Notice d'utilisation P-ANDROIDE

Juin 2020

1 Installation

Il vous suffit ensuite de cloner le répertoire de la solution `https://github.com/herpsonc/ChicNRP`.
Créez ensuite, avec votre IDE préféré, un nouveau projet à partir du répertoire.

Il vous faudra ensuite installer une librairie qui est utilisée dans le projet (pour la programmation linéaire). Pour cela, dirigez vous ici : `https://scip.zib.de/index.php#download` et choisissez la version 7 de la librairie et l'OS qui vous correspond. Une fois le fichier téléchargé, suivez les instructions d'installation.
Après avoir fait cela il faudra configurer le projet pour pouvoir utiliser la librairie et pour ce faire vous devez allez dans les propriétés du projet et indiquer les valeurs suivantes :

En haut de la fenêtre de propriétés, sur `Plateforme`, choisir : `Toutes les plateformes`.

Puis, dans `Propriétés de configuration`, indiquez les valeurs suivantes :

- Répertoires VC++ :
  - Répertoire Include : `/$(SolutionDir)/src/scip/include;`
  - Répertoire de bibliothèques : `/$(SolutionDir)/src/scip/lib;`
- C/C++ :
  - Général :
    - Autres répertoires Include : `/$(SolutionDir)/src/scip/include;`
- Editeur de liens :
  - Général :
    - Répertoires de bibiliothèques supplémentaires : `/$(SolutionDir)/src/scip/lib;`
  - entrée :

* Répertoires de bibiliothèques supplémentaires : `libscip.lib; soplex.lib; soplex-pic.lib; zimpl.lib; zimpl-pic.lib`

Si vous comptez utiliser les plannings de Mars et Avril 2020 il vous faudra ajouter le dossier data dans le dossier de l'executable, ce dossier contient les fichiers csv permettant de charger les plannings pré défini.

2 Modèle

L'instanciation du modèle est indispensable pour décrire le problème de planning, que ce soit pour utiliser la résolution par méthode heuristique ou par la méthode de programmation linéaire. L'architecture est relativement simple, le Modèle possède des services qui eux même possèdent des postes et des agents.

Pour créer un modèle on commence par déclarer un Model:

```
int firstDay = 5;
int nbDays = 31;
float overtime = 25
Model m = Model(firstDay,nbDays,overtime);
```

firstDay est le premier jour du mois à choisir entre 0 et 6 (correspond à Lundi, Mardi...), on défini dans l'exemple au dessus le premier jour du mois comme étant un samedi. *nbDays* correspond au nombre de jours dans le mois et *overtime* le nombre d'heures supplémentaires autorisées pour les agents.

2.1 Service

On étoffe ensuite le Model en ajoutant des Services en renseignant simplement un identifiant pour le service (ici on ajoute le service "GHR"):

```
Service* ghr = new Service("GHR");
m.addService(ghr);
```

2.1.1 Ajout d'un planning pré défini

* Pour ajouter un planning pré défini il faut préalablement créer un fichier au format csv et le charger avec la fonction suivante:

```
m2.getService()[0]->loadPredefinnedPlanning("data/PrePlannings/GHR.csv", m.getNbDays(),
```

Vous pouvez vous inspirer des pré plannings déjà crées pour en ajouter de nouveaux, il n'y a pas d'obligations sur le nombre de lignes, il peut y en avoir plus ou moins que d'agents, de même pour le nombre de jours.

2.2 Postes

2.2.1 Création et ajout d'un Poste à un Service

Les postes s'ajoutent directement à un ou plusieur Services:

```
Post* jg = new Post("Jg",12.25, true);
int workAtt = 0;
int dayAtt = 2;
jg->addAttribut(workAtt);
jg->addAttribut(dayAtt);

//On ajoute le poste au service GHR
ghr->addPost("Jg");
```

Pour la création du poste il suffit de déclarer un identifiant (String) et la durée du poste (float), le dernier paramètre qui est un booléen sert à définir si les heures doivent être comptées le week-end (Samedi et Dimanche) et si elles doivent être comptées dans le calcul des heures effectuées par les agents dans le planning final, utilisé principalement pour les congés annuels. Ce paramètre est défini sur *true* si il n'est pas précisé.
La méthode addAttribut permet d'ajouter au poste des attributs qui serviront pour les Contraintes qui seront décrites plus tard.

