TP1 CSP - Modélisation

Nous avons réalisé le TP1 chacun de notre coté puis fait le projet ensemble. Par ailleurs, 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 2 dossiers sur le problème des nReines.

Nous avons fourni une annexe pour savoir dans quels dossiers se trouve les éléments du projet.

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 vu de toutes les contraintes données.

Puis nous avons calculé toutes les solutions et **on n'en a pas d'autre** alors la seule solution possible, est celle trouvée ci-dessus.

C. Modéliser le problème du Zèbre en intention

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, nous avons chacun fait notre propre TP, ainsi nous avons deux dossiers différents pour chaque TP. Parmi eux il y a une autre modélisation que nous aurions pu faire pour le problème des nReines.

Ainsi nous allons expliquer les deux possibilités traitées :

- **Modélisation 1 :** La première modélisation est celle montrée dans le sujet.

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 2 ; on prend 5 variables qui sont les reines (Ri) positionnée aux lignes i et qui peuvent se déplacer que de 5 cases de droite à gauche.

Ainsi, on a reines[i] = ¡numéro de colonne de là où est positionné Ri¿.

En prenant donc l'exemple sur la figure 2 à la page 2, on a chaque reine Ri positionnée sur sa propre colonne i, et les indices de colonnes en haut. On considère donc l'intervalle [1;5] pour les indices.

Sur la figure 2, on voit donc que la reine R2 (rouge) est positionnée à la ligne 2, et la reine R4 (bleue) est positionnée à la ligne 4.

Pour cette solution on a modélisé le problème CSP des 5 reines en fixant les contraintes en extension (en donnant chaque valeur possible ou non) et en intention (fixer des opérations arithmétiques pour calculer les contraintes).

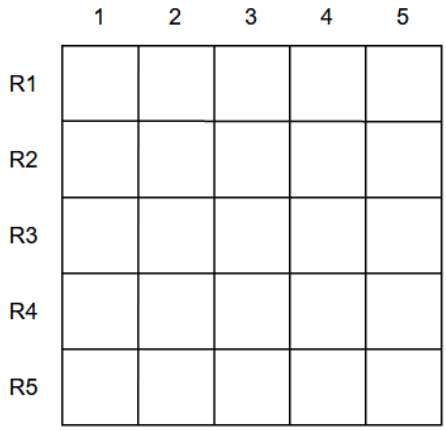

Figure 1: Plan d'un échiquier de 5 reines pour modéliser la solution 1

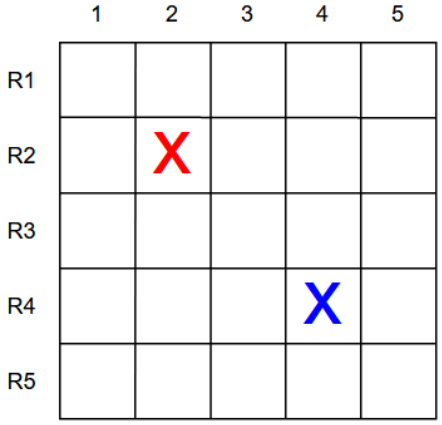

Figure 2: Plan d'un échiquier de 5 reines avec deux exemples de reines

> ***Extension :***

On peut donc voir dans les fichiers joints toutes les valeurs possibles ou non possibles en extension des différentes contraintes. 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 n'a pas besoin de mettre des 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 les 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 commençant par la grande diagonale du milieu vers la droite, puis on a listé les diagonales d'en dessous, puis nous sommes revenus sur celles d'au-dessus. Puis nous avons recommencé avec les diagonales vers la gauche.

> ***Intension :***

En intention nous pouvons voir les opérations arithmétiques choisies 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 dois pas être égal à la différence des valeurs des reines aux indices i et j, soit,

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

  .

- **Modélisation 2 :** La seconde modélisation 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 se déplacer 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] = "numéro 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 reine Ri positionnée sur sa propre colonne i et les numéros 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 reine. Et reines[3] = 19, pour la reine bleue (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

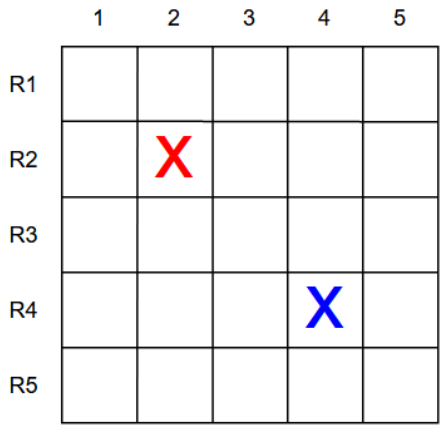

Figure 4: Plan d'un échiquier de 5 reines avec deux exemples de reines

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 contraintes. Voyons pour les 3 catégories de contraintes des exemples :

* Pas la même ligne : Cette fois-ci on doit fixer que la reine 2 ne peut pas être sur la ligne de la reine 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 respectivement 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 différence 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 fixerait des tuples interdits qui ne le sont pas.

> ***Intension :***

En intention nous pouvons voir les opérations arithmétiques choisies 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 parcourt i, puis j commençant à i+1, alors j est forcément plus grand que i. Si on prend un exemple, $reine[j] = 18$ alors on ne 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 reine 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 interdiraient 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 beaucoup plus de valeur en jeu. Et donc les contraintes sont plus compliquées à modéliser.

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

- **Conclusion :**

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

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

A. Materiel fourