Compte rendu du Projet d'IA

« Des coulisses à la scène »

Corentin BOUCHAUDON & Lydia OULD OUALI

Table of Contents

I. MODELISATION ET MISE EN OEUVRE

Le projet est divisé en trois parties : la récupération des données, les plannings et les résultats. La récupération des données se fait à partir du fichier lectureInstance.js. Ce fichier a pour but de récupérer les données et les traiter afin qu'elles puissent être utilisées. Les plannings (planning1.mod, planning2.mod, planning3.mod) sont les programmes en eux-mêmes, ceux qui vont calculer les solutions à partir des données récupérées grâce au fichier lectureinstance.js. Les résultats, eux sont stockés dans des fichiers situés dans le dossier « résultat ». Ils contiennent toutes les solutions trouvées grâce aux plannings.

a) Planning1.mod

Ce programme cherche à générer un planning des sessions qui soit compatible avec l'ensemble des contraintes temporelles (précédences, écart entre sessions, disponibilités des participants) mais en faisant abstraction des contraintes liées aux salles et au budget. Le fichier est scindé en grandes parties.

La lecture du fichier d'instance

Pour avoir un code orienté objet, nous représentons les instances avec plusieurs paramètres par des tuples. Ainsi, nous avons :

- Bloc : Représente un bloc
- Session : Représente une session
- Précède : Représente l'organisation chronologie entre deux sessions
- Indisponible : Formalise l'indisponibilité d'un individu

Ces tuples permettent entre autres de faciliter la récupération des données brutes (fichier .txt) en données utilisables par OPL.

Il y a des instances qui n'ont qu'un seul paramètre. Ces instances se trouvent dans les fichiers dont les noms suivent le format cal-nombre de jours-nombre de créneaux.txt. Nous les avons représentés par de simples variables :

- nbJoursMax : le nombre de jour maximum sur lequel se déroule l'instance
- nbCreneauxMaxParJour : le nombre de créneaux totaux maximum sur lequel se déroule l'instance

Nous avions besoin de déclarer des variables permettant de récupérer les tableaux de string des tuples pour pouvoir les passer en paramètre (car on ne pouvait pas faire de add en javascript).

- blocsIn : est un tableau de string permettant de récupérer les intervenants du bloc
- listeCreneaux : est un tableau d'integer qui récupère les créneaux d'indisponibilités d'un intervenant ou d'une salle de la session
- listeSession : est un tableau de string qui récupère la liste des intervenants de la session
- listJours : est un tableau d'integer qui récupère les jours d'indisponibilités d'un intervenant ou d'un salle de la session

Ensuite, nous remplissons toutes ces structures à l'aide des données depuis un execute. Cet execute fait appels à des fonction de lectureInstance.js pour l'extraction ainsi que le traitement des données.

Prétraitement sur les données d'instance

Nous avons décidé de stocker dans des tableaux les informations suivantes :

- {string} codeDeSession : Les identifiants de toutes les sessions
- {string} codeDeBloc : Les identifiants de tous les blocs

- {string} intervenantsDuBloc[codeDeBloc] : Les intervenants interne au bloc
- {string} blocDuBloc[codeDeBloc] : Les blocs se situant dans le bloc
- {string} temp1[codeDeBloc] : Un tableau temporaire qui permet la modification des intervenants Du Bloc
- {string} intervenantsDeSession[codeDeSession] : Les intervenants à la session
- {int} sessionIndisponible[codeDeSession] : Les indisponibilités d'une session par rapport aux indisponibilités des personnes)

- {string}intervenants : Les noms des intervenants de l'instance
- {string} sessionIntervenant[intervenants] : Les noms des sessions auquel participe l'intervenant

Et dans des variables simples :

- dureeMinimaleSession : La durée minimale d'une session
- dureeMaximaleSession : La durée maximale d'une session

Les structures ci-dessus sont avant tout un moyen de simplifier le traitement des données dans le but d'avoir des contraintes sur les dvar les plus simples possible.
Ensuite, on remplit toutes ces structures dans un execute. Cet execute est divisé en plusieurs parties. Tout d'abord, il remplit les tableaux des intervenants par bloc. Puis, il replie le tableau des intervenants par session. Ensuite, il gère l'indisponibilité des personnes et finalement, il récupère tous les intervenants de l'instance et donne toutes les sessions auxquelles participe l'intervenant.

