Si l'étudiant i est inscrit dans une UE j alors il doit suivre tous les cours k de l'UE :

$$\forall i, \forall j, \forall k,\ y_{ij} = \sum_{k} x_{ij(cours)k} \tag{7}$$

Les groupes ont une capacité maximale : chaque groupe ne peut contenir qu'un nombre limité d'étudiants. On doit donc sommer tous les étudiants inscrit dans l'UE. On note c_{jk} la capacité du groupe k de l'UE j :

$$\sum_{i} x_{ijk} \leq c_{jk} \tag{8}$$

Le nombre d'UE attribués à l'étudiant : chaque étudiant i doit suivre un nombre précis d'UE qu'on a noté nb_i.

$$\sum_{j} y_{ij} = nb_i \tag{9}$$

Si cette contrainte est relaché, il peut suivre nb_i UE ou moins :

$$\sum_{j} y_{ij} \leq nb_i \tag{10}$$

Horaire partagé : à chaque cours est attribué un horaire et à chaque groupe est attribué un ou deux horaires pour le TD et le TME s'il y en a un. On doit faire en sorte que si l'étudiant est inscrit à la fois dans a et dans b, l'horaire $h(x_{ij_ak_a})$ ne soit pas le même que $h(x_{ij_bk_b})$.

$$\sum_{(j'k')} x_{ij'k'} \leq 1 \quad \text{où } \forall (j'k'),\ h(x_{ij'k'}) = \text{ horaire } \alpha \tag{11}$$

1.2.5 Contraintes supplémentaires

Parmi les contraintes présentes dans l'application originale, il y en a une qui ne semblent pas nécessaire. En effet, elle est déjà présente implicitement dans une autre équation. On peut également avoir des contraintes qui sont trop spécifique pour être décrites ici.

Inscription dans un seul groupe par UE : On ne peut pas être inscrit dans plusieurs groupes d'une même UE. Cette contrainte est déjà présente dans l'équation (6) car y_{ij} ne peut être égal qu'à 0 ou 1.

$$\sum_{k} x_{ij(\text{groupe})k} \leq 1 \tag{12}$$

Contraintes personnalisées : sur la figure 1, il y a un fichier `contraintesPersonnalisees` qui contient,par exemple, des contraintes spécifiques au semestre 2 de M1 ou bien au semestre 1 de M2. Celles-ci n'ont pas été codés pour le moment dans le solveur GLPK.

2 Corrections de bugs et nouvelles fonctionnalités

Les premières modifications apportées à l'application de bases consistaient tout d'abord à corriger les bugs présents dans celle-ci. Ensuite, une amélioration de l'interface afin de rendre les actions de l'utilisateur plus claire pour celui-ci. On va détailler ci-dessous quels ont été ces modifications, leur intérêts et comment elles ont été implémentées.

2.1 Modifications apportées

L'objectif principal a été d'implémenter de nouvelles fonctionnalités et corrections de bugs en essayant, au minimum, de modifier ce qui était déjà présent de base.
Dans ce but, un maximum de fichiers ont été gardés intact et de nouveaux fichiers ont été créés au fut et à mesure des besoins décrit ci-dessous.

La figure 1 liste les programmes qui ont été modifiés [1]. Les parties du codes qui ont été modifiées vont être décrites par la suite. Le seul fichier dont on ne va pas parler est `parametres.py` car cela n'a pas d'intérêt.

Mise à jour de l'affichage lors du calcul : Lorsque l'utilisateur voulait faire un second calcul, l'affichage ne se mettait pas à jour et il fallait fermer et rouvrir l'application afin d'effectuer ce calcul. Il faut donc trouver un moyen d'effacer l'affichage. De ce fait, on en a aussi profité pour rajouter une action à l'utilisateur pour effacer l'affichage sans avoir besoin de lancer un nouveau calcul.

Vérifications préliminaires des fichiers : Lors du lancement de calcul, des vérifications ne sont pas faites sur les fichiers de voeux des étudiants et d'emploi du temps afin d'éviter que des erreurs ne s'enclenchent lors du calcul. Entre-autre, il y a :

- chercher les cases vides qui ne devrait pas l'être comme par exemple un numéro étudiant manquant
- regarder la cohérence entre le nombre de groupe, le nombre de capacités indiquées et le nombre d'horaires attribués à chaque groupe pour chaque UE
- compare les UEs indiquées pour chaque étudiant et celles se trouvant dans l'emploi du temps

Cela se fait en trois temps : on vérifie les deux fichiers séparément puis on le fait avec les deux ensemble.

