RÉSOLUTION APPROCHÉE DU PROBLÈME D'ORDONNANCEMENT

Rapport de projet

Auteurs : Corentin COUDRAY
Christophe JULIEN
Khanh An Noël TRAN
Groupe : M2_9
Date : 03/04/2014

ESME sudria Paris

38 rue Molière
94200 IVRY sur SEINE
Tel. : 01 56 20 62 00
Fax. : 01 56 20 62 52
http ://www.esme.fr

Sommaire

1 Introduction **3**

2 Contexte **4**

3 Revue de littérature **5**

4 Cahier des charges **6**
4.1 Mise en contexte du problème 6
4.2 Langage utilisé 6
4.3 Interface graphique 6
4.4 Les données d'entrée du problème 6
4.5 Les données de sortie du problème 6
4.6 Modélisation des données 7
4.6.1 Les professeurs 7
4.6.2 Les classes 7
4.6.3 Les cours 7
4.6.4 Les salles 7

5 Représentation formelle **8**

6 Les acteurs **9**
6.1 Premier acteur : Saisie des données 9
6.2 Second acteur : Maintenance de l'emploi du temps 9

7 Cas d'utilisation **10**
7.1 Génération de l'emploi du temps 10
7.1.1 Objectif 10
7.1.2 Acteurs 10
7.1.3 Données échangées et description des enchaînements 11
7.2 Maintenance de l'emploi du temps 12
7.2.1 Objectif 12
7.2.2 Acteurs 12
7.2.3 Données échangées 12
7.2.4 Description des enchaînements 12
7.2.4.1 Pré-condition 12
7.2.4.2 Séquence 13

8 Organisation des données **14**
8.1 Les professeurs 14
8.2 Les classes 14
8.3 Les matières 14
8.4 Les salles 14
8.5 Les semaines 15
8.6 Les fichiers externes 15

8.7 Diagrammes de classes 15

9 Simplification du problème **18**

10 Pré-traitements **19**
10.1 Le nombre de professeurs 19
10.2 Le nombre de cours total sur le semestre 20

11 Réalisation de l'emploi du temps **22**
11.1 Répartition du programme sur le semestre 23
11.2 Répartion des cours sur leurs créneaux 26
11.2.1 Placement de cours déjà fixé la semaine précédente 27
11.2.2 Planification des nouveaux cours du semestre 29

12 Déplacements des cours **32**
12.1 Déplacement d'un cours existant 32
12.2 Ajout d'un cours non placé 33

13 Discussions **34**

14 Conclusion **35**

Table des figures **36**

Références **37**

1 Introduction

La planification sous contraintes est la discipline des mathématiques appliquées consistant à ordonner diverses séquences ou évènements dans un espace-temps limité. Ces planifications impliquent un grand nombre de contraintes, qu'elles soient liées aux disponibilités de personnes, d'emplacements ou à tout autre type de problème.

La réalisation d'une planification est longue et périlleuse. Il est très difficile de concilier l'ensemble des contraintes qui accompagnent cet ordonnancement et de les organiser de manière optimale. La mise en place d'un calendrier est souvent faite à la main ou de manière approximative avec d'autres applications.

L'objectif de ce projet est donc de réaliser une application, permettant de générer une planification tenant compte de toutes les contraintes que l'on aura indiquées au préalable.

2 Contexte

Ce sujet a été proposé dans le but d'alléger le travail de M. Maeso, responsable administratif des études. En effet, chaque année, il doit créer manuellement l'intégralité des emplois du temps de l'école, en tenant compte de toutes les contraintes de disponibilités des professeurs, des salles, des élèves et des matières. L'application pourra lui faire gagner un temps non-négligeable dans son travail.

3 Revue de littérature

Aujourd'hui, il existe déjà des systèmes de planification informatique, mais ils correspondent toujours à des cas bien spécifiques : organisation du personnel médical dans un hôpital, organisation du personnel dans une base militaire, etc. Mais il semble impossible à l'heure actuelle d'imaginer un logiciel générique permettant de lister des contraintes et d'élaborer le planning correspondant et adapté quel que soit le milieu professionnel.

Ainsi, il nous faudra concevoir notre solution dans son ensemble. Nous pourrons tout de même nous inspirer des méthodes de résolution de ces plannings pour essayer de les adapter à notre problème.

En l'occurrence, nous avons trouvé quelques pistes de recherche :

— Algorithme des colonies de fourmis : cet algorithme est basé sur le comportement réel des fourmis, qui parviennent, par le biais de rétroactions, à trouver le chemin le plus court entre une source de nourriture et leur nid. [1]

— Recherche tabou : cet algorithme consiste à partir d'une position donnée, à explorer le voisinage pour choisir la position qui minimise la fonction "objectif". Le mot "tabou" vient du fait qu'une fois une position écartée, nous ne pouvons plus y revenir. Cet algorithme procède donc par élimination jusqu'à trouver une solution convenable. [2]

4 Cahier des charges

4.1 Mise en contexte du problème

Les algorithmes nécessaires à l'élaboration d'un tel projet relèvent des problèmes dit "NP-complets". Les problèmes NP-complets sont des problèmes qui sont au minimum aussi difficiles que les problèmes NP. Ceux-ci sont des problèmes de décision à temps polynomial.

Cela signifie qu'il n'existe pas de solution exacte, et qu'on ne peut les résoudre avec les méthodes de résolution classiques.

Même s'il n'existe pas de solution déterministe, nous pouvons malgré tout arriver à une solution "acceptable", c'est à dire répondant à l'ensemble des exigences du problème sans pour autant être optimale dans son organisation, en mettant en oeuvre des algorithmes de résolution rétroactifs.

Le temps de résolution de ces problèmes augmente de manière exponentielle par rapport à l'augmentation des paramètres d'entrée. Nous devrons toujours nous contenter d'une solution qui sera au mieux "satisfaisante". [3]

4.2 Langage utilisé

Nous avons décidé de programmer ce projet en C++ car c'est un langage objet qui permet de bien organiser son code, notamment grâce au système de classes, au polymorphisme, et l'héritage, etc. De plus, il s'agit d'un langage que nous avons beaucoup pratiqué durant notre cursus. Le Java répond également à ces critères, mais il est moins rapide, et un langage rapide est primordial pour une résolution raisonnable en temps de notre problème.

4.3 Interface graphique

Pour concentrer nos efforts sur la résolution du problème, nous avons également décidé de ne pas inclure l'interface graphique à notre projet, pour des raisons de temps. Cependant, nous pourrons mettre en place une petite interface utilisateur pour le programme qui gèrera les données.

4.4 Les données d'entrée du problème

Les données d'entrée du problème sont stockées dans des fichiers externes. En effet, la nature et la quantité des données ne nécessitent pas de base de données. Cette méthode rend facile la vésication de l'exactitude des données saisies lors de l'exécution du programme. Ainsi il n'est pas nécessaire de relancer le programme lorsque nous voulons modifier des données.

Ce système permet également de mettre en place un archivage des données : l'utilisateur n'aura qu'à copier-coller les fichiers dans un dossier séparé.

4.5 Les données de sortie du problème

En sortie, chaque classe, professeur et salle auront leur emploi du temps, stockés dans des fichiers externes.

