Devoir 2 - Résolution de problèmes combinatoires

Remise le 24 Mars sur Moodle.

Consignes

- Le devoir doit être fait par groupe de 2 au maximum. Il est fortement recommandé d'être 2.
- Lors de votre soumission sur Moodle, donnez votre fichier python `solver.py`, et vos 3 modèles Minizinc (`.mzn`) à la racine d'un seul dossier compressé (matricule1_matricule2_Devoir2.zip).
- Indiquez votre nom et matricule en en-tête des fichiers soumis.
- Toutes les consignes générales du cours (interdiction de plagiat, etc.) s'appliquent pour ce devoir.
- Il est permis (et encouragé) de discuter de vos pistes de solutions avec les autres groupes. Par contre, il est formellement interdit de reprendre le code d'un autre groupe ou de copier un code déjà existant (StackOverflow ou autre). Tout cas de plagiat sera sanctionné de la note minimale pour le devoir.

Conseils

Ce devoir est volontairement *challenging* pour obtenir la totalité des points. Voici quelques conseils pour le mener à bien :

1. Prenez vous y à l'avance. Il y a également la courbe d'apprentissage de Minizinc à considérer.
2. Travaillez efficacement en groupe, et répartissez vous bien les tâches.
3. Tirez le meilleur parti des séances de laboratoire encadrées afin de demander des conseils.
4. Pour la partie recherche locale, ne consacrez pas tout votre temps à l'amélioration de votre solution. Gardez cela pour la fin une fois que le reste est clôturé.
5. Pour la partie de modélisation, identifiez quelles contraintes globales pourraient vous aider.

Bonne chance !

1 Résolution par recherche locale (10 pts)

Le Québec aimerait développer son réseau ferroviaire. Il dispose d'un réseau de gares dites *principales* $(G_1, \ldots, G_n)$ et compte implémenter un nouveau réseau de gares *satellites* $(S_1, \ldots, S_m)$. Chaque gare satellite doit être connectée à une station principale. Cependant, pour pouvoir recevoir les trains des stations satellites, des travaux doivent d'être réalisés dans les gares principales. Si les travaux sont réalisés dans une gare principale, cette dernière est considérée comme *ouverte*. Ainsi, on a deux niveaux de décision à considérer :

1. Quelles gares principales va t-on ouvrir ?
2. Quelle gare principale va t-on assigner à chaque gare satellite ?

Connecter une station satellite j à une gare principale i a un coût $c_{i,j}$ (qui est en réalité la distance entre la gare principale et satellite) et l'ouverture d'une gare principale a un coût o_i. L'objectif est de minimiser les coûts totaux engendrés sur base de nos deux niveaux de décision (c-à-d la somme des coûts d'ouverture, si la gare est ouverte, et de tous les coûts de connexions). Il vous est demandé de résoudre

ce problème en utilisant un algorithme de recherche locale. Notez qu'il faut au minimum qu'une gare principale soit ouverte. Si aucune gare n'est ouverte la solution est infaisable, car il existe des gares satellites non desservies. Afin de vous aider dans votre tâche, trois instances, avec une indication du coût à battre, vous sont fournies.

- `instance_A_4_6` : une instance simple avec 4 gares principales et 6 satellites. Le coût optimal est de 187. C'est-à-dire qu'il s'agit du meilleur coût possible et qu'il est impossible de faire mieux.
- `instance_B_25_50` : une instance moyenne avec 25 gares principales et 50 satellites. Le coût à battre est de 1758.
- `instance_C_50_75` : une instance difficile avec 50 gares principales et 75 satellites. Le coût à battre est de 3383.

Notez que les valeurs des coûts sont arrondies dans cet énoncé. Référez vous au correcteur automatique pour obtenir le coût exact à battre.