Le cadre rouge est ici pour indiquer que le traitement de données dans ces tableaux consomme beaucoup de ressources et donc de temps lors de l'exécution. En effet pour récupérer les intervenants de chaque session il a fallu d'abord récupérer les intervenants de chaque bloc. Comme un intervenant pouvait être aussi un bloc il a fallu faire une fonction récursive qui s'occupe de récupérer les informations récursivement.

Variables de décision

Nous avons trois variables de décision pour résoudre le problème :

- dureeTotaleInstance : Récupère le plus grand creneaux de finSession. Cette variable n'est pas nécessaire, mais on l'a ajouté pour obtenir un résultat optimisé
- debutSession[codeDeSession] : tableau indexé par les noms des sessions et contenant le créneau où commence la session
- finSession[codeDeSession] : tableau indexé par les noms des sessions et contenant le créneau où fini la session

Contrainte du modèle

Même s'il n'était pas demandé d'optimiser le résultat, nous avons décidé de le faire sur la contrainte durée totale. Cela nous permettait de savoir si le temps d'exécution restait intéressant en calculant tous les résultats. Voici les contraintes auxquelles est soumis le minimize :

- La durée totale de l'instance est égale à la fin de la dernière session, calculée en créneaux (utilisation d'un max)
- La fin d'une session est définie par le début de cette dernière additionnée à sa durée (utilisation d'un forall sur les sessions pour que toutes les fin respect la contrainte)
- La session ne peut pas s'étendre sur plusieurs jours (utilisation d'un forall sur les session)
- La contrainte précède (utilisation d'un forall sur les précèdes)
- Un intervenant ne peut pas participer à deux sessions en même temps (un forall sur les intervenants avec un forall à l'intérieur sur les sessions de ces intervenants puis comparaison des sessions entre elles)
- Gestion des indisponibilités (un forall sur les sessions avec à l'intérieur un forall sur les sessions indisponibles)

Résultats

Finalement, nous écrivons les résultats obtenus dans le fichier résultat à travers un simple execute selon le format demandé c'est-à-dire l'ajout des id des salles

Planning <S> <J> <H>

→ La session a été programmée le jour J et doit débuter au créneau H de cette journée

b) Planning2.mod

Ce programme cherche à préciser dans quelle salle se déroule chaque session et s'assurer de la compatibilité avec les disponibilités des salles et leur capacité d'accueil. Il est scindé en plusieurs parties. Étant très similaire au planning1.mod, nous n'allons pas tout réécrire mais juste exposer les changements.

La lecture du fichier d'instance :

De nouveau tuples ont été ajoutés pour répondre à la gestion des nouvelles données :

- BesoinSpecifique : Représente un besoin spécifique
- Salle : Représente la salle avec un ajout d'identifiant en integer pour faciliter la gestion des dvar. En effet, une dvar ne peux pas être de type string donc nous avons ajouté un identifiant en integer

Egalement, un nouveau tableau a été ajouté, celui-ci récupère le tableau de string du tuple de BesoinSpecifique : listeBesoinSpecifique qui contient pour chaque salle, les sessions qui doivent obligatoirement se dérouler dans cette salle.

Prétraitement sur les données d'instance :

Nous avons décidé de stocker dans des tableaux les nouvelles informations suivantes :

- {String} codeDeSalle : Liste des identifiants des salles
- {int} indisponibiliteSalle[codeDeSalle] : Les indisponibilités pour chaque salle, en créneaux
- salleDeSession[codeDeSession] : La salle où se déroule chaque session
- int nbPersonnesSessions[codeDeSession] : Le nombre d'intervenants pour chaque session

Ces tableaux permettent entre autres de simplifier les contraintes sur les salles en limitant le nombre de boucle forall dans le code
Par ailleurs, le tableau *sessionIndisponibleClient* est devenu :

- {int} sessionIndisponibleClient[codeDeSession] : Les indisponibilités d'une session (par rapport aux indispos des personnes)

Des variables simples ont également été ajoutées :

- int nbSalles : Le nombre total de salle dans l'instance
- {int} sessionIndisponible[codeDeSession] : Les indisponibilités d'une session (par rapport aux indispos des personnes)

Puis dans l'execute, on remplit les nouvelles variables créées/modifiées.

Variables de décision :

Une variable de décision a été ajoutée :

- salleSession[codeDeSession] : tableau indexé par les noms des sessions et contenant le numéro de la salle où doit se situer la session

Elle permet de récupérer l'identifiant en integer de la salle où doit se faire la session. Lors de l'écriture dans un fichier nous nous servirons de cette identifiant en integer pour retrouver le nom de la salle en string

Contrainte du modèle :

Nous avons ajouté de nouvelles contraintes :

- Une salle ne peut pas dépasser un certain nombre d'intervenants
- Sessions devant absolument se dérouler dans une salle particulière
- Deux sessions ne peuvent pas se dérouler dans la même pièce en même temps
- Respect des indisponibilités des salles

Résultats :

Finalement, nous écrivons les résultats obtenus dans le fichier résultat à travers un simple execute selon le format demandé c'est-à-dire l'ajout des id des salles à ce qui était déjà proposé.

planning <S> <J> <H> <idSalle>

Comme expliqué ci-dessus l'identifiant de la Salle (string) est récupérer grâce à son identifiant en integer.

c) Planning3.mod

Ce programme cherche à préciser les indemnités des intervenants, en fonction de s'ils travaillent plusieurs jours consécutifs ou s'il y a des écarts dans les répétions. Étant très similaire au planning2.mod, nous n'allons pas tout réécrire mais juste exposer les changements.

La lecture du fichier d'instance :

Un nouveau tuple a été ajouté, il s'agit du tuple indemnité qui représente les indemnités d'un groupe d'intervenant. Il y a également la création d'un nouveau tableau de String (personnels) qui récupère les intervenants qui suivent ce régime d'indemnité.

Prétraitement sur les données d'instance :

Dans cette partie du code nous avons ajouté uniquement un entier « coutMaximal. Qui prendra comme valeur le coût maximal qu'il est possible d'atteindre avec les indemnités de l'instance (ce cout est normalement impossible à atteindre car il a pour valeur les indemnités dans les pires cas). Cette variable permettra de réduire la range du coût total de l'instance.

Variables de décision

Deux variables de décision ont été ajoutées :

- Couttotal : Il s'agit de la variable de décision qui prendra la somme de toutes les indemnités de l'instance
- indemniteParPersonneEtSession[intervenants][codeDeSession] : tableau indexé par les intervenants et le nom des sessions et contenant l'indemnité à payer pour cet intervenant dans cette session.

Contrainte du modèle :

Nous avons ajouté deux contraintes :

- Coût total qui doit être égal à la somme des indemnités par personne et par salle.En fonction de si l'intervenant enchaine deux jours de travail ou s'il a une pose entre ses sessions de répétitions

Résultats :

Malheureusement, pour ce planning, nous n'avons pas réussi à obtenir de résultats, sauf si l'on demande de sortir le premier résultat possible (enlever le minimize).

D) LectureInstance.js

Ce fichier est l'un des plus importants du projet, en effet il s'agit du fichier qui va s'occuper de la gestion des différents fichiers du projet. Dans ce fichier nous avons donc créé un grand nombre de function de type Javascript et surtout OplScript. Il y quatre types de fonction dans ce fichier.

Récupération des données

Dans cette partie, il s'agit de fonctions permettant qui permettent de récupérer les informations de l'instance du projet. La première est recupérerDonnées, c'est une fonction qui va charger les documents de l'instance pour récupérer les informations et les écrire dans un tableau de string.

La seconde fonction est supprimée espace. En effet dans le projet nous avons observé que les données des fichiers .txt contenait différents types d'espace qui nous empechait de récupérer facilement les données c'est pourquoi à l'aide de la fonction isEspace nous supprimons les espaces pour retourner une liste de String sans « » à l'intérieur.