4.6 Modélisation des données

4.6.1 Les professeurs

Un professeur sera défini par :
— son nom
— ses disponibilités
— la liste des cours qu'il peut enseigner

4.6.2 Les classes

Une classe sera définie par :
— son nom
— le nombre d'élèves
— la liste des cours au programme

4.6.3 Les cours

Un cours sera défini par :
— son nom
— le nombre d'heures de cours
— le type de salle dans laquelle il doit être donné (un type de cours)

4.6.4 Les salles

Une salle sera définie par :
— son nom
— le nombre de places
— le type : salle de cours, salle de TP

5 Représentation formelle

Dans l'ensemble de notre travail, un certain nombre de concepts mathématiques apparaîtront de manière récurrente. Cette section rappelle de façon formelle ces principes.

Soit $\mathbb{E}$ un ensemble et $\mathcal{R}$ une relation binaire sur les éléments de $\mathbb{E}$.

Définition 1 Ordre strict : $\mathbb{E}$ *muni de* $\mathcal{R}$ *est une structure d'ordre strict si et seulement si* $\mathcal{R}$ *est antisymétrique, non réflexive et transitive.*

Définition 2 Ordre partiel : $\mathbb{E}$ *muni de* $\mathcal{R}$ *est une structure d'ordre partiel quand une partie seulement des éléments de* $\mathbb{E}$ *sont soumis à une une relation d'ordre strict.*

Définition 3 Préordre : $\mathbb{E}$ *muni de* $\mathcal{R}$ *est une structure de pré-ordre si et seulement si* $\mathcal{R}$ *est une relation binaire réflexive et transitive.*

$(\mathcal{X}, \preceq)$ ordre partiel.
$(\bigcup \mathcal{X}, \prec)$ ordre strict.

Le problème de l'organisation l'emploi du temps est la donnée de :

1. Un ensemble de matières (une promotion) $\mathcal{X} := \{X_1, \ldots, X_n\}$ muni d'un ordre partiel $\mathcal{O}$.
2. Un ensemble de blocs ordonnés (ordre strict) W correspondant à un ensemble de semaines de cours. Nous sommes convenus que $|W| := k22$. [1]
3. Un élément $X_{i_{\overline{i\in 1,n}}} \in \mathcal{X}$ est un ensemble de cours muni d'un ordre strict sur des blocs de quatre heures quelle que soit la matière .i.e. quels que soient i, i', deux indices de matière, et j, j' deux indices de cours, nous avons $x_{i,j} \prec x_{i',j'}$ ou $x_{i,j} \succ x_{i',j'}$

Tâche : Déterminer un ordre des $x_{i,j}$ (i correspond à un indice de matière, j un indice de cours de la matière X_i) tel que les propriétés suivantes soient respectées :

— Tous les cours sont placés au sein des $k22$ blocs.
— l'ordre partiel sur les matières est respecté
— l'ordre partiel sur les cours est respecté
— un cours ne peut avoir lieu plus d'une fois une même semaine quelle que soit la matière.
— un cours occupe un même bloc sur l'ensemble des $k22$ blocs.

Nous introduisons un niveau de conflit compris entre 1 et n. Il représente le nombre de professeurs disponibles pour enseigner une matière donnée [2].

1. Considérant qu'un bloc est composé de quatre heures consécutives et considérant qu'une semaine est une juxtaposition de 22 blocs.
2. ce niveau de conflit peut vraisemblablement concerner les salles aussi

6 Les acteurs

Il existe deux principaux types d'acteur pour notre logiciel :

6.1 Premier acteur : Saisie des données

Le rôle du premier acteur est de saisir toutes les données relatives aux professeurs, salles, et élèves, et tous les paramètres nécessaires à la génération de l'emploi du temps.

6.2 Second acteur : Maintenance de l'emploi du temps

Etablir un emploi du temps est une bonne chose, mais il faut également pouvoir le modifier au cours de l'année si des évènements imprévus doivent être rajoutés. Le rôle du deuxième acteur est donc de gérer la maintenance de l'emploi du temps au fur et à mesure que l'année avance. Il doit pouvoir rajouter des évènements dans l'emploi du temps des professeurs et/ou des élèves.

7 Cas d'utilisation

7.1 Génération de l'emploi du temps

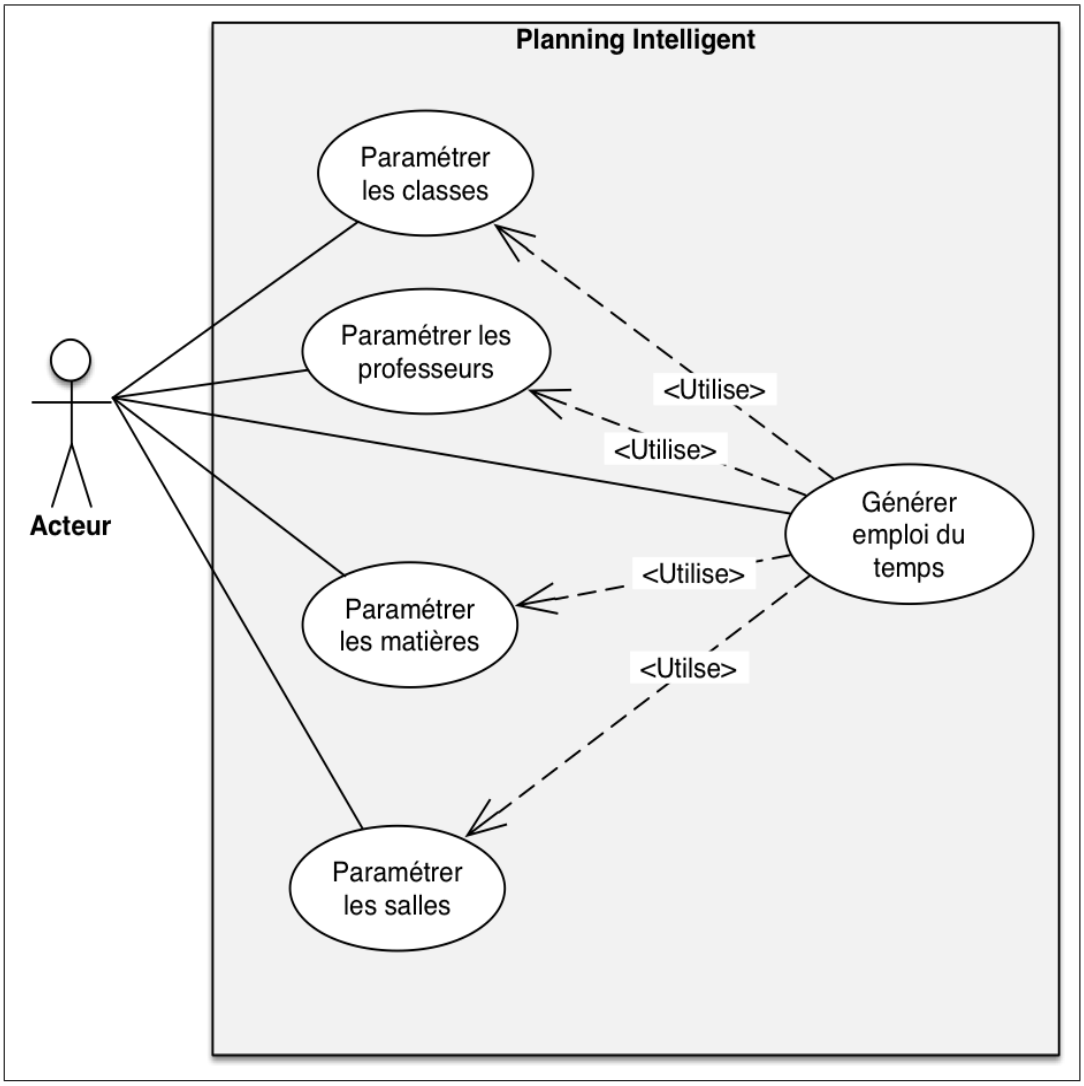

FIGURE 1 – Diagramme de cas d'utilisation, génération de l'emploi du temps

7.1.1 Objectif

L'objectif est de créer un emploi du temps à partir de données brutes entrées par l'utilisateur.

7.1.2 Acteur

L'acteur est celui qui est chargé de la saisie des données.

7.1.3 Données échangées et description des enchaînements

L'acteur en question entre les différentes données propres aux professeurs, classes, salles, et matières. Pour chaque élément, il devra remplir des critères bien spécifiques :

- Pour chaque professeur, il doit entrer son nom, les matières que celui-ci enseigne et ses disponibilités dans la semaine.
- Pour chaque matière, il doit entrer le nom de la matière, le programme qui devra être dispensé au cours de l'année (matières et nombre d'heures).
- Pour chaque classe, il doit préciser le nom de la classe, la promotion à laquelle elle appartient et le nombre d'élèves qu'elle contient.
- Pour chaque salle, il doit entrer leur contenance et leur type (salle de TP informatique, salle de cours, etc.)

Une fois que toutes les données ont été entrées, le logiciel génère tous les emplois du temps de l'école.

7.2 Maintenance de l'emploi du temps

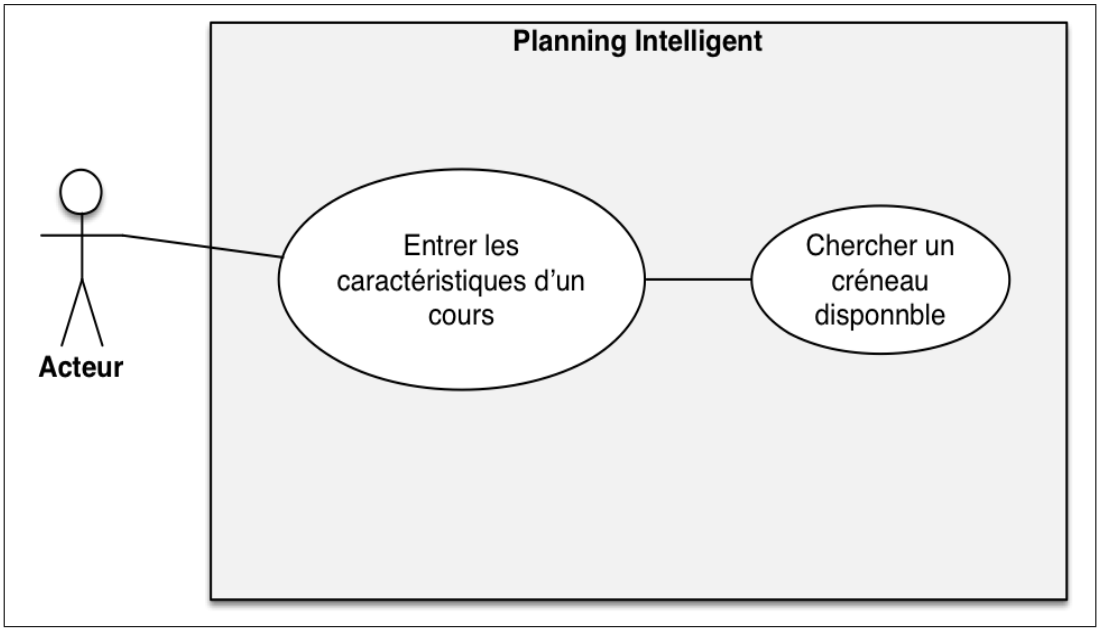

FIGURE 2 – Diagramme de cas d'utilisation, maintenance de l'emploi du temps

7.2.1 Objectif

Pour une raison ou pour une autre, l'utilisateur peut être amené à rajouter un évènement dans l'emploi du temps. Le logiciel doit donc le permettre en tenant compte des évènements déjà placés et des contraintes que cela implique, sans avoir à générer à nouveau la totalité de l'emploi du temps.

7.2.2 Acteur

Le seul acteur à intervenir dans ce cas est celui chargé de la maintenance.

7.2.3 Données échangées

Pour placer un cours dans l'emploi du temps, le programme consulte les disponibilités de chaque élément :

- Les disponibilités des professeurs ou des intervenants
- Les disponibilités des élèves
- Les disponibilités d'une salle convenant à l'évènement, et dont la capacité est suffisante pour accueillir tous les élèves.

7.2.4 Description des enchaînements

7.2.4.1 Pré-condition

Lorsque l'utilisateur désire rajouter un évènement, il lui faut connaître les disponibilités de chaque entité, et savoir quelles classes sont concernées par l'évènement, si elles doivent

recevoir l'évènement par classe ou par promotion, etc.

7.2.4.2 Séquence

L'utilisateur entre les disponibilités de chacun dans le programme, et celui-ci cherche les différents créneaux possibles pour placer le cours. L'utilisateur a alors deux possibilités : soit il laisse le programme placer automatiquement le cours, soit il demande au programme d'afficher la liste des créneaux disponibles afin de placer manuellement le cours. Celui-ci n'a alors qu'à choisir le créneau qui lui semble le plus adapté.

8 Organisation des données

Nous avons défini la semaine sur 22 créneaux de 2 heures chacun. Avec 2 créneaux le matin et 2 l'après-midi du lundi au samedi matin.

8.1 Les professeurs

Les disponibilités du professeur sont représentées par un mot binaire de 22 bits, un bit par créneau. 1 représentant une disponibilité et 0 une indisponibilité. Le mot binaire d'un professeur correspond à une semaine : chaque nouvelle semaine aura un nouveau mot binaire.

Lorsque nous ajoutons un cours à un professeur, ses disponibilités sur la semaine concernée changent, et il faut donc modifier le mot binaire en conséquence.

8.2 Les classes

Une classe comprend une liste de semaines, représentant le semestre ou l'année. Celles-ci possèdent un numéro (ID) définissant leur ordre d'arrivée dans l'année, et 22 créneaux assignés que l'on doit remplir avec les cours planifiés à la classe.

Chaque classe possède également un nombre d'élèves déterminant la taille minimum de la salle dans laquelle se déroulera le cours.

Chaque classe a une liste de cours qu'elle devra obligatoirement suivre.

8.3 Les matières

Une matière est définie par son nom, la promotion à laquelle elle est rattachée, et le nombre d'heures à enseigner. Dans le cas d'une matière commune à plusieurs promotions, nous les distinguons par des noms différents : un pour chacune des promotions (ex : Algèbre B1, Algèbre B2).

De la même manière, nous distinguons les cours en classe entière, des cours en demi-groupes et des TPs. Cela implique un dernier paramètre qui détermine le type de salle dans laquelle la matière doit être enseignée (ex : un TP ne peut pas être fait dans un amphithéâtre).