Changement automatique d'onglets : lorsqu'on lance le calcul, celui-ci affiche les résultats de la répartition des UE pour chaque étudiant. Ensuite si on demande d'afficher le remplissage des groupes, on reste sur l'onglet précédent et ne bascule pas sur l'onglet affichant ce résultat.

1. En vérité, tous les programmes ont été retouchés, même ceux non indiqués comme tel sur la figure, pour au minimum les aérer et les commenter plus clairement

Indiquer à l'utilisateur les fichiers chargés : dans l'application de base, il n'y a aucun moyen permettant à l'utilisateur de savoir quel fichier va être chargé lors du calcul. Une solution simple était possible où on avait simplement à afficher ces fichiers dans le terminal. A la place, on est allé plus loin et la création d'un nouvel onglet affichant les informations nécessaire a été la solution envisagée.

2.2 Vérification des erreurs dans les fichiers

Lors de l'insertion des fichiers, le programme ne vérifiait pas si les fichiers avaient des coquilles. Il n'y avait pas de comparaisons faites entre le fichier des voeux et celui de l'emploi du temps pour vérifier qu'il y a une cohérence entre eux.

Il fallait dans un premier temps repérer où réaliser cette modification, c'est-à-dire le moment où la lecture du fichier se fait : dans `gui.py`, la méthode `calculer` connait les deux variables pour le chemin vers les deux fichiers et les donne à deux méthodes `recuperer_ue` et `recuperer_etudiants`, une pour chaque fichier.

```
# original
def calculer(self):
    try :
        self.dictionnaire_ue = recuperer_ue (Parametres.chemin_edt)
        self.liste_etudiants = recuperer_etudiants(Parametres.chemin_voeux)

        ...
```

On souhaite pouvoir afficher de possibles erreurs dans ces fichiers, ces deux fonctions devront donner en retour une variable permettant de savoir s'il y a eu une erreur. En plus de cela, il faudra comparer les deux fichiers pour savoir si ceux-ci sont cohérents : on utilise le résultat des deux méthodes `dictionnaire_ue` et `liste_etudiants` et on crée une nouvelle méthode qui prend ces deux variables en paramètres et retourne également une variable signalant les erreurs. On arrête l'appel de la fonction `calculer` si c'est le cas.

```
# modification
def calculer(self):
    try:
        self.succes_calcul = True
        self.err_edt = ""
        self.err_voeux = ""
        self.err_coh = ""


        self.dictionnaire_ue,erreurs_ue = recuperer_ue(Parametres.chemin_edt)
        if len(erreurs_ue) > 0 :
            self.err_edt = erreurs_ue
            self.succes_calcul = False
```

```
        self.liste_etudiants,erreurs_voeux = recuperer_etudiants(Parametres.
    chemin_voeux)
        if len(erreurs_voeux) > 0 :
            self.err_voeux = erreurs_voeux
            self.succes_calcul = False
        # arret du calcul car peut induire des erreurs par la suite
        if not self.succes_calcul :
               return

         erreurs_coherence = verification_coherence(self.dictionnaire_ue,self.
    liste_etudiants)
           if len(erreurs_coherence) > 0 :
            self.err_coh = erreurs_coherence
            self.succes_calcul = False
            return
```

Les trois variables `err_voeux`, `err_edt`, `err_coh` sont des listes contenant des tuples. L'initialisation laisse penser qu'il s'agit d'un chaîne de caractère mais cela ne pose pas de problème comme on peut utiliser la méthode `len` sur les trois qui renverra 0 si elles sont vides quoi qu'il arrive. L'affichage de ce que contient ces trois variables est géré plus loin dans le code mais on en parlera quand on décrira la partie 2.3 sur Tkinter et la modification de l'affichage.