2.2.2 Postes requis

Pour renseigner la contrainte C5 concernant les postes requis par jour on utilise la méthode *addPostRequired* de la classe Service:

```
//Chaque jour il faut 1 agent affecté au poste jg.
ghr->addPostRequired(jg,1);
```

On peut aussi utiliser la surcharge de la méthode pour y préciser le jour *Day* où le nombre de poste est requis:

```
//Chaque mercredi il faut 2 agent affecté au poste jg.
ghr->addPostRequired(jg,2,Day::Wednesday);
```

2.2.3 Ajout d'un poste par défaut au Modèle

Pour le bon fonctionnement des algorithmes il faut renseigner un poste par défaut avec la méthode *setDefaultPost* de Model, par exemple un poste qui décrit un repos pour l'agent :

```
Post* repos = new Post("Repos",0);
repos->addAttribut(5);

m.setDefaultPost(repos);
```

2.3 Contraintes

Les contraintes C2, C6 et C7 doivent être décrites à l'aide des dérivés de la classe *Constraint* et doivent être ajoutées au service concerné avec la méthode *addConstraint* de la classe Service.

2.3.1 ConstraintDaysSeq

Cette classe de contrainte concerne la contrainte C2. On ajoute ici la contrainte qui interdit la présence de 3 postes consécutifs qui ont l'attribut 0 (work) et on lui met une priorité de 30:

```
auto v = vector<int>();
v.push_back(0);
v.push_back(0);
v.push_back(0);
ghr->addConstraint(new ConstraintDaySeq(v, 30));
```

2.3.2 ConstraintInvolved

Cette classe de contrainte concerne la contrainte C6, on ajoute ici la contrainte qui oblige la présence de 2 postes qui ont l'attribut 5 (rest) après 2 postes qui ont l'attribut 0 (work) et on lui met une priorité de 20.
Le troisième paramètre permet d'indiquer le jour où le premier poste de la première séquence peut être détecté, par exemple si on veut que cette contrainte s'active uniquement à partir d'un samedi on peut mettre 5, ici -1 indique que la contrainte s'active pour tout les jours de la semaine.

```
auto v = vector<int>();
v.push_back(0);
v.push_back(0);
auto v2 = vector<int>();
v2.push_back(5);
v2.push_back(5);
ghr->addConstraint(new ConstraintInvolved(v, v2, -1, 20));
```

2.3.3 ConstraintSeqMinMax

Cette classe de contrainte concerne la contrainte C7, on ajoute ici la nécessité d'avoir 2 postes consécutifs qui ont l'attribut 0 (work) à partir d'un samedi et ce, au moins une fois par mois.

```
auto v = vector<int>();
v.push_back(0);
v.push_back(0);
ghr->addConstraint(new ConstraintSeqMinMax(5, MinMax::Min, 1 v, 10));
```

2.4 Agent

On crée les Agents et on les ajoute au Model en précisant le service auxquels ils appartiennent.

```
Agent* a1 = new Agent("1", nbHoursMonth, nbHoursWeek, status);
m.addAgent(a1,ghr);
```

On peut aussi préciser des affectations dans le planning de l'agent, ici on affecte à l'agent 1 le poste Jg pour quatrième jour du mois (les indices commencent à 0). Le dernier argument est un boolean permettant de bloquer cette affectation, si il est à *true* alors les algorithmes ne changeront pas cette valeur :

```
a1->setCalendarDay(jg, 3, true);
```

On peut aussi ajouter des postes impossibles pour l'agent (contraintre C4). Ici, on suppose l'existence du poste *ng* qui est un poste de nuit et son affectation à l'agent a1 est impossible:

```
auto ip = vector<Post*>();
ip.push_back(ng);
a1->setImpossiblePosts(ip);
```

2.5 Génération d'instances

Pour créer un Model il existe également un autre moyen. Au lieu de déclarer un modèle vide sur lequel on va ajouter nous même des attributs comme les services, les postes ou les agents, on dispose d'une fonction qui permet de faire tout cela automatiquement en se basant sur de l'aléatoire :

```
auto m_6 = Model::generateModelInstance(2, 30, 25, 6, 20, 70, 60.0, 155);
```