8.4 Les salles

Une salle est définie par son nom, sa capacité (nombre de places), et son type. La capacité est un paramètre déterminant dans le choix de la salle à attribuer lors du placement d'un cours dans l'emploi du temps.

De même, le type de la salle va de pair avec le type d'une matière explicité précédemment. Ces deux paramètres doivent concorder pour que la salle puisse être attribuée à un cours.

Enfin, il faut également préciser dans quel bâtiment se trouve la salle : dans les locaux d'Ivry-sur-Seine, ou dans ceux de Montparnasse, car comme expliqué précédemment, les professeurs ne peuvent pas se permettre de donner de cours dans les deux locaux lors d'une même demi-journée.

8.5 Les semaines

Chaque entité de notre emploi du temps (classes, professeurs, salles) doit être organisée dans le temps. Il faut donc leur attribuer un paramètre 'semaine', correspondant à leur planning d'occupation. Ces semaines sont mises à jour à chaque fois qu'un nouveau cours sera ajouté au planning.

8.6 Les fichiers externes

Les données d'entrée à stocker dans les fichiers sont toutes celles que l'utilisateur devra saisir pour lancer le programme. C'est à dire les données sur les professeurs (nom, disponibilités, matières), sur les matières (nom, nombre d'heures, promotions associées) et les salles (nom, capacité, type, bâtiment).

Chaque type de donnée est stockée dans un fichier séparé. Au sein du fichier, une ligne contient tous les paramètres d'un seul élément. Les données seront donc récupérées ligne par ligne à chaque exécution du programme.

Les données en sortie du problème, c'est à dire les emplois du temps de chaque éléments (classes, professeurs, salles) seront également dans des fichiers externes.

8.7 Diagrammes de classes

Ce diagramme représente les liens entre les différentes classes de notre programme avec les simplifications que nous avons décidé de réaliser dans un premier temps.

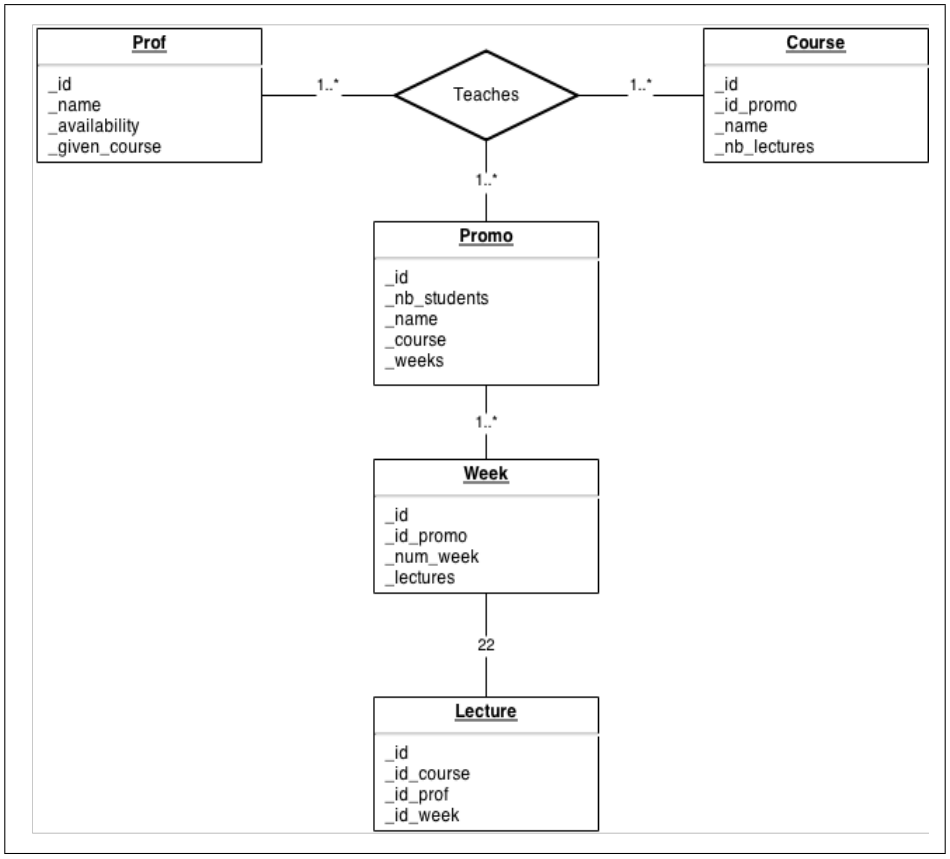

FIGURE 3 – Diagramme de classes de mi-projet

Le bloc "Teaches" représente les relations de *n* à *n* entre

- les professeurs("Prof") et les matières("Course") :
  - un professeur peut enseigner plusieurs matières
  - une matière peut être enseignée par plusieurs professeurs
- les promotions ("Promo") et les professeurs :
  - une promotion a plusieurs professeurs
  - un professeur enseigne à plusieurs promotions
- les matières et les promotions :
  - une promotion a plusieurs matières
  - une matière est enseignée à plusieurs promotions

Ce second diagramme est une ébauche du diagramme des classes final sous réserve de modifications lorsque nous ajouterons les salles à notre problème.

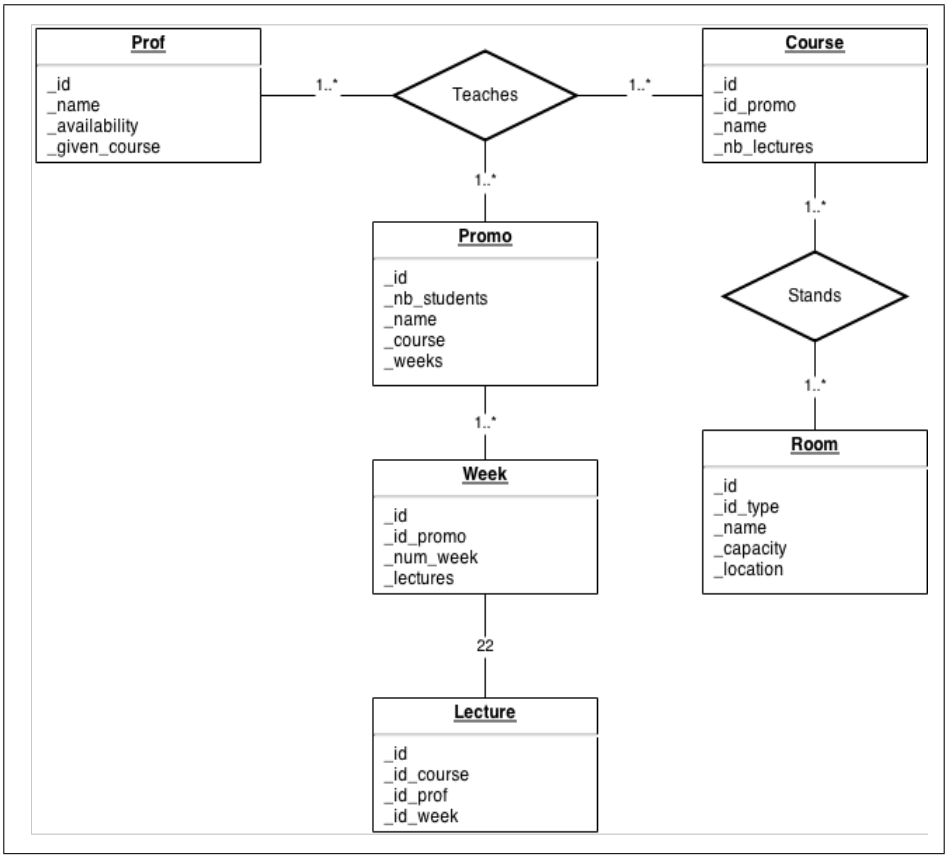

FIGURE 4 – Diagramme de classes final envisagé

Le bloc "Stands" représente les relations de n à n entre les matières ("Course") et les salles de cours ("Room") :

— une matière peut être enseignée dans plusieurs salles

— une salle peut recevoir plusieurs matières

Chaque salle va avoir un type spécifique correspondant aux différents types de cours (cours classe entière, TP, cours de demi classe).

