TP1 CSP - Modélisation

Nous avons réaliser le TP1 chacun de notre coté ainsi c'est pour cela que dans le fichier .zip il y a 2 projets TP1 avec chacun notre nom dessus.

Cependant nous avons trouvé les mêmes solutions mais un de nous deux a trouvé une autre solution possible pour modéliser le problème des n reines. C'est pour cette raison que nous avons voulu séparer nos deux TP1.

A. Configurer son environnement de travail

B. Modéliser le problème du Zèbre en extension

Notre **première solution** était :

1 - Yellow = Norwegian = Water = Fox = Kool

2 - Blue = Ukrainian = Tea = Horse = Chesterfield

3 - Red = English = Milk = Snail = Old Gold

4 - Ivory = Spanish = Orange Juice = Dog = Lucky Strike

5 - Green = Japanese = Coffee = Zebra = Parliament

C'est une **solution correcte** au vue de toutes les contraintes données.

Puis nous avons calculer toutes les solutions et **on en a pas d'autre** alors la seule solution possible est celle trouvé au dessus.

C. Modéliser le problème du Zèbre en intension

Après avoir remplacé les couples de valeurs possibles par des calculs arithmétiques on se retrouve avec la même et seule solution du problème.

D. Modéliser le problème des n-reines

Comme nous l'avons dit plus haut on a chacun fait son propre TP, ainsi on a deux dossier différents pour chaque TP d echaun. Parmis eux il y a une autre modélisation qu'on aurait pu faire pour le problème des nReines.

Ainsi nous allons expliquer les deux possibilités traités :

- **Solution 1 :** La première solution est la solution que le sujet nous explique et que la professeur de TD nous a indiquer de faire.

En considérant qu'on prend le problème des 5 reines (comme le sujet l'indiquait), on donne la modélisation sur la figure 1 à la page 2on prend 5 variables qui sont les reines (Ri) positionnées aux lignes i et qui peuvent ce deplacer que de 5 cases de droite à gauche.

Ainsi, on a reines[i] = "numero de colonne de la où est positionné Ri".

En prenant donc l'exemple sur la figure 2 à la page 2, on a chaque reine Ri positionné sur sa propre colonne i et les indices de colonnes en haut. On précise que dan le tableau de reines les indices sont de 0 à 4 mais que ici on prend l'interval [1,5] pour que ce soit plus facile à formuler.

Alors, reines[1] = 2 pour la reine rouge (R2) positionné à la ligne d'indice 1, dans le tableau de reines. Et reines[3] = 4, pour la reine bleu (R4) positionnée à la ligne d'indice 3, dans le tableau.

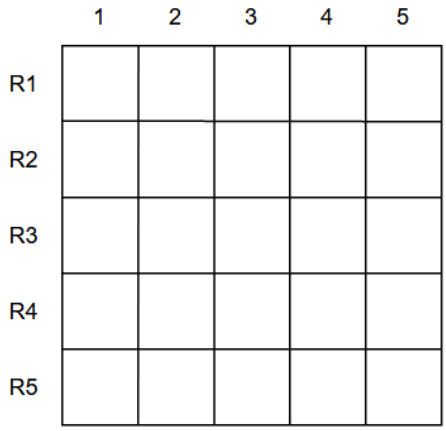

Figure 1: Plan d'un échiquier de 5 reines pour modéliser la solution 1

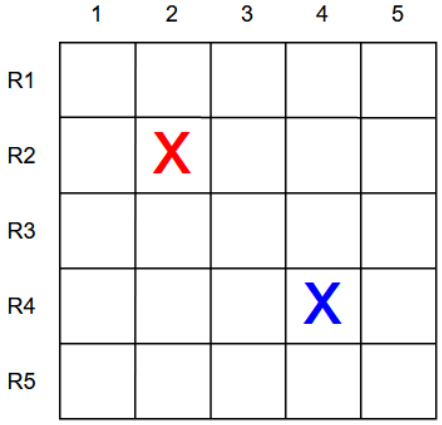

Figure 2: Plan d'un échiquier de 5 reines avec deux exemples de reines

Pour cette solution on a modélise le problème CSP des 5 reines en fixant les contraintes en extension (en donnant chaque valeur possible ou non) et en intension (fixer des operation arithmétique pour calculer les contraintes).