Il doit donc modifier de la même manière les deux méthodes `recuperer_..` qui se trouvent dans le fichier `traitement.py`. On souhaite modifier le moins possible celui-ci : les deux fichiers de données sont ouvert au début, c'est ici qu'on va vérifier toutes les coquilles possibles avec une nouvelle fonction.

```
# meme modification pour recuperer_ue
def recuperer_etudiants(chemin_fichier):
    # ligne originale
    # lignes, fichier = dictionnaire_csv(chemin_fichier)

    # modification
    res = []
    lignes,fichier,erreur,liste_erreurs = verification_voeux(chemin_fichier)

     # reste du code avec simples ajouts
     try:
        if not erreur :
      liste_erreurs = []  # ajout pour une liste vide
      for ligne in lignes:
                res.append(Etudiant(ligne))
    except (Exception) as e:
        raise e
    finally:
```

```
        fichier.close()
    return res,liste_erreurs     # retourne la liste en plus
```

Enfin, on crée un nouveau fichier `verification.py` qui contient nos trois fonctions principales :

- `verification_voeux` va appeler la fonction `dictionnaire_csv` pour ouvrir le fichier et récupérer chaque ligne. Ensuite il va lire ligne par ligne, effectuer une analyse et renvoyer son rapport. Le rapport est stocké dans la liste s'il y a une erreur (pour chaque élément, on utilise un tuple avec le numéro de la ligne et le message décrivant le(s) erreur(s))
- `verification_ue` a exactement le même fonctionnement
- `verification_coherence` est différent car il prend en argument la liste d'étudiants et la hash map (ou dictionnaire) des UE. Les UE et etudiants sont des objets ce qui permet de comparer leurs données facilement. La fonction va simplement vérifier qu'il n'y a pas d'incohérence entre ces deux structures de données (nom d'une UE souhaité par l'étudiant qui n'existe pas par exemple).

Le reste des méthodes dans ce fichier sont considérés comme *private* et ne sont pas censés être utilisé en dehors du fichier. Par ailleurs, pour la 3ème méthode, on peut facilement rajouter d'autres comparaisons en fonction des données stockées si besoin.

2.3 Mise à jour de l'affichage

Pour effectuer cela, il a été nécessaire de se familiariser avec le module `Tkinter`. Le fichier `gui.py` est le seul fichier qui a été modifié mais celui-ci a subit une énorme refonte afin de faciliter les modifications apportés. De plus, avant de toucher quoi que ce soit dans le fichier, un test des nouvelles fonctionnalités a été réalisé dans un fichier [2] séparé afin de voir si celles-ci étaient possibles. Les morceaux de codes ci-dessous ont été simplifiés au maximum afin que, si on le souhaite, des ajouts et/ou modifications soient faciles.

2.3.1 Objectifs

Il y avait quatre objectifs :

- ajout d'un nouvel onglet permettant à l'utilisateur de visualiser les fichiers enregistrés pour le calcul, les actions des utilisateurs (une sorte d'historique) et les affichages de messages.
- ouverture d'une nouvelle fenêtre agissant comme un message d'alertes afin de signaler les erreurs dans les fichiers lorsque l'utilisateur veut effectuer un calcul.
- changement automatique d'onglets lorsque l'utilisateur fait une action. Par exemple, lancer *calculer* et avoir un calcul sans erreur doit mener automatiquement l'utilisateur vers la fenêtre affichant les UE obtenus par les étudiants.
- le lancement d'un nouveau calcul doit effacer les résultats affichés précédemment. On en a profité pour rajouter une nouvel option dans le menu permettant d'effacer ces résultats sans avoir à lancer un nouveau calcul.

2.3.2 Refonte des actions, du menu et ajout d'un nouvel onglet

L'affichage visuel de l'application de départ se décompose de deux onglets, d'une barre de menu en contenant trois et chaque menu est composé d'une série de bouton. ci-dessous est une version

2. Ce fichier `test_tkinter/test.py` n'a pas d'utilisations autre que des tests et de ce fait, il est truffé de bugs

simplifié de ce programme.

```
# fenetre de depart
self.fenetre = Tk() # dimension, titre, style, etc. ne sont pas detailles ici

# ajout d'une barre de menu
self.menubar = Menu(self.fenetre)

# ajout de deux menu dans la barre de menu
self.nouveau_menu = Menu(self.menubar, tearoff=0)
self.nouveau_menu2 = Menu(self.menubar, tearoff=0)

# ajout d'une cascade pour les boutons de chaque menu
self.menubar.add_cascade(label="menu1", menu=self.nouveau_menu1)
self.menubar.add_cascade(label="menu2", menu=self.nouveau_menu2)

# Creation d un notebook pour les onglets
self.notebook = ttk.Notebook(self.fenetre)
self.notebook.pack()

# ajout d'un onglet dans le notebook
self.nouvel_onglet1 = Frame(self.notebook)
self.nouvel_onglet2 = Frame(self.notebook)

# chargement des actions du nouveau menu a detailler
self.chargerNouveauMenu1()
self.chargerNouveauMenu2()

# remplissage de l'onglet de trace
self.afficher_onglet_log()

# configuration du menu dans la fenetre
self.fenetre.config(menu=self.menubar)
```