9 Simplification du problème

Afin de simplifier le problème, nous avons décidé de le résoudre sans tenir compte des contraintes imposées par les classes de M2. En effet, contrairement aux autres promotions, la M2 est composée de 4 classes ayant des programmes différents. Chacune de ces classes étant elle-même séparée en plusieurs spécialités ayant un tronc commun et un programme spécialisé. Par conséquent nous avons décidé de commencer la résolution du problème en se focalisant sur les promotions ayant un programme unique.

De plus, nous avons décidé de nous affranchir des contraintes relatives aux salles, notamment en ce qui concerne la distinction entre les locaux de Montparnasse et d'Ivry-sur-Seine.

10 Pré-traitements

Avant de réaliser l'emploi du temps, nous procédons à des vérifications sur les données d'entrées afin de détecter toutes les incohérences. Ainsi, nous éliminons au préalable une partie des traitements qui n'aboutiront pas.

10.1 Le nombre de professeurs

La première vérification concerne le nombre de professeurs en entrée. Nous vérifions qu'il y a assez de professeurs pour dispenser les cours de chaque classe. Ainsi pour un cours donné, l'algorithme somme les disponibilités des professeurs puis compare le résultat au nombre de classe, en tenant compte du fait que pour des cours de 4h, le nombre de disponibilités nécessaire est doublé.

Soit n le nombre de professeurs pouvant donner un cours c et m le nombre de classes devant suivre ce cours. Pour un cours de 2 heures, nous avons :

$$\sum_{i=0}^{n} dispo_{prof_i} > m$$

Pour un cours de 4 heures, nous avons :

$$\frac{\sum_{i=0}^{n} dispo_{prof_i}}{2} > m$$

Pour un cours dispensé sur 4h, le nombre de disponibilités nécessaire est doublé. Pour simplifier nos calculs, nous divisons par 2 le nombre de disponibilités trouvées pour un professeur pour le comparer aux disponibilités nécessaires pour dispenser le cours.

L'opération est répétée pour l'ensemble des cours.

Algorithme 1 : Pré-traitement du nombre de professeurs

```
for all Cours do
    idCours ← identifiant de Cours
    idPromo ← identifiant de la promotion recevant Cours
    nbClasses ← nombre de classe de la promotion idPromo
    for all Profs do
        if Profs donne le cours idCours then
            for all CreneauxProf do
                if Profs est disponible then
                    nbCreneaux ← nbCreneaux + 1
                end if
            end for
        end if
    end for
    if Cours est sur 4h then
        nbCreneaux ← nbCreneaux/2
    end if
    if nbClasses > nbCreneaux then
        display (Erreur sur le nombre de professeurs pour la promotion idPromo)
        EXIT FAILURE
    end if
end for
display (Nombre de professeurs ok)
```

10.2 Le nombre de cours total sur le semestre

La seconde vérification porte sur le nombre d'heures de cours à dispenser à une classe. Ce nombre ne doit pas excéder la totalité des heures du semestre. Le programme somme l'ensemble des cours que possède une classe et le compare au nombre d'heures total du semestre.

$$\sum_{i=0}^{n} nbHeures_{cours_i} \leq s * c * h$$

Avec :
— n le nombre de cours d'une classe
— s le nombre de semaines du semestre
— c le nombre de créneaux sur une semaine
— h le nombre d'heures d'un créneau

Algorithme 2 : Pré-traitement du nombre d'heures sur le semestre

```
for all Classes do
    listCours ← ensemble des cours que suit une classe
    for all cours in listCours do
        nbHours ← nbHours + nombre d'heures du cours cours
    end for
    if nbHours > (nombre de semaines du semestre * nombre de créneaux par semaine *
    nombre d'heures par créneau) then
        display(Erreur, trop d'heures pour la classe Classes)
        EXIT FAILURE
    end if
end for
display(Nombre d'heures de cours ok)
```

Une fois ces pré-traitements réalisés, nous pouvons commencer la conception de l'emploi du temps.

11 Réalisation de l'emploi du temps

Il est difficile de trouver une solution exacte pour un problème d'ordonnancement, donc pour faciliter la génération de l'emploi du temps, nous avons mis en place une fonction de répartition aléatoire des cours dans la semaine. Cette fonction est rappelée un certain nombre de fois (nombre d'itérations à définir), mais le programme s'arrête si une solution est trouvée avant la dernière itération.

Afin de clarifier l'explication, nous rappelons que le programme de l'année est un ensemble de matières (Algèbre, Analyse, Electricité, etc.), et que chaque matière est un ensemble de cours (Algèbre : 14 cours de 2h, Analyse 12 cours de 2h, etc.), répartis sur les créneaux de la semaine. De plus, chaque promotion (B1, B2, B3, etc.) est constituée d'un certain nombre de classes (B1A, B2C, M1B, etc.). Enfin, un créneau correspond à 2h dans la semaine (Lundi 8h30-10h30, Mercredi 14h-16h, etc.), un cours de 2h se place donc sur un créneau, et un cours de 4h sur 2 créneaux.

Afin d'optimiser la répartition des cours dans la semaine, nous répartissons au préalable les matières du programme, et les enseignants correspondant sur le semestre, et ce pour chaque promotion. Ainsi, chaque classe d'une même promotion suivra les mêmes cours chaque semaine, ce qui garantit l'homogénéité de l'emploi du temps.

Pour ce faire, nous déterminons la semaine à laquelle doit se dérouler le premier cours de chaque matière, ainsi que le nombre de semaines nécessaires. Nous pouvons alors répartir les cours sur chaque semaine à l'aide de la fonction de répartition aléatoire. Cette fonction sera rappelée jusqu'à trouver une solution dans laquelle tous les cours ont été placés correctement. En cas d'échec (aucune solution parfaite), le programme donnera la solution dans laquelle un minimum de cours n'ont pas été placés correctement. Ces cours pourront toutefois être placés manuellement par la suite.

Algorithme 3 : Principe général de conception des emplois du temps

```
meilleurEDT ← ∞
repeat
  for all Promo do
    idMatieres ← liste contenant les id des matières que doivent suivrex Promo
    programmeSemestre ← liste contenant la répartition des idMatieres sur le semestre
    planningOk ← booléen indiquant si la génération de l'emploi du temps a rencontré des
    erreurs
  end for
  if planningOk then
    if nbCoursNonPlaces < meilleurEDT then
      meilleurEDT ← nbCoursNonPlaces
      Ecriture des emplois du temps dans les fichiers
    end if
  end if
until meilleurEDT > 0 and cmpt < nombreIteration
```

11.1 Répartition du programme sur le semestre

Afin d'optimiser la répartition des cours sur chaque semaine, nous commençons par répartir les matières sur les semaines du semestre. Pour chaque matière, nous allons donc indiquer la semaine dans laquelle doit commencer le premier cours, ainsi que le nombre de semaines durant lequel elle va être enseignée.