> ***Extension :***

On peut donc voir dans les fichiers joints toutes les valeurs possibles ou non possibles en extension des différentes contraites. Voyons pour les 3 catégories de contraintes des exemples :

* Pas la même ligne : Comme on suppose déjà que chaque reine i reste sur la ligne i on a pas besoin de mettre de contraintes.
* Pas la même colonne : Les tuples {t,t} pour des reines i et j différentes sont des tuples interdits (avec $1 < t < 5$).
* Pas les mêmes diagonales : Nous avons également réalisé tous le tuples interdits pour les diagonales. Par exemple les tuples $\{1, 25\}$ et $\{4, 16\}$ sont interdits car ils sont sur les mêmes diagonales.

  Pour faire la liste, nous avons procédé en commencant par la grande diagonales du millieu vers la droite, puis on a lister les diagonales d'en dessous, puis nous sommes revenu sur celles d'au-dessus. Puis nous avons recommencé avec les diagonales vers la gauche.

> ***Intension :***

En intension nous pouvons voir les opérations arithmétiques choisis pour modéliser les contraintes :

* Pas la même ligne : De même qu'en extension.
* Pas la même colonne : Les valeurs des reines entre elles ne doivent pas être égales car les valeurs correspondent aux colonnes des reines, soit

$$reines[i] \neq reines[j]$$

  .

* Pas les mêmes diagonales : La différence entre les indices i et j ne doit pas être égale à la différence des valeurs des reines aux indices i et j, soit,

$$|reines[i] - reines[j]| \neq |i - j|$$

  .

- **Solution 2 :** La seconde solution est une solution qui se rapproche à l'exemple donné dans le sujet.

En considérant qu'on prend toujours le problème des 5 reines, on donne la modélisation sur la figure 3 à la page 3. On prend toujours 5 variables qui sont les reines (Ri) positionnées aux lignes i et qui peuvent ce deplacer sur la totalité de l'échiquier, donc sur les 25 cases.

On choisis alors de donner comme valeurs le numéro de la case sur laquelle la reine i est, soit reines[i] = "numero de la case de la où est positionné Ri".

En prenant donc l'exemple sur la figure 4 à la page 3, on a chaque reines Ri positionné sur sa propre colonne i et les numéro de case. De même que tout a l'heure le tableau des reines à pour indice 0,1,2,3 et 4 alors que sur l'échiquier on considère que c'est 1,2,3,4 et 5.

Alors, reines[1] = 7 pour la reine rouge (R2) positionné à la ligne d'indice 1, dans le tableau de reines. Et reines[3] = 19, pour la reine bleu (R4) positionnée à la ligne d'indice 3, dans le tableau.

	1	2	3	4	5
R1	1	2	3	4	5
R2	6	7	8	9	10
R3	11	12	13	14	15
R4	16	17	18	19	20
R5	21	22	23	24	25

Figure 3: Plan d'un échiquier de 5 reines pour modéliser la solution 2

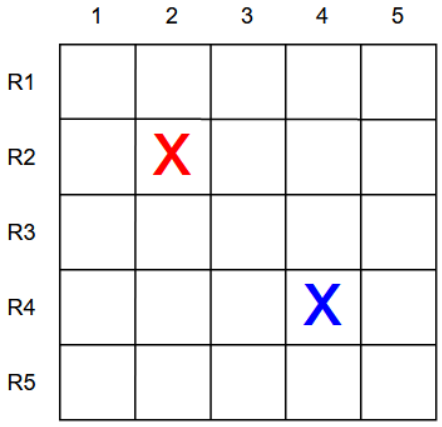

Figure 4: Plan d'un échiquier de 5 reines avec deux exemples de reines

Comme tout à l'heure nous avons fixer les contraintes en Extension puis en Intension.

> ***Extension :***

On peut donc voir dans les fichiers joints toutes les valeurs possibles ou non possibles en extension des différentes contraites. Voyons pour les 3 catégories de contraintes des exemples :

* Pas la même ligne : Cette fois-ci on doit fixer que la reines 2 ne peut pas être sur la ligne de la reines 1 donc ne peut pas prendre les valeurs de 1 à 5. Ainsi, au lieu de fixer tous les tuples interdits, on a fixé toutes les valeurs valides pour chaque reine. Alors par exemple, la reine 1 peut prendre 1,2,3,4 ou 5 comme valeur. On fait donc de même pour toutes les autres reines.
* Pas la même colonne : Pour prendre un exemple, les tuples $\{6, 11\}$ et $\{4, 24\}$ sont interdits car ce sont des numéros de cases qui sont sur les mêmes colonnes.
* Pas les mêmes diagonales : Par exemple, les couples {1,2} et {1,4} sont interdits respectivements pour les reines i et j tel que $|i - j| = 2 - 1 = 1$ et pour les reines i et j tel que $|i - j| = 4 - 1 = 3$.

  Ainsi, on doit alors créer pour chaque couple de reine {i,j}, selon la differences de leurs indices, des tuples interdits. C'est pourquoi dans notre fichier nous mettons 4 ensembles de tuples pour les 4 cas possibles. Sinon cela ne mène à aucune solution sachant qu'on fixerai des tuples interdits qui ne le sont pas.

> ***Intension :***

En intension nous pouvons voir les opérations arithmétiques choisis pour modéliser les contraintes :

* Pas la même ligne : On indique pour chaque reine son domaine en donnant des bornes, soit

$$1 + i * 5 < reines[i] < 5 + i * 5$$

. Cependant, dans le fichier nous avons mis des bornes en brut.

* Pas la même colonne : Pour les colonnes nous avons construit une nouvelle boucle pour passer d'une ligne à l'autre, elle commence donc à 5 et augmente de 5 en 5 jusqu'à 20, soit

$$reines[j] \neq reine[i] + k$$

avec k qui est l'indice de la 3ème boucle expliqué plus haut. Comme on parcours i puis j qui commence à i+1 alors j est forcément plus grand que i. Si on prend un exemple, $reine[j] = 18$ alors on peut pas avoir $reines[i] = 13$ si $k = 5$, ou alors $reines[i] = 3$ si $k = 15$.

* Pas les mêmes diagonales : Pour les diagonales, c'est encore un peu plus compliqué. Au lieu de mettre une $3^{\text{ème}}$ boucles, on met 2 compteurs :
  - t = 4 et augmente de 4 en 4 : pour représenter les diagonales en bas à gauche
  - k = 6 et augmente de 6 en 6 : pour les diagonales en bas à droite.

  En procédant de la même manière que la boucle d'avant on regarde si la reines à la ligne j n'est pas sur la diagonale de la reine à la ligne i dans les deux sens. On ne pouvait pas mettre 2 nouvelles boucles car selon interdirait des valeurs qui ne doivent pas l'être. Ainsi, cela donne ces deux contraintes :

$$reines[j] \neq reine[i] + k$$

$$reines[j] \neq reine[i] + t$$

  Prenons un exemple : pour $reines[j] = 11$ alors on doit avoir $reines[i] \neq 23$ pour $k = 6 * 2 = 12$ ou pour $reines[j] = 14$ alors on doit avoir $reines[i] \neq 18$ pour $k = 4 * 1 = 4$.

On voit donc que cette solution 2 est beaucoup plus compliqué que la solution 1, car il y a beeaucoup plus de valeur en jeu. Et donc les contraintes sont plus compliqué à modéliser.

De plus, cette solution n'est pas pratique lorsqu'on veut changer le n. On a du créer tous les fichiers pour chaque n car les boucles et compteurs sont réalisé en dur, et ne sont adapté que pour n=5. Il faudrait donc adapter le code en fonctiond e chaque n.

- **Conclusion :**

Quelque soit la solution, nous trouvons bien les mêmes valeurs, ainsi nous pouvons dire que c'est deux solutions repondent correctement au problème. Ainsi en regardant les différents n proposé nous voyons que le nombre de solution possible est **exponentiel** en fonction du n :

- n = 1 : Auncune solution
- n = 2 : Auncune solution
- n = 3 : Auncune solution
- n = 4 : 2 solutions
- n = 8 : 92 solutions
- n = 12 : 14 200 solutions
- n = 16 : nombre solution >> 14 200

PROJET CSP – Génération de benchmark et évaluation de méthode

L'objectif de ce projet est de comprendre la méthodologie d'expérimentation et d'évaluation d'algorithmes de résolution de problèmes. Le travail demandé comporte 3 parties :

1. Se construire un jeu d'essai pertinent permettant une réelle évaluation des performances des algorithmes de résolution.
2. Évaluer sur ce jeu les performances d'un algorithme de résolution.
3. Éventuellement, proposer quelques modifications de l'algorithme par défaut et évaluer comparativement les bénéfices/pertes de ces modifications.