L'ajout d'un nouvel onglet est assez simple ici, cela se fait en une seule ligne. Le contenu de cet onglet est par contre beaucoup plus complexe et décrite dans la partie 2.3.4.

Il faut ensuite détailler ce que fait chaque menu. Ici, on a effectué beaucoup de modification car la gestion des actions n'était pas idéal.

```
# description de la fonction chargerNouveauMenu2 utilise precedemment
def chargerNouveauMenu2(self, resetHard=False):

        # on efface et reecrit le menu (avec un autre titre par exemple)
        if (resetHard):
```

```
            for i in range(2):
                self.nouveau_menu2.delete(0)
            self.menubar.entryconfig(2,label="Menu2")

        # ajoute une commande 1
        self.nouveau_menu2.add_command(label="commande1", command=self.commande1)

        # ajoute une commande 2 deactivee au debut
        self.nouveau_menu2.add_command(label="commande2", command=self.commande2,
    state=DISABLED)
```

La partie `command=self.commande1` appelle la méthode `commande1(self)` uniquement. Or, si on veut changer d'onglet après l'exécution de cette fonction, pourquoi ne pas rajouter une liste de méthodes à appeler dont `aller _vers_onglet(onglet)` qui permet de passer à un autre onglet. Il faudrait également créer cette méthode qui est très simple.
La seule subtilité à savoir est l'utilisation du `lambda` qui est nécessaire ici, notamment dû à la succession de commandes et à la nouvelle commande qui demande un argument.

```
def aller_vers_onglet(self,onglet):
    self.notebook.select(onglet)

def chargerNouveauMenu2(self, resetHard=False):
    ...
    # commande1 s'execute puis on se dirige vers l'onglet nouvel_onglet2
    self.nouveau_menu2.add_command(label="commande1", command=
        lambda:[self.commande1(),self.aller_vers_onglet(self.nouvel_onglet2)]
```

Mais une des commandes retravaillé précédemment dans la partie 2.2 doit pouvoir s'arrêter à toutes erreurs ou incohérences dans les fichiers. Si c'est le cas, on doit pouvoir afficher ces erreurs au lieu de changer d'onglet. On a besoin d'une conditionnelle et la liste de commande ne tient plus.
Ainsi, on décide de changer `add_command` des menus pour qu'elles n'appellent en fait qu'une seule fonction chacune qui sera une liste d'actions.

```
def chargerNouveauMenu2(self, resetHard=False):
    ...
    self.nouveau_menu2.add_command(label="commande1", command=lambda:self.
    commande1_menu2())
    self.nouveau_menu2.add_command(label="commande2", command=lambda:self.
    commande2_menu2(),state=DISABLED)

# change d'onglet a la reussite du calcul sinon ouvre la nouvelle fenetre pour
    afficher des erreurs
def commande1_menu2():
```

```
    self.calculer()
    if self.succes_calcul :
        self.aller_vers_onglet(self.nouvel_onglet2)
    else :
        self.afficher_liste_erreur_newWindows(self.err_edt, "nom_fichier")

# description plus generique
def commande2_menu2():
    self.action1()
    self.action2()
    if succes :
        self.action_success()
    else
        self.action_failure()
```

2.3.3 Ouverture d'une fenêtre pour l'affichage des erreurs

Les erreurs qui ont été trouvées, on souhaite les afficher dans une nouvelle fenêtre. Il faut savoir comment ouvrir cette nouvelle fenêtre et écrire dans celle-ci. On ajoute aussi un bouton fermant la fenêtre et nous renvoyant vers la fenêtre principale.

```
def afficher_liste_erreur_newWindows(self, messages,nom_fichier):

    nouvelleFenetre = Toplevel()
    ...   # dimension et titre

    # affichage dans la fenetre
    display = Label(nouvelleFenetre).pack()

    # on souhaite pouvoir avoir une barre de defilement a droite sur l'axe des Y
    scrollbarNew = Scrollbar(nouvelleFenetre,orient='vertical')
    scrollbarNew.pack(side=RIGHT,fill=Y)

    # l affichage est un texte avec une barre de defilement
    display = Text(nouvelleFenetre, yscrollcommand=scrollbarErr.set)

    # ajout du texte
    for nom in nom_fichier :
        display.insert(END,"nom"+nom+"\n")
    for (i,message) in messages:
        display.insert(END,"\n"+"ligne"+i+"\n"+message)

    # ajustement de la barre
```

```
    scrollbarNew.config(command = display.yview)
    display.pack()

    # bouton pour le retour avec commande de destruction
    boutonBack = Button(nouvelleFenetre, text= "Retour", command=lambda:
    nouvelleFenetre.destroy()).pack()
```

2.3.4 Ecrire dans un onglet et changer son contenu

Enfin, on doit pouvoir changer ou effacer le contenu des onglets :
- effacer les résultats des deux onglets pour l'affichage des résultats
- dans l'onglet de trace, l'utilisateur doit pouvoir savoir quel fichier sera chargé mais on va permettre aussi à l'utilisateur d'avoir un historique de ses actions et par la même occasion afficher de nouveau les erreurs des fichiers en plus de la nouvelle fenêtre (partie 2.3.3)

La fonction `afficher_onglet_log` permet de faire cela : on crée de boîte de texte dont une non modifiable qui affichera les fichiers et se mettra à jour à chaque fois que l'utilisateur choisit un nouveau fichier via l'appel de la fonction `insertion_texte_log`. La seconde boîte de texte, qui est modifiable, se mettra à jour à chaque action de l'utilisateur via l'appel d'une fonction `insertion_texte_fichier` .

Les deux méthodes `insertion_texte_...` peuvent être appelées n'importe où dans le fichier `gui.py` : beaucoup de méthodes de l'application originale ont été modifiées afin de simplement écrire dans la boîte de texte de l'onglet de trace.

Pour éviter d'avoir trop de texte dans la boîte, on ajoute également un bouton qui permet d'effacer son contenu.

Le code est similaire à celui décrit dans la partie 2.3.3 mais on va créer à la place deux `Text` qu'on va ajouter à un onglet plutôt qu'à une fenêtre et les positionner de sorte à ce qu'elles ne se chevauchent pas. Ces deux boîtes de texte doivent être nommées afin que les deux méthodes ci-dessus puissent savoir où écrire.

```
def insertion_texte_log(self,texte):
    # on ecrit apres la derniere ligne dans la boite de texte logAction
    self.logAction.insert("end",texte+"\n")
```

3 Passage à GLPK

Avec les correctifs testés et implémentés, on va pouvoir passer à une nouvelle étape : passer du solveur de contrainte gurobi à un autre solveur de contrainte. Le solveur choisi est pyGLPK qui est facile à implémenter en python et qui est gratuit, il faut simplement l'installer.

3.1 Fonctionnement de GLPK

La première étape est de comprendre la fonctionnement de GLPK et de voir ses différences avec Gurobi. On a dû créer de petits programmes qui permettent de résoudre des problèmes très simple puis regardé comment s'adapter pour s'occuper de problèmes beaucoup plus grands.
On se demande donc comment construire le modèle, le résoudre et récupérer les résultats.

Initialisation : on commence par créer la variable qui sera le modèle de manière similaire à Gurobi, on peut lui donner un nom.

```
import glpk
modele = glpk.LPX()              # on initialise le parametre modele
modele.name = "nom␣du␣modele"    # nomme le modele
```

Ajout de nouvelles variables : le modèle GLPK a un paramètre nommé `cols` qui est une liste initialement vide. Lorsqu'on ajoute une nouvelle variable, elle se retrouve en fin de liste. On peut préciser son nom `name`, son type `kind` et son domaine `bounds`.

```
modele.cols.add(2)               # ajout de deux variables en fin de liste

modele.cols[0].name = "x0"       # nomme la variable x0
modele.cols[1].name = "x1"       # nomme la variable x1

modele.cols[0].bounds = 0,10     # variable x0 comprise entre 0 et 10
modele.cols[1].bounds = 0,None   # variable x1 comprise entre 0 et infini

modele.cols[0].kind = int        # variable x0 est un entier
modele.cols[1].kind = bool       # variable x1 est binaire (valeur 0 ou 1)
```