La répartition se déroule en deux étapes :
— Le tri des matières
— Le placement des matières sur le semestre

L'objectif est de répartir les cours sur le semestre de manière homogène. Il faut donc réussir à placer le maximum de matières les unes à la suite des autres. Pour ce faire, nous trions les matières de la plus longue à la plus courte (en nombre de semaines), puis nous les plaçons les unes après les autres dans le semestre, en faisant en sorte de les placer à la suite d'une autre matière dès que possible.

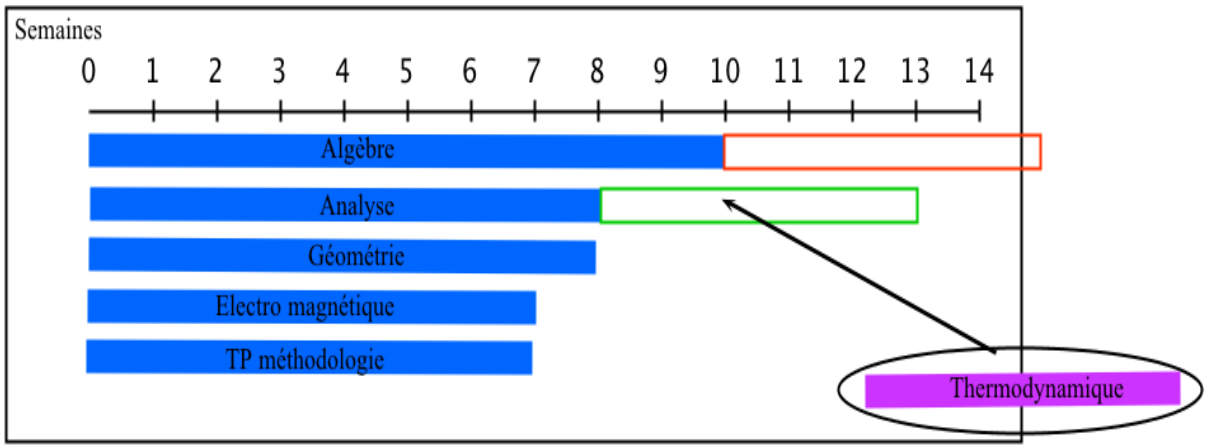

FIGURE 5 – Ajout d'un cours à la suite d'un autre sur le semestre

Un cours de 4 heures impose plus de contraintes. En effet, il s'agit d'un cours où le professeur et la classe doivent avoir en commun deux créneaux libres consécutifs dans la même demi-journée. C'est pourquoi un cours de 4 heures doit être planifié sur le semestre avant un cours de 2 heures.

Pour ce faire, les matières sont séparées en deux listes : une pour les cours de 4 heures et une autre pour les cours de 2 heures, et nous effectuons les répartitions sur le semestre comme expliqué précédemment.

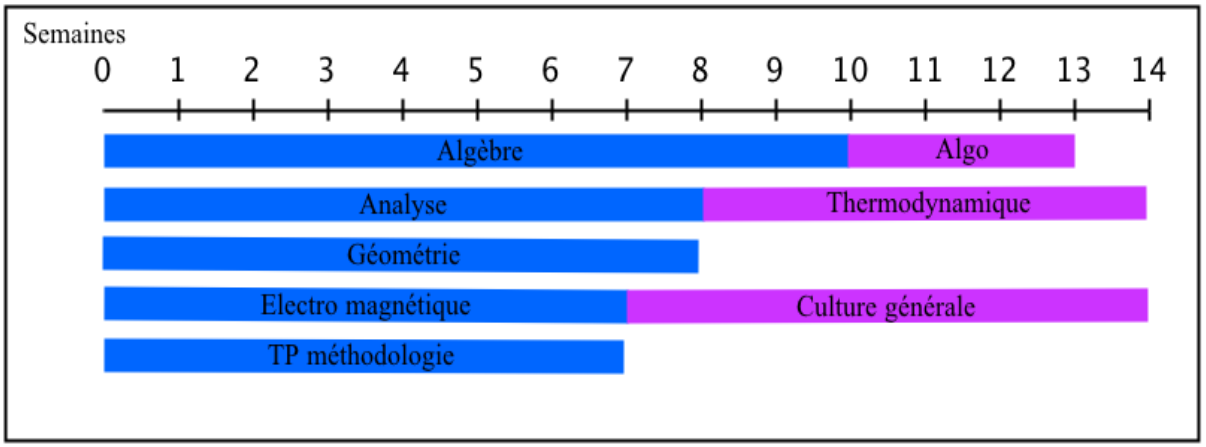

FIGURE 6 – Modélisation du planning du semestre

Pour chaque matière, nous avons besoin des informations suivantes :
— L'identifiant de la matière
— Le numéro de la semaine dans laquelle sera donné le premier cours
— Le nombre de semaines nécessaires pour enseigner l'ensemble de la matière
— La matière qui la suit, s'il y en a une

Algorithme 4 : Algorithme principal de la répartition des matières sur le semestre

Require: liste *idMatieres*, liste *programmeSemestre*
  for all *idMatieres* **do**
    if *idMatieres* est un cours sur 2h **then**
      *idMatieres*2 ← pushback *idMatieres*
    else
      *idMatieres*4 ← pushback *idMatieres*
    end if
  end for
  Tri de *idMatieres*2 par nombre de semaines de matière décroissant
  Tri de *idMatieres*4 par nombre de semaines de matière décroissant
  idMatieres est vidé
  for all *idMatieres*4 **do**
    idMatieres ← pushback *idMatieres*4
  end for
  for all *idMatieres*2 **do**
    idMatieres ← pushback *idMatieres*2
  end for
  return *programmeSemestre* ← repartitionDesCours(*idMatieres*)

Algorithme 5 : repartitionDesMatieres(*idMatieres*)

Require: liste *idMatieres* triée par nombre de semaines d'une matière et par cours de 4h et 2h
  initialisation de *programmeSemestre*
  for all *idMatieres* **do**
    for all *Matieres* in *programmeSemestre* **do**
      if *Matieres* a été placé **then**
        checkNextCourse(*idCourses*, *Matieres*)
        if *idMatieres* a été programmé **then**
          coursPlace ← *true*
          BREAK
        end if
      end if
    end for
    if *coursPlace* == *false* **then**
      programmeSemester ← pushback *idMatieres* en le programmant en début de semestre
    end if
  end for
  return *programmeSemestre*

Algorithme 6 : checkNextCourse($idMatieres$, $Matieres$)

if $Matieres$ a une autre matière après lui **then**
  checkNextCourse($idMatieres$, $Matieres$ du $Matieres$ suivant)
else if $semaineDebut_{coursProgrammes} + nbSemaine_{coursProgramme} + nbSemaine_{idMatieres} \leq nbSemaine_{semestre}$ **then**
  $programmeSemestre \leftarrow$ pushBack $idMatieres$ en le programmant après le cours $Matieres$
end if

Après avoir réalisé cet emploi du temps, nous pouvons commencer à placer les cours sur les créneaux des classes concernées.