Dans l'ordre, les paramètres sont les suivants (les optionnels n'étant pas indiqués dans l'exemple) :

- $firstDay$ de type `Day`, représente le premier jour du mois du modèle
- $nbDays$ de type `int`, représente le nombre de jours du mois
- $overtime$ de type `float`, représente le nombre d'heures supplémentaires autorisées Δ
- $nbServices$ de type `int`, représente le nombre de services à créer
- $nbPosts$ de type `int`, représente le nombre de postes à créer
- $nbAgents$ de type `int`, représente le nombre de postes à créer
- $nbHoursWeek$ de type `float`, représente le nombres d'heures maximum autorisées par semaine

- *nbPosts* de type `int`, représente le nombre de postes total dans l'instance
- *nbAgentsPerService* de type `int` (optionnel), représente le nombre d'agents à créer par service
- *nbPostsPerService* de type `int` (optionnel), représente le nombre de postes à créer par service
- *proba_1er_conge* de type `int` (optionnel), représente la probabilité d'attribuer un congé à un agent sur un jour donné
- *proba_suite_conge* de type `int` (optionnel), représente la probabilité d'attribuer un congé à un agent après qu'il ai eu un congé la veille

On obtient donc dans cet exemple, un modèle basé sur un mois de 30 jours qui commence par un mercredi, d'un *overtime* possible de 25h; de 6 services, 20 postes, 70 agents créés aléatoirement, avec 60h de travail par semaine max et 155h de travail par mois max pour un agent (voir détails en 6.1.1 du rapport).

3 Résolution

Une fois le Model construit nous passons à la recherche d'une solution, pour cela nous avons plusieurs méthodes.

3.1 Méthode itérative

L'algorithme itératif s'utilise comme tel :

```
auto m2 = heuristicSolver::iterativeFast(m, nbIteration, distance, idPool);
```

La méthode renvoie un Model contenant la meilleur solution trouvée par l'algorithme. Le premier paramètre est le Model qu'on souhaite résoudre, le deuxième est le nombre d'itération que l'algorithme effectuera, le troisième est la distance maximal des voisins générés et le dernier est l'indice d'un service "Pool", les seuls échanges inter services seront fait avec ce service et un indice négatif désactive cette fonctionnalités.

3.1.1 Mars et Avril 2020

Les modèles de Mars et Avril 2020 sont déjà implémentés dans le code, voici comment les charger:

```
auto m = HandWrittedModels::generateMarch20();
auto m2 = HandWrittedModel::generateApril20();
```

3.2 Par programmation linéaire

L'algorithme utilisant la programmation linéaire est le suivant :

```
LPSolver::linearProgram(m);
```

La fonction prend en paramètre un modèle de base sur lequel le programme linéaire sera executé et renvoie le modèle solution avec les postes attribués aux agents en respectant les contraintes implémentées dedans. La fonction met quelques secondes à s'exécuter pour un modèle avec un service de taille 10 par exemple.

4 Utilitaire

4.1 Observer les résultats

Il y a plusieurs façon d'avoir accès aux résultats, la première est par la sortie par défaut de la fonction *cout*. On peut ainsi y afficher sur cette sortie le calendrier complet et/ou une évaluation de la solution qui indique le score et les contraintes brisées :

```
//Affiche le calendrier complet
m2.printPlanning();
//Affiche le score et les contraintes brisées
cout << m2.getValuation()->getScore() << endl;
cout << m2.getConstraintInformations() << endl;
```

Attention, pour l'affichage des contraintes les agents, les contraintes et les jours sont notés par leur indice, par exemple l'agent 1 correspond au deuxième agent ajouté au service, de même le jour 14 correspond au jour 15 sur le calendrier.

Il est aussi possible de générer le calendrier sous forme d'un fichier .xml (Excel, Libre Office...) :

```
m2.generateXMLPlanning("nomDeFichier.xml");
```

4.2 Sauvegarder et charger des instances de Model

Pour sauvegarder une instance :

```
m.generateXML("nomDuFichier.xml");
```

Pour charger une instance :

```
m.loadXML("nomDuFichier.xml");
```

Attention charger une instance remplacera toutes les informations de l'instance utilisé pour l'appel par les informations de celle chargée.