Il vous est demandé de résoudre au mieux ce problème en utilisant un algorithme de recherche locale. Mis-à-part la première instance, il n'est pas demandé de trouver la solution optimale mais de vous en approcher le plus possible. Vous avez une très grande liberté pour résoudre ce problème, vous pouvez également aller plus loin que les algorithmes vus au cours. Le barème de cotation est le suivant.

- **La note minimale de 0 sur 10**, si votre code ne compile pas ou si la solution est incorrecte.
- **Entre 1 et 4**, si votre agent ne bat pas les performances de l'agent aléatoire.
- **Entre 5 et 8**, si votre agent bat l'agent aléatoire mais ne bat pas l'agent caché implémenté par les chargés de laboratoire. Le code de l'agent est caché mais on vous indique qu'il donne des solutions de coût 187, 1758 et 3383 pour les trois instances.
- **Entre 8 et 10**, si votre agent donne un meilleur score que l'agent caché, ou égal pour la première instance. La différence entre un 8 et 10 se fera sur la qualité de votre implémentation (présence de commentaires, code structuré, résolution intelligente, etc.). Pour aider à la notation, mettez dans un commentaire en haut de votre fichier, le ou les voisinages utilisés et les méthodes de sorties des minimums locaux si vous en avais implémenté.

La base du code est implémentée en Python et utilise seulement la librairie standard, dès lors aucun environnement Conda n'est fourni. Le temps d'exécution alloué pour votre algorithme de recherche sera de 2 minutes par instance. Le multithreading n'est pas autorisé. **ATTENTION**, vous devez vous assurer que votre code s'arrête AVANT les deux minutes.

Il vous est demandé de compléter la fonction `solver.py` avec votre propre algorithme. Vous ne devez pas changer les autres fichiers mais vous trouverez toutes les fonctions utiles dans le fichier `uflp.py`. L'output final est le suivant. Notez bien qu'un vérificateur de solutions est disponible.

```
$ python main.py --agent random --infile instance_A_4_6 --preview

************************************************************
[INFO] Solution obtained
[INFO] Execution time : 0.0 minutes
[INFO] Main stations opened : [1, 0, 1, 0] // les gares principales ouvertes (1) ou non (0)
[INFO] Satellite station association : [2, 2, 2, 0, 2, 2] // l'indice des gares principales a laquelle chaque gare satellite est connectee
[INFO] Penality obtained (value to minimize) : 349.53776741200886 // le score a minimiser
[INFO] Sanity check passed : True // True si la solution passe les tests de verification
************************************************************
```