11.2 Répartion des cours sur leurs créneaux

Grâce à la répartition des matières sur le semestre, nous allons savoir quels sont les cours à organiser sur chaque semaine. Cette répartition a été effectuée avec un objectif majeur : maintenir un emploi du temps homogène de manière à ce que chaque semaine, toutes les classes d'une même promotion aient reçus les mêmes cours.

Afin d'obtenir le meilleur emploi du temps possible, nous allons mettre en place un système de répartition aléatoire des cours. Le programme va réitérer cette fonction un certain nombre de fois, le nombre d'itération étant un paramètre modifiable. Le programme arrête d'itérer lorsqu'il trouve une solution parfaite (tous les cours placés correctement) ou lorsqu'il atteint le nombre d'itérations maximum. Il va alors générer les emplois du temps à partir de la solution retenue.

Cette fonction va récupérer pour chaque semaine la liste des cours à dispenser, et procéder au placement des cours, qui s'articule en deux étapes :

— Copie des cours placés la semaine précédente et qui doivent encore l'être sur cette semaine
— Placement des nouveaux cours de la semaine après récupération des professeurs qualifiés pour les enseigner

Lorsqu'un cours ne peut être placé faute de disponibilité, il est ajouté à une liste dédiée aux cours non placés, et est retiré de la liste des cours à placer. Cette liste permet d'établir la qualité de la solution : plus elle est remplie, et plus la solution est mauvaise. Cette liste contiendra l'identifiant de la matière, l'identifiant du professeur et les semaines où le cours aurait dû être placé.

Algorithme 7 : Algorithme principal de la répartition des cours sur les créneaux des classes

Require: le programme de la semaine *prog*
  for all *numSemaine* du semestre **do**
    progSemaine ← getProgrammeSemaine(*prog*, *numSemaine*)
    placementAncienCours(*progSemaine*, *listeClasses*, *numSemaine*)
    if il y a des nouveaux cours à placer dans la semaine **then**
      profSemaine ← getProfSemaine(*progSemaine*)
      placementNouveauCours(*listeClasses*, *progSemaine*, *profSemaine*, *numSemaine*)
    end if
    if une erreur est survenu dans la réalisation du planning **then**
      return 0
    end if
  end for
  return 1

Algorithme 8 : Méthode pour récupérer le programme d'une semaine

Require: le programme de la semaine *prog* et la semaine du semestre *numSemaine*
  for all *cours* du programme **do**
    if semaineDebut de *cours* ≤ *numSemaine***and** semaine fin de *cours* > *numSemaine* **then**
      progSemaine ← pushback *cours*
    end if
  end for
  return *progSemaine*

11.2.1 Placement de cours déjà fixé la semaine précédente

Lorsqu'on récupère le programme d'une semaine, la première étape consiste à vérifier si certains cours ont déjà été placés la semaine précédente, et le cas échéant à les copier dans l'emploi du temps de cette semaine, en vérifiant bien que le professeur est toujours disponible. Le cours sera ajouté à la liste des cours non placés si le professeur n'est plus disponible. Lorsqu'un cours a été placé pour toutes les classes d'une promotion, nous le supprimons de la liste des cours à dispenser. Dans le cas contraire, le cours est ajouté à la liste des cours non placés. Ceci va permettre à une classe d'avoir le même cours sur le même créneau avec le même professeur d'une semaine à l'autre.

Algorithme 9 : Méthode pour placer les cours précédemment planifiés

Require: le programme *prog* de la semaine, la liste des classes *classes*, la semaine *numSemaine*

nbCourseAjout ← 0
nouveauCours ← *false*
if La première semaine a déjà été planifiée **then**
  for all *cours* du programme de la semaine **do**
    coursDejaProgrammeAvant(*cours*, *classes*, *nbCoursAjout*, *nouveauCours*)
    if *nbCoursAjout* =nombre *classes* **then**
      coursASupprimer ← pushback *cours*
    else
      nouveauCours ← *faux*
    end if
    nbCourseAjout ← 0
  end for
  for all *coursASupprimer* **do**
    progSemestre ← supprimer *progSemestre*(*coursASupprimer*)
  end for
end if

Algorithme 10 : Méthode pour savoir si un cours a déjà été programmé avant

Require: le cours de la semaine *cours*, la liste des classe *classes* , la semaine du semestre *numSemaine*

for all *classes* **do**
  if *classes* a reçu le cours la semaine *numSemaine* − 1 **then**
    ajoutDuCours(*classes*, *cours*, *numSemaine*)
    nbCoursAjoute ← *nbCoursAjoute* + 1
  end if
end for
if *nbCoursAjoute* =nombre de *classes* **then**
  nouveauCours ← *faux*
else
  nouveauCours ← *vrai*
end if

Algorithme 11 : Méthode pour ajouter le cours par rapport à la semaine d'avant

Require: la classe $classe$, la matière $cours$, la semaine du semestre $numSemaine$
$idProf \leftarrow$identifiant du professeur donnant $cours$ la $numSemaine - 1$
$creneau \leftarrow$créneau de $cours$ la $numSemaine - 1$
if $cours$ est sur 4 heures **then**
if $prof$ est disponible à $numSemaine, creneau$**and**$prof$ est disponible $numSemaine, creneau + 1$**and**$classe$ est disponible à $numSemaine, creneau$**and**$classe$ est disponible $numSemaine, creneau + 1$ **then**
planification $cours$ avec $prof$ sur $numSemaine$ et $creneau$
planification $cours$ avec $prof$ sur $numSemaine$ et $creneau + 1$
end if
else
if $prof$ est disponible à $numSemaine, creneau$**and** $prof$ est disponible $numSemaine, creneau + 1$ **then**
planification $cours$ avec $prof$ sur $numSemaine$ et $creneau$
end if
end if

11.2.2 Planification des nouveaux cours du semestre

Une fois que tous les cours de la semaine précédente ont été traités, nous pouvons nous occuper des nouveaux cours. Nous disposons de la liste des cours de la semaine dans laquelle il ne reste que les nouveaux cours. Nous commençons par récupérer l'ensemble des professeurs capables d'enseigner ces matières.

Nous procédons alors à la sélection du couple classe-professeur ayant le moins de créneaux en commun. Ce couple sera prioritaire sur les autres car c'est celui qui dispose du moins de créneau et qui est par conséquent le plus contraignant.

C'est maintenant qu'intervient le facteur aléatoire : nous allons répertorier les créneaux communs au professeur et à la classe, et en choisir un au hasard.

Lorsqu'un cours n'a pas pu être ajouté à une classe, il est ajouté à la liste des cours non placés.

Dans le cas d'un cours de 4 heures, il se peut que nous n'ayons aucun créneau permettant de mettre les 4 heures à la suite. Dans ce cas nous mettons le cours dans la liste des cours n'ayant pu être planifiés.