Vous rendrez au plus tard le 14/11 à minuit sur le devoir Moodle prévu pour ce projet une archive au format zip contenant :

- un **rapport correctement rédigé au format PDF** détaillant très clairement le travail réalisé et donnant tous les éléments permettant de refaire vos évaluations (paramètres de génération des réseaux, les exécutions à faire...)
- l'ensemble de vos **sources Java** (de ce projet et du TP Modélisation)
- les **scripts bash de génération** des jeux d'essais (NE PAS RENDRE LES JEUX D'ESSAIS qui peuvent prendre beaucoup de place).

Ce travail peut être fait en binôme mais chaque membre du binôme pourra être interrogé individuellement sur le travail réalisé !

A. Materiel fourni

Fichiers fournies :

- un fichier bench.txt contenant dans un format texte ad-hoc un jeu d'essai de 3 réseaux de contraintes binaires en extension ;
- une classe java Expe.java contenant une méthode de lecture de tels fichiers texte de réseaux de contraintes ;
- un programme C urbcsp.c implémentant un générateur aléatoire de réseaux de contraintes binaires en extension.

Le générateur urbcsp va vous permettre de tester vos algorithmes sur des instances plus ou moins difficiles de réseaux de contraintes binaires. C'est un programme en C adapté du uniform random binary CSP generator (by D.Frost, C. Bessiere, R. Dechter, and J.C. Régin). Tous les domaines sont de même taille et toutes les contraintes ont le même nombre de tuples.

Pour l'utiliser, il vous faudra d'abord compiler le source : gcc -o urbcsp urbcsp.c Puis lancer la génération de réseaux en spécifiant les 5 paramètres de génération et en ajoutant une redirection vers un fichier : ./urbcsp nbVariables tailleDomaine nbConstraints nbTuples nbRes ¿ fic.txt

Exemple : ./urbcsp 10 15 10 30 3 ¿ bench.txt est la commande qui a permis de générer le fichier bench.txt de 3 réseaux de 10 variables et 10 contraintes. Il y a 15 valeurs dans le domaine de chaque variable et 30 tuples dans chaque contrainte.

Le fichier java Expe.java doit être importer dans votre projet. Il contient une méthode lireReseau qui lit un réseau au format de sortie de urbcsp et le charge comme un Model Choco. Un exemple de main lisant les 3 réseaux du fichier bench.txt et affichant les réseaux lus est également fourni. Attention le fichier bench.txt doit être mis à la racine de votre projet Java tp-ia-choco.

Travail à faire :

1. Récupérer les différents fichiers
2. Importer le fichier Expe.java dans votre projet
3. Mettre le fichiers bench.txt à la racine de votre projet
4. Exécuter le programme Expe.java et observer l'affichage produit
5. Compiler le programme urbcsp.c Université de Montpellier Intelligence Artificielle Master Info
6. Générer un fichier benchSatisf.txt de 3 réseaux de 10 variables et 10 contraintes. Les domaines ont 15 valeurs et le nombre de tuples par contraintes est de 80.
7. Générer un fichier benchInsat.txt de 3 réseaux de 10 variables et 15 contraintes. Les domaines ont 15 valeurs et le nombre de tuples par contraintes est de 20.
8. Modifier le programme Expe.java pour qu'il calcule pour chacun des 2 nouveaux jeux d'essai le nombre de réseaux qui ont une solution. Qu'observez-vous ?

1) On est aller récupéré les fichiers fournies.
2) On a créer un projet qui s'appelle : *Projet-ia-choco*. Puis on a importer *Expe.java*.
3) On a également mis le ficheir .txt à la racine du projet c'est à dire dans le même dossier que le *pom.xml*.
4) On voit qu'il n'y a pas de solution possible avec ce premier fichier.

5) On compile et exécute le fichier .c qui nous permet d'obtenir des fichier .txt celon les paramètres que l'on veut.

6) Ainsi on commence par un fichier *benchSatisf.txt* tel que :

```
$ gcc urbcsp.c -o generateur
$ ./generateur 10 15 10 80 3 > benchSatisf.txt
```

Ainsi ces deux lignes de commande ont créé un fichier *benchSatisf.txt*.

7) De même pour *benchInsat.txt* :

```
$ gcc urbcsp.c -o generateur
$ ./generateur 10 15 15 20 3 > benchInsat.txt
```

8) On modifie donc le code pour obtenir que aucun des 3 réseaux pour les 3 fichiers n'a de solutions.