Ajout de la fonction objectif : la fonction objectif se trouve dans le paramètre `obj` du modele. Ce paramètre est composé d'une liste où chaque élément de la liste correspond à une variable pour lesquelles on précise le facteur. On doit également préciser si la fonction objectif est maximale.

```
# on considere ici qu'il y a 4 variables x0 x1 x2 x3
# objectif : max ( x0 + 2.x1 + 3.x3 )
modele.obj.maximise = True       # fonction objectif MAX

# methode o1 : on precise tous les facteurs
modele.obj[:] = [1.0, 2.0, 0.0, 3.0]

# methode o2 : on rajoute les variables une par une, par default = 0
modele.obj[0] = 1        # x0
modele.obj[1] = 2        # x1
```

```
modele.obj[3] = 3       # x2
```

Ajout des contraintes : on a deux parametres qui sont `rows` une liste de contraintes où chaque index correspond à une contrainte et `matrix` qui s'occupe des coefficients. Pour chaque contrainte, on ajoute toujours une ligne à `rows`. Pour `matrix`, il y a différentes manières de la remplir :

- de manière directe en précisant tous les coefficients les uns après les autres. Dans ce cas, `matrix` fusionne toutes les contraintes en une seule ligne d'entiers (ou floats).
- `matrix` devient une liste de tuples composés de l'index de la contrainte, l'index de la variable et son coefficient.
- on utilise `rows` à laquelle on donne une `matrix` où on précise tous les coefficients les uns après les autres
- ou bien on utilise aussi `rows` et `matrix` qui est composé, cette fois, de tuples (index,coefficient)

On doit donner le domaine de chaque contrainte et on peut également, comme les variables, nommer chaque contrainte.

```
# on considere ici qu'il y a 4 variables x0 x1 x2 x3
# et 3 contraintes
# x0 + 2.x2 - 3.x3 = 5
# 0 <= x1 + 3.x3 <= 10
# -x0 + 2.x1 + x2 + x3 <= 8

modele.rows.add(3)      # ajout de 3 contraintes

modele.rows[0].name = "contrainte␣1"    # nomme la contrainte
modele.rows[1].name = "contrainte␣2"    # nomme la contrainte
modele.rows[2].name = "contrainte␣3"    # nomme la contrainte

modele.rows[0].bounds = 5,5     # domaine egal a 5
modele.rows[1].bounds = 0,10    # domaine entre 0 et 10 inclus
modele.rows[2].bounds = None,8  # domaine entre -infini et 8 inclus

# methode c1 : directe avec matrix
modele.matrix = [
1.0, 0.0, 2.0, -3.0,
0.0, 1.0, 0.0, 3.0,
-1.0, 2.0, 1.0, 1.0
]

# methode c2 : avec matrix et tuple (icontrainte,ivariable,coeff)
modele.matrix = [
(0,0,1.0),(0,2,2.0), (0,3,-3.0),
(1,2,1.0),(1,3,3.0),
(2,0,-1.0),(2,1,2.0),(2,2,1.0),(2,3,1.0)
]
```

```
# methode c3 : directe avec row et matrix
modele.rows[0].matrix = [1.0,0.0,2.0,-3.0]
modele.rows[1].matrix = [0.0,1.0,0.0,3.0 ]
modele.rows[2].matrix = [-1.0,2.0,1.0,1.0]

# methode c4 : avec row et matrix et tuple (ivariable,coeff)
modele.rows[0].matrix = [(0,1.0),(2,2.0),(3,-3.0)]
modele.rows[1].matrix = [(1,1.0),(3,3.0)]
modele.rows[2].matrix = [(0,-1.0),(1,2.0),(2,1.0),(3,1.0)]
```

Récupération des résultats : pour pouvoir obtenir le résultat d'une variable, il nous faut simplement l'index de la variable

```
modele.simplex()    # resolution par la methode du simplex
z = modele.obj.value         # z = valeur de la fonction objectif
x1 = modele.cols[1].primal   # x1 = valeur de la variable x1
```

Notes supplémentaires : tous n'a pas été détaillé ici simplement parce que cela n'a pa été utilisé par la suite. On peut par exemple supprimer des variables du modèle mais si on en supprime une, on doit faire attention à la supprimer dans les contraintes et fonction objectif. Par ailleurs, l'utilisation des tuples dans les listes est préférable car cela permet des modifications plus faciles à la volée de ces dernières.
Enfin, il est toujours bon de stocker quelque part le nombre de `cols` et de `rows` afin d'éviter de réécrire par dessus lors de nouveaux ajouts (si on se trompe d'index par exemple).