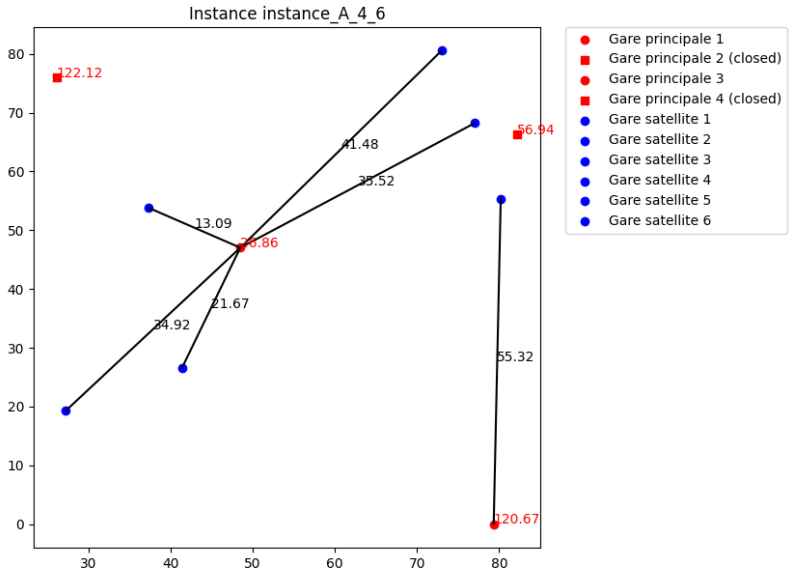

FIGURE 1 – Preview retournée à la fin de l'exécution

Une visualisation d'une solution pour l'instance simple est proposée à la Figure 1. Dans cette dernière, les gardes principales 1 et 3 sont ouvertes, et le traits indiquent les liaisons faites pour les gares satellites. Vous pouvez obtenir cette visualisation pour vos solutions en utilisant l'argument `--preview`. L'argument `--agent` peut être suivi de l'argument l'option `random` ou `advanced` (votre agent) et les 3 instances sont `instance_A_4_6`, `instance_B_25_50` et `instance_C_50_75`.

Le fichier `autograder.py` peut vous aider à vous donner une idée de votre note finale et vérifier que tout fonctionne correctement. Pour aider à la correction, commentez votre code. Vous pouvez décrire en une phrase, les différentes briques de votre algorithme (initialisation, voisinage, sélection, etc...) et le ou les mécanismes pour échapper aux minimas locaux que vous avez mis en place.

2 Modélisation en programmation par contraintes (10 pts)

Si ce n'est pas déjà fait, vous aurez besoin de télécharger MiniZinc et son environnement de développement. Il est recommandé de travailler avec la version **2.8.3** de MiniZinc. Un tutoriel se trouve également sur ce lien, vous pourrez passer au travers avant d'attaquer les exercices.

Pour ce deuxième devoir, il vous est demandé de modéliser différents problèmes, de difficulté croissante, sur MiniZinc. Pour chaque exercice, un fichier de base (`.mzn`), les fichiers vérificateurs de solutions (`.mzc`), ainsi que les différentes configurations à résoudre (`.dzn`), vous seront fournis. Vous êtes fortement encouragés à utiliser l'API de MiniZinc et d'identifier les contraintes globales les plus pertinentes pour résoudre les problèmes.

Lorsque vous lancez votre modèle dans MiniZinc, avoir le fichier `.mzc` correspondant ouvert dans les onglets vous permettra d'obtenir une vérification des solutions en sortie afin de vous indiquer si vos contraintes et la fonction objectif sont bien respectées. Le message **CORRECT** devrait s'afficher lorsque la solution renvoyée par votre modèle respecte les consignes. Attention, les vérificateur ont besoin d'avoir une fonction objectif définie pour fonctionner, il est donc probable que les premiers lancement de vos modèles se fassent sans les utiliser (redémarrer Minizinc s'ils ont déjà été ouverts dans les onglets).

Les fonctions d'output des modèles Minizinc (`.mzn`) ont été implémentées pour vous, vous n'avez donc pas à les modifier. Veuillez ne pas modifier les variables déjà données au risque de ne plus pouvoir utiliser les vérificateurs, si vous avez besoin de variables supplémentaires il vous suffit de les déclarer à la suite. Ajoutez des commentaires lorsque vous jugez que c'est nécessaire. Cela peut aider le correcteur à comprendre une partie de votre modèle si ce dernier est incorrect. Vos modèles devraient être capable de résoudre chaque configuration en **moins de 2 minutes** sur une configuration similaire aux ordinateurs de la salle de travaux pratiques.

Problème 0 : code à 4 chiffres (introductory level - 0 pt)

On vous demande de trouver un **nombre** composé de 4 **chiffres** qui satisfait les critères suivants :

1. C'est un nombre pair.
2. Le chiffre 0 n'est pas présent dans le nombre.
3. Les 4 chiffres sont différents.
4. Le chiffre à la position des milliers est supérieur à celui à la position des centaines.
5. Le chiffre à la position des dizaines est inférieur à celui à la position des unités.
6. Le chiffre à la position des centaines est supérieur à celui à la position des unités.
7. La somme des 4 chiffres est supérieure à 15.
8. Le produit des 3 derniers chiffres (chiffre à la position des centaines$\times$chiffre à la position des dizaines $\times$ chiffre à la position des unités) doit être minimisé.