Traitement des données.

Les fonctions permettant de traiter les informations des données représentent la majorité des fonctions de lectureInstance.js. On se sert des identifiants de chaque ligne, qui sont les premières string de la première colonne d'un tableau de string pour reconnaître le type d'information que l'on souhaite récupérer(« session » pour récupérer les sessions ou bloc pour récupérer les blocs).

Les fonctions les plus compliquées dans cette partie sont celles traitant les indisponibilités car il fallait en plus de récupérer les informations vérifier si les jours étaient un interval de jours ou un jour simple. Ainsi que vérifier la présence des créneaux

Parmi toutes ces fonctions certaines ont besoins de recevoir un tableau d'integer ou de string en paramètre. Cela nous permet de remplir les tableaux se situant dans les tuples puis de remplir les tableaux de tuples. De plus ces fonctions ont pour paramètre un tableau de tuples du type in/out ce qui permet de récupérer plus facilement les différents tuples des données.

Ecriture des résultats

Dans le but d'écrire les résultats des différentes instances sur les différents modèles nous avons écrit une fonction ecrireResultat qui va créer ou modifer un fichier dans le dossier résultat.

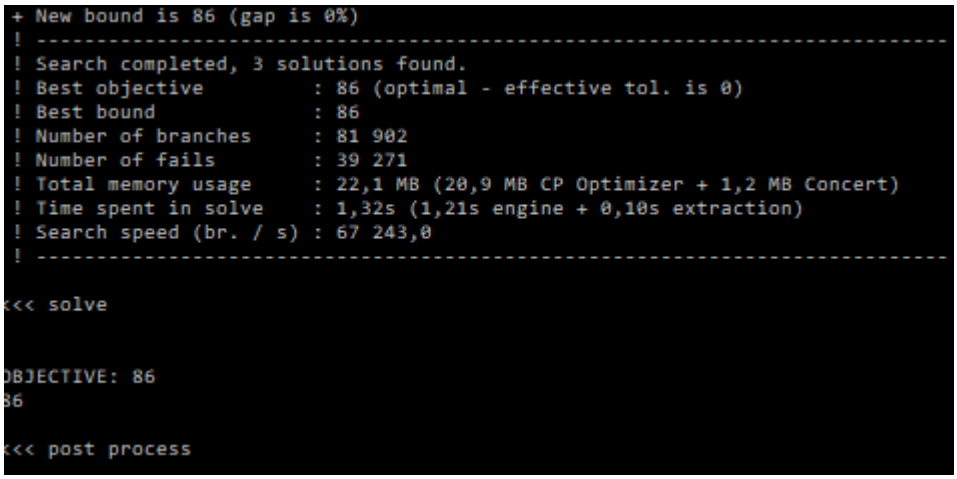
Autres

Comme expliqué dans le planning1.mod nous avons utilisé une fonction récursive pour trouver les intervenants de chaque session. Cette fonction pour éviter de prendre trop de place sur le fichier planningX.mod a été déplacemer sur lectureInstance.js

II. ANALYSE DES RESULTATS

a. Planning 1.mod

Nous avons donc commencé par exécuter l'instance bourgeois1.dat, nous avons trouvé qu'il y avait au minimum 86 créneaux pour trouver une solution possible au problème qui nous a été donnés.



Nous avons décidé d'afficher le nombre de créneaux minimal pour ce planning car il nous permettait d'ensuite comprendre pourquoi les autres instances (bourgeois 2 à 4) ne donnait pas de résultat. En effet le nombre de créneaux maximum qu'ils pouvaient avoir ne dépassait pas les 86 (22 jours (21+1)) du coup il était impossible d'avoir des résultats. Pour vérifier cette théorie nous avons créé de nouvelles instances ayant plus de 86 créneaux disponibles. Bourgeois5 avait moins de jours mais autant de créneau que le 1. Le résultat est identique à bourgeois 1. Nous avons alors testé bourgeois6 qui avait moins de jours que les autres bourgeois mais qui avait 6 créneaux par jour. Même si le nombre de créneaux utilisé est beaucoup plus important (100), le nombre de jours est passé de 22 pour bourgeois1 à 17 pour bourgeois6.