Algorithme 12 : Méthode pour ajouter un nouveau cours

```
Require: progSemaine, profSemaine, listClasses
  nbCours ← nombre de cours dans progSemaine∗ nombre de classes dans listClasses
  for i := 0 to nbCours do
    meilleureConnexion(progSemaine, profSemaine, listClasses, numSemaine)
    if on trouve une connexion then
      ajoutCours(progSemaine, profAAjouter, classesAAjouter, numSemaine)
    else
      return 0
    end if
  end for
  return 1
```

Algorithme 13 : Méthode pour trouver la meilleure connexion

```
Require: progSemaine, profSemaine, listClasses, numSemaine
  buf ← 23
  for all profSemaine do
    for all listClasses do
      nbConnections ← nbCreneauxCommuns(profSemaine, listClasses, numSemaine)
      if nbConnections > 0 and nbConnections < buf then
        buf ← nbConnections
        profAAjouter ← profSemaine
        promoAAjouter ← listClasses
      end if
    end for
  end for
```

Algorithme 14 : Méthode pour compter le nombre de créneaux communs

Require: $prof, classe, numSemaine, progSemaine$)
  for all $cours$ donnés par $prof$ **do**
    if *promo* doit recevoir $cours$ sur $numSemaine$ **and** $cours$ n'a pas encore été placé pour *promo* sur $numSemaine$ **then**
      $coursPossible \leftarrow vrai$
      BREAK
    else
      $coursPossible \leftarrow faux$
    end if
  end for
  if $coursPossible$ **then**
    return $nbConnection \leftarrow$ somme des disponibilités communes de $prof$ et *promo*
  end if
  return -1

Algorithme 15 : Méthode pour ajouter un cours à une classe

Require: $progSemaine, profAAjouter, classesAAjouter, numSemaine$
  for all $cours$ de $profAAjouter$ **do**
    if *promo* doit recevoir $cours$ sur $numSemaine$ **and** $cours$ n'a pas encore été placé pour *promo* sur $numSemaine$ **then**
      creationCours($prof$, *promo*, $cours$, $numSemaine$)
      BREAK
    end if
  end for

Algorithme 16 : Méthode pour créer le cours à la classe

```
if cours n'a pas été programmé à numSemaine − 1 pour classes then
    ajout de cours dans la liste des cours non planifiés
else if cours est sur 4 heures then
    for all creneau do
        if classe est libre à numSemaine, creneau and classe est libre à
        numSemaine, creneau + 1 and prof est libre à numSemaine, creneau and prof est
        libre à numSemaine, creneau + 1 then
            creneauxPossibles ← pushback creneau
        end if
    end for
    if creneauxPossibles n'est pas vide then
        creneau ← choix aléatoire dans creneauxPossibles
        mise en place du cours et des données (cours, classe, prof, numSemaine, creneau)
        mise en place du cours et des données (cours, classe, prof, numSemaine, creneau + 1)
    end if
    ajout de cours dans la liste des cours non planifiés
else
    for all creneau do
        if classe est libre à numSemaine, creneau and prof est libre à numSemaine, creneau
        then
            creneauxPossibles ← pushback creneau
        end if
        creneau ← choix aléatoire dans creneauxPossibles
        mise en place du cours et des données (cours, classe, prof, numSemaine, creneau)
    end for
end if
```

12 Déplacements des cours

Une fois l'emploi du temps réalisé, nous avons la possibilité de déplacer des cours et/ou de placer manuellement les cours qui n'ont pu être placés lors de l'exécution du programme.

12.1 Déplacement d'un cours existant

Dans un premier temps, il faut choisir la classe pour laquelle nous souhaitons effectuer le changement. Ensuite nous sélectionnons le cours à déplacer, et la liste des créneaux communs au professeur et à la classe apparait. En sélectionnant le créneau dans lequel nous souhaitons déplacer le cours, le changement s'effectue directement en mettant à jour les données relatives à la plannification.

12.2 Ajout d'un cours non placé

La liste des cours non positionnés apparait. Il n'y a qu'à sélectionner le cours que nous voulons placer, et la liste des créneaux sur lequel il peut être mis apparait. Les instructions suivantes sont les mêmes que pour le déplacement d'un cours.

13 Discussions

Le problème étant vraisemblablement Np-difficile, nous doutons de la possibilité d'établir un algorithme de résolution exacte dans un temps raisonnable, étant donnée la taille des instances que nous avons observées. Nous avons considéré qu'une approche heuristique (voire méta-heuristique), offrant une solution réalisable mais non nécessairement optimale, serait plus appropriée pour des raisons pratiques évidentes.

Le temps nous faisant défaut, nous n'avons pas pu implémenter les contraintes liées à la M2 et résoudre le problème des salles et des locaux.

Par ailleurs, les TPs posent problème parce que cela revient à avoir plusieurs cours sur un même créneau et d'avoir plusieurs fois le même cours dans la semaine.
Le système des TP est un peu particulier : chaque série peut avoir 1 à 2 TP par semaine, et quoi qu'il arrive, il y aura toujours au moins une série de la classe dans chaque TP. Pour résoudre le problème des TP, on peut donc envisager de créer 2 sessions TP1 et TP2 à placer dans l'emploi du temps, et de réaliser la rotation des TP ultérieurement à l'aide d'un programme tiers.

Concernant le problème des salles, à partir de l'emploi du temps que nous avons créé, nous pouvons allouer à chaque créneau une salle aléatoirement jusqu'à avoir une répartition possible.

14 Conclusion

A l'issue de ce projet, nous avons pu aboutir à une résolution heuristique du problème de planification dans le cadre de l'école. Même si cette solution n'est pas optimale, elle constitue tout de même une base solide pour établir les emplois du temps des 4 premières promotions de l'école.

A partir de la saisie des données sur les professeurs, les promotions et les matières, le programme est capable de générer les emplois du temps de chacune des classes et d'afficher les cours non placés le cas échéant, en proposant de les rajouter manuellement. Le programme propose également des fonctions de maintenance pour déplacer des cours en cas d'imprévu au fil de l'année.

Dans l'état actuel des choses, les données d'entrée/sortie du programme sont stockées dans des fichiers textes, sans aucune mise en forme. Afin de rendre le programme compréhensible et ergonomique, une interface graphique est en cours de réalisation dans le cadre d'un projet de M1.

Table des figures

1 Diagramme de cas d'utilisation, génération de l'emploi du temps 10
2 Diagramme de cas d'utilisation, maintenance de l'emploi du temps 12
3 Diagramme de classes de mi-projet 16
4 Diagramme de classes final envisagé 17
5 Ajout d'un cours à la suite d'un autre sur le semestre 24
6 Modélisation du planning du semestre 24

Liste des Algorithmes

1 Pré-traitement du nombre de professeurs 20
2 Pré-traitement du nombre d'heures sur le semestre 21
3 Principe général de conception des emplois du temps 23
4 Algorithme principal de la répartition des matières sur le semestre 25
5 repartitionDesMatieres($idMatieres$) 25
6 checkNextCourse($idMatieres$, $Matieres$) 26
7 Algorithme principal de la répartition des cours sur les créneaux des classes . . . 27
8 Méthode pour récupérer le programme d'une semaine 27
9 Méthode pour placer les cours précédemment planifiés 28
10 Méthode pour savoir si un cours a déjà été programmé avant 28
11 Méthode pour ajouter le cours par rapport à la semaine d'avant 29
12 Méthode pour ajouter un nouveau cours 30
13 Méthode pour trouver la meilleure connexion 30
14 Méthode pour compter le nombre de créneaux communs 31
15 Méthode pour ajouter un cours à une classe 31
16 Méthode pour créer le cours à la classe 32

Références

[1] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system : optimization by a colony of cooperating agents. *Trans. Sys. Man Cyber. Part B*, 26(1) :29–41, February 1996.

[2] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman &amp ; Co., New York, NY, USA, 1990.