Implémentez un modèle générique pour résoudre ce problème. Cet exercice est **non côté** et a juste pour objectif de vous familiariser avec MiniZinc. Vous ne devez pas l'intégrer dans votre soumission.

Problème 1 : optimisation des plages radio (easy level - 3 pts)

Le gouvernement de la nouvelle république a besoin de votre expertise en modélisation pour améliorer les communications radio sur la planète Coruscant. Cependant, le budget alloué à l'installation radio

est limité. Vous devez donc trouver la plage de fréquences la plus courte possible qui peut regrouper les numéros de m fréquences différentes demandées. Chaque fréquence est représentée par un numéro entier, qui doit être placé dans un ordre croissant sur la plage. La position d'une fréquence sur la plage est déterminée par un numéro entier (≥ 0), la première fréquence commençant toujours à 0, avec la dernière valeur indiquant également la longueur totale de la plage. De plus, il est essentiel que toutes les fréquences soient réparties à des distances différentes les unes des autres sur la plage afin d'éviter les interférences. Vous pouvez vous référer à la Figure 2 pour mieux comprendre le problème, qui est illustré ici pour $m = 4$. Vous pouvez remarquer que chaque paires de numéros de fréquence possède une distance unique (e.g., les fréquences 0 et 4 sont à une distance de 4, ce qui est différent de 1 et 6 qui sont à une distance de 5)

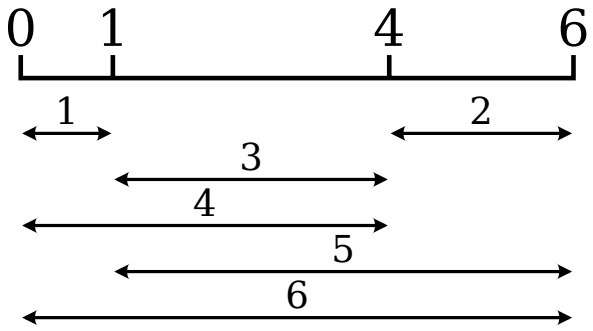

FIGURE 2 – Exemple de plage réalisable de 4 fréquences.

Implémentez un modèle générique pour résoudre ce problème. Un bon modèle modèle doit contenir au moins une contrainte globale. Trois configurations vous sont données dans des fichiers `.dzn`. Trouver la **solution optimale** en moins de 2 minutes avec le solveur `Gecode` pour chaque configuration vous rapporte 1 point. Une configuration d'exemple avec $m = 4$ vous est également fournie, vous devriez trouver la plage `[0, 1, 4, 6]` et une longueur égale à 6.

Problème 2 : les tours de communication (normal level - 3 pts)

Suite à la mise en place des nouvelles fréquences, vous êtes à présent mandatés pour installer les tours de relais dans plusieurs villes. Les plans des villes sont représentés par des terrains carrés de taille $n \times n$ sur lesquels il faudra placer des tours de deux types distincts (bleu et rouge). Un équilibre parfait entre les types de tours doit être maintenu, pour cela elles doivent être de nombre égal sur le terrain. De plus, les signaux envoyés par les tours ne doivent pas se croiser si elles sont d'un type différent, les tours communiquent de façons horizontales, verticales et en diagonales. Les illustrations suivantes présentent 3 situations.

Afin d'assurer l'efficacité des communications, il vous faudra maximiser le nombre total de tours de chaque type (i.e. un terrain composé de 5 tours bleues et de 5 tours rouges donne un score de 5).

Vous pouvez facilement remarquer qu'intervertir les groupes de tours rouges et bleues donne les mêmes résultats pour notre fonction objectif. De même, si vous appliquez une rotation de 90° au terrain, vous vous trouverez dans un état équivalent à celui avant votre rotation. Votre solveur risque de parcourir de nombreux états équivalents à ceux qu'il a déjà rencontré. En programmation par contraintes, on appelle cela des *états symétriques*. Une bonne pratique est de rajouter au modèle des contraintes (aussi appelées *contraintes symétriques*) pour briser les symétries et réduire l'espace de recherche. Pour cette question, on vous demande d'implémenter au moins une contrainte permettant de briser une symétrie du problème.