B. Construction d'un jeu d'essais conséquent et identification de la transition de phase

L'objectif de cette partie est de se doter d'un jeu d'essais suffisamment gros pour " voir quelque chose " mais pas trop gros pour ne pas avoir à attendre trop longtemps la résolution. On s'attend à ce que vous fixiez un nombre de variables, un nombre de contraintes (cela fixera la densité du réseau), et une taille de domaines. Puis que vous fassiez varier le nombre de tuples pour balayer la zone des réseaux très satisfiables aux réseaux très insatisfiables. Pour chaque dureté retenu, vous générerez au moins 10 réseaux (mais si possible plus) et calculerez le % de réseaux ayant au moins une solution.

Il vous faudra donc lancer la génération de plusieurs fichiers de dureté croissante. Vous pourrez vous aider d'un script bash du type :

```
for i in 10 30 50 80 110
do
    ./urbcsp 10 15 10 $i 3 > csp$i.txt
done
```

Par ailleurs, il vous faudra modifier le programme Expe pour qu'il puisse calculer pour chaque niveau de dureté le % de réseaux ayant au moins une solution. Vous tracerez alors la courbe % de réseau ayant au moins une solution en fonction de la dureté. Le mieux est sans doute que votre programme Expe génére un fichier CSV dureté;% que vous exploiterez avec Libre Office pour dessiner la courbe (cf. annexe). Votre objectif ici est de mettre en évidence une transition de phase.

Travail à faire :

1. Modifier son programme Expe.java pour qu'il puisse calculer la fonction
2. Réaliser un script de construction d'un benchmark
3. Procéder à plusieurs essais pour identifier des paramètres intéressants de génération permettant de bien mettre en évidence une transition de phase. On attend de vous que vous produisiez différents jeux d'essais en particulier pour différentes densités de réseau.
4. Expliquer clairement les paramètres des jeux d'essais produits.
5. Dessiner les courbes

C. Évaluation de la méthode de résolution par défaut de Choco sur vos jeux d'essais

L'objectif de cette partie est de mesurer expérimentalement le comportement de Choco sur vos jeux d'essais. On s'attachera à produire différentes mesures d'évaluation de ce comportement : temps de calcul, taille de l'arbre développé... Référez-vous au document " Méthodologie d'évaluation des méthodes ".

Vous aurez certainement besoin de mettre en place un mécanisme de Time-Out. Pour cela, vous pourrez vous définir une limite de temps grâce à la fonction solver.limitTime("10s") et tester en sortie si le solver s'est arrêté par qu'il a trouvé une solution, parce qu'il a atteint la limite imparti de temps ou parce qu'il a trouvé qu'il n'y avait pas de solution à l'aide de la séquence :

```
if(solver.solve()){
    // do something, e.g. print out variable values
}else if(solver.hasReachedLimit()){
    System.out.println("The solver could not find a solution nor
    prove that none exists in the given limits");
}else {
    System.out.println("The solver has proved the problem has no
    solution");
}
```

Vous expliquerez bien votre méthodologie d'évaluation (nombre d'exécutions, gestion des time-out...) et tracerez des courbes mesure d'évaluation/dureté et ferez une analyse des résultats obtenus.

Travail à faire :

1. Ajouter à votre programme Expe.java un mécanisme de Time-Out (pour ne pas attendre trop longtemps les exécutions).
2. Modifier son programme Expe.java pour qu'il puisse calculer vos mesures d'évaluation/dureté.
3. Tracer les courbes de ces mesures
4. Expliquer la méthodologie d'évaluation mise en place
5. Analyser les résultats obtenus.

D. Modification du solveur par défaut

Dans cette dernière partie, optionnelle, il vous est demandé de modifier le comportement par défaut du solver, par exemple en jouant sur l'heuristique de choix des variables. En exploitant la documentation Choco vous verrez qu'il est assez facile de programmer (ou simplement de sélectionner) une heuristique spécifique.
L'objectif est d'évaluer l'influence du choix d'une heuristique sur les performances de la résolution. Pour ça, il faut relancer votre évaluation sur les même jeux d'essais avec votre solver modifié.
Puis mettre sur un même graphique les deux résultats d'évaluation afin d'observer si l'une des heuristiques est meilleure qu'une autre.

Travail à faire :

1. Prévoir un système de choix permettant de lancer votre évaluation sur un solveur paramétré.
2. Lancer les évaluations avec les différents choix de solver.
3. Tracer les courbes.
4. Faire une analyse comparative des résultats.