De plus le temps d'exécution est court, vous pouvez voir sur la capture d'écran ci-dessus, que l'exécution de bourgeois1.dat avec le planning1 prend moins de 2 secondes.

Pour les résultats, vous pouvez vous renseigner en regardant dans le dossier résultat, vous y trouverez tous les résultats de nos exécutions

b. Planning 2.mod

Le second planning permettait la mise en place du système de salle dans le problème. Nous avons donc exécuté les mêmes instances sur ce modèle.

```
+ New bound is 86 (gap is 0%)
! ------------------------------------------------------------------------
! Search completed, 2 solutions found.
! Best objective         : 86 (optimal - effective tol. is 0)
! Best bound             : 86
! Number of branches     : 428 084
! Number of fails        : 203 290
! Total memory usage     : 54,5 MB (53,2 MB CP Optimizer + 1,3 MB Concert)
! Time spent in solve    : 8,17s (8,07s engine + 0,10s extraction)
! Search speed (br. / s) : 53 013,5
! ------------------------------------------------------------------------

<<< solve


OBJECTIVE: 86

<<< post process


<<< done
```

Comme pour le modèle 1 les instances bourgeois 2 à 4 n'ont pas de solution, s'il n'avait pas de solution dans le planning 1 il aurait été incompréhensible qu'ils en aient une dans le 2. Concernant l'instance bourgeois1, le résultat en nombre de jours est le même, et l'attribution des salles pour les sessions respectent bien les contraintes données. On observe cependant une hausse du temps de recherche pour trouver une solution. En effet, le temps passe d'environ 1 seconde pour le planning1 à 8 secondes pour le planning2.

Cette observation peut être justifiée par l'ajout des nouvelles contraintes qui forcent le programme à effectuer une recherche plus poussée. Dans le cas de planning 2 nous avons encore optimisé le résultat sur le nombre de créneaux pour avoir un temps d'exécution plus intéressant.

On peut aussi comparer les résultats avec bourgeois6, le temps d'exécution de bourgeois6 restes très court on peut conclure que le fait d'avoir autant de créneaux permet d'éviter d'avoir un temps d'exécution trop grand.

c. Planning3.mod

Le dernier planning permettait l'ajout de la gestion des indemnités des intervenants. Nous avons rencontré un problème avec ce planning. En effet le temps d'exécution a été augmenté de manière exponentielle. Cela explique le fait qu'il n'y a pas de résultat concret de planning3 dans le dossier résultat. Cependant vous pourrez trouver les fichiers bourgeois1_planning1.res et bourgeois6_planning1.res qui correspondent à un résultat possible de répartition des indemnités dans ces instances.

```
*                 9 514  35,72s       7     41 355  = couttotal
! ------------------------------------------------------------------------------
! Search completed, 1 solution found.
! Number of branches     : 74 514
! Number of fails        : 30 925
! Total memory usage     : 213,4 MB (211,9 MB CP Optimizer + 1,5 MB Concert)
! Time spent in solve    : 35,73s (35,10s engine + 0,62s extraction)
! Search speed (br. / s) : 2 122,5
! ------------------------------------------------------------------------------

<<< solve



OBJECTIVE: no objective
41355

<<< post process


<<< done
```

Nous avons déterminé que le problème venait de la variable de décision indemnitéParPersonneEtSession qui force le programme à tester toutes les combinaisons de créneaux et de salles. Ce qui est très gourmand. Pour augmenter la performance du programme nous avons réfléchi à différents moyens. Le premier c'est l'utilisation de variable de décision de type interval, grâce aux fonctions sur interval qui sont très dures nous aurions plus de chances de trouver un résultat rapidement. L'autre solution serait la mise en place d'un niveau d'exigence pour les contraintes, ce qui permettrait d'avoir un résultat qui correspond aux contraintes les plus importantes et dont les autres contraintes sont moins respectées. Dans ce cas, le cout total a les contraintes les plus souples donc on pourrait essayer de trouver une solution qui ne serait pas le cout minimal mais qui s'en approcherai.