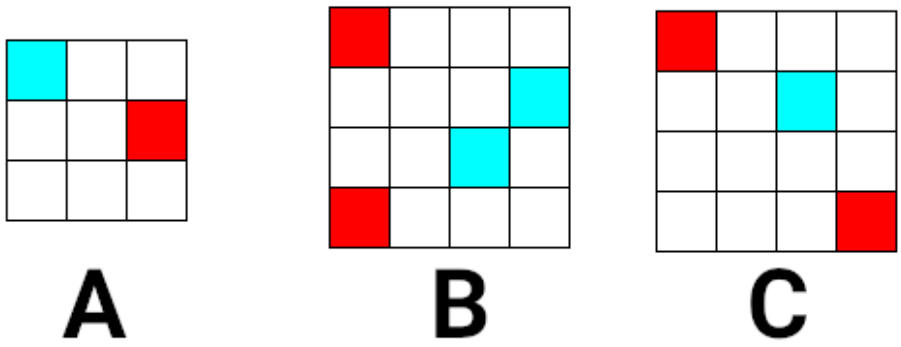

FIGURE 3 – Illustration de trois placements de tours pour différents terrains. La solution A est réalisable car elle respecte toutes les contraintes (même nombre de tours, et aucun croisement horizontal, vertical, et diagonal). La solution B est irréalisable car deux tours de couleur différente se croisent sur la diagonale. La solution C est aussi irréalisable car il y a plus de tours rouges que de tours bleues.

Implémentez un modèle générique pour résoudre ce problème. Votre modèle devra contenir au moins une contrainte globale. Trois configurations vous sont données dans des fichiers `.dzn`. Trouver une **solution optimale** en moins de 2 minutes avec le solveur `Gecode` pour chaque configuration vous rapporte 1 point.

Problème 3 : le tournoi d'ouverture (hard level - 4 pts)

Pour fêter en grande pompe le développement de la nouvelle planète, un grand tournoi de rugby est organisé. Vous serez chargé de modéliser un calendrier pour les 8 équipes qui doivent se rencontrer durant les 16 jours de compétition.

La compétition doit respecter les règles suivantes :

1. Pour effectuer une rencontre, deux équipe doivent être présente sur le même terrain.
2. Un match doit avoir lieu sur le terrain désigné comme le domicile de l'une des deux équipes.
3. Un terrain ne peut accueillir plus de deux équipes.
4. Si une équipe ne joue pas lors d'une journée, elle est placée sur le terrain n°0.
5. Une équipe ne doit pas jouer trop souvent à domicile, elle n'est donc pas autorisée à jouer plus de 7 jours sur son propre terrain.
6. Les équipes doivent jouer deux fois l'une contre l'autre.
7. Les deux rencontres de deux équipes doivent être espacées de 3 jours au minimum. Par exemple, si A joue contre B le jour 2, elles ne pourront pas rejouer ensemble avant le jour 6.
8. Les deux rencontres de deux équipes doivent avoir lieu sur des terrains différents.

Enfin, on souhaite maximiser le nombre de fois où deux matchs entre les mêmes équipes sont espacés par aux moins 6 jours. Par exemple, l'équipe A rencontre l'équipe B lors du jour 5, il faudra alors que leur deuxième rencontre arrive lors du jour 12 ou plus pour être comptabilisée.

Implémentez un modèle pour résoudre ce problème. Une seule configuration vous est donnée. Un modèle qui trouve une **solution optimale** maximisant notre objectif en moins de 2 minutes avec le solveur `Chuffed` vous assure la totalité des points. Pour vous aider, la valeur objectif à atteindre est de 28.

Vos missions sont maintenant achevées !