3.2 Implémentation du solveur pour notre problème

On s'est imposé une contrainte : le solveur Gurobi est confiné dans un seul fichier `solveur.py` où la fonction principale `resoudre` est appelé par la fonction `calculer` dans `gui.py`. Pour passer d'un solveur à un autre avec un simple changement dans les `import`, il a été décidé de faire la même chose : une fonction appelé `résoudre` dans le `solveur_glpk.py` prenant les mêmes arguments et renvoyant les résultats sous la même forme.

Afin de faire des tests plus facilement du nouveau solveur, on a créé un fichier faisant office de bouchon. Celui-ci fera un appel de `resoudre` auquel on a ajouté un argument "invisible" appelé `DEBUG` qui par défaut est égal à `False`.

3.2.1 Adapter le modèle à la taille du problème

On a vu dans le partie 3.1 qu'il y a plusieurs manière de construire le modèle. En considérant le nombre de variables à construire, c'est-à-dire y_{ij} et x_{ijk}, on en déduit qu'il y en aura beaucoup trop pour utiliser les méthodes directes (qui ne repose pas sur des tuples). On peut coder des fonctions

qui permette de facilement ajouter les variables une par une et les contraintes une par une.

Pour les variables, on crée la fonction `ajoute_variable_modele(modele,nom)` qui prend en paramètre le nom de la variable et le modèle. Cette fonction doit au moins renvoyer ce même modèle avec la nouvelle variable et l'index de cette variable dans le modèle afin de pouvoir récupérer son résultat par la suite (cf partie 3.2.3).

Pour les contraintes, en se basant sur les équations décrites dans la partie 1.2.4, on remarque que les coefficients peuvent être égaux à -1, 0 ou 1. On peut donc se faciliter la tâche en créant une fonction `ajoute_contrainte_modele(modele,p,n,min,max)` qui ajoute une nouvelle contrainte au modèle. Cette fonction prend en argument le modele, une liste `p` contenant les index des variables de coefficient positif, une liste `n` contenant les index des variables de coefficient négatif et le domaine `min` et `max` de la contrainte. la fonction doit au minimum renvoyer le modèle.

Pour la fonction objectif, on a besoin d'une liste ordonnancée des index des variables y_{ij} où les y_{ij} d'un étudiant i sont de l'UE la plus préférée à la moins préférée. Dans le code, on ajoute les variables une par une dans `obj` en précisant son coefficient et index.

3.2.2 Structures de données

Afin d'avoir un code efficace, on veut éviter d'avoir à parcourir des listes entières afin de récupérer une variable.
Par exemple, parmi toutes les variables qui vont se trouver dans le modèle, il y aura des x_{ijk} et des y_{ij}. Or, la fonction objectif n'est composée que de y_{ij} : pourquoi après résolution devront-nous lire la valeur de x_{iak} si on sait que $y_{ia} = 0$. De même, si $y_{ib} = 1$, alors on a besoin de chercher les valeurs x_{ibk} des cours et des groupes.
Si on connaît leur index dans le modèle, on peut retrouver leur valeur en une ligne sans avoir à parcourir entièrement `cols`. Ou alors, si on liste tous les variables associés à un étudiant dans une table de hachage, on peut savoir quel index consulter.

Ainsi, on va construire des structures de données qui vont faciliter cette recherche :

- `dico_xy` est une table de hachage classé par nom de variable où à chaque variable est associé toutes les informations qui peuvent sembler utile : l'index dans `cols`, l'étudiant, l'UE et son intitulé, le numéro de groupe/cours si besoin (nul si la variable est y)
- `dico_etu` est une table de hachage classé par étudiant. A chaque étudiant est associé une seconde table de hachage pour chaque UE (intitulé de l'UE). Enfin pour chaque UE, on a trois éléments : le nom de la variable, le type de la variable y, x cours ou x groupe et le numéro de cours/groupe

Ces deux tables combinées permettent d'avoir accès à toutes les informations dont on a besoin. De plus, `dico_etu`, est construit afin de récupérer efficacement les résultats des variables dont on a besoin.

3.2.3 Organisation du solveur

Definissons maintenant les étapes de la fonction `resoudre(dico_ue, liste_etudiants)`.

Algorithme 1 `resoudre` : solveur

Entrée: `hashmap{String:UE}` dico_ue ; `[etudiant]` liste_etudiant
Sortie: `String` resultat

```
init(modele) , init(listes_contraintes) , init(dico) , init(liste_variables_objectif)
Pour etudiant dans liste_etudiant Faire
    ynom ← nomme_variable(etudiant)
    index ← ajoute_variable_modele(ynom,modele)
    ajoute_dico_contrainte(index,ynom,dico,liste_contrainte)
    ajoute_liste_objectif(ynom,liste_variables_objectif)
    Pour ue dans etudiant.liste_ue Faire :
        Pour crs dans dico_ue[ue].liste_cours Faire
            xnom ← nomme_variable(crs)
            index ← ajoute_variable_modele(xnom,modele)
            ajoute_dico_contrainte(index,xnom,dico,liste_contrainte)
        Fin Pour
        Pour grp dans dico_ue[ue].liste_groupe Faire
            xnom ← nomme_variable(crs)
            index ← ajoute_variable_modele(xnom,modele)
            ajoute_dico_contrainte(index,xnom,dico,liste_contrainte)
        Fin Pour
    Fin Pour
Fin Pour
Pour contrainte dans liste_contrainte Faire
    ajoute_contrainte_modele(contrainte, modele)
Fin Pour
solve(modele)
dico_resultat ← traduire_resultat(dico,modele)
resultat ← texte(dico_resultat)
Retourne resultat
```

Les étapes de cet algorithme 1 doivent être les suivantes :

1. initialiser le modele, les dictionnaire `dico_xy` et `dico_etu` et les listes des contraintes
2. créer une variable, l'ajouter au modèle et conserver l'index
3. ajouter la variable et index aux dictionnaires et aux listes de contraintes/objectif (listes `p` et `n` pour chaque contrainte) dans lesquelles elle doit apparaître
4. répeter l'étape 2 jusqu'à ce que toutes les variables soient créées
5. ajouter les contraintes au modele
6. définir la fonction objectif et l'ajouter au modele
7. résoudre
8. extraire les résultats et les traduire sous forme de chaine de caractères

3.2.4 Débuggeur

A la différence du solveur Gurobi, la fonction `resoudre` a un paramètre supplémentaire "invisible" qui par défault est égal à `False`. Quand celui-ci est `True`, le programme va afficher sur le terminal[3] des représentations textuelles des dictionnaires, listes...

Cela permet de vérifier étape par étape que ce qui a été créé est en accord avec ce qu'on veut. Par exemple, on peut vérifier quelles sont les variables présentes dans chaque contrainte. Ou bien obtenir les résultats de chaque variable contrainte par contrainte : si l'équation (12) a 5 variables sur 10 qui sont égales à 1 et que la capacité $c_{jk} = 5$, on peut en déduire que la résolution a probablement fonctionné, surtout si toutes les autres contraintes qu'on peut voir affichées mènent à la même conclusion.

Cela s'est fait étape par étape et une conséquence fait qu'une partie des fonctions vont demander des arguments en plus pour pouvoir construire une chaîne de caractères qui sera affichée par le débuggeur.

Par ailleurs, on a créé un fichier `bouchon.py` pour jouer le rôle de bouchon afin de pouvoir afficher de manière plus claire les erreurs déclenchées spécifiquement par le solveur et éviter d'avoir à relancer l'interface graphique à chaque modification apporté au solveur.

Conclusion

Au moment de l'écriture de ce rapport, on a pu faire tout ce qui est décrit ici. On a pu constater que les résultats obtenu après avoir lancé la résolution du modèle semblent en accord avec les contraintes et objectif.

Il faudrait par la suite effectuer des comparaisons entre les résultats du GLPK et de Gurobi qui semblent avoir quelques différences. Cela peut être dû au fait que les solveurs ne classent pas les variables de la même manière (il y a probablement plusieurs solutions possibles) mais aussi, il ne faut pas oublier, qu'une contrainte, correspondant à l'équation (7), a été rajouté par rapport à l'application de base.

3. Si c'est trop volumineux, on peut automatiquement créer un fichier texte de ce rendu avec la commande "`python3 programme.py > texte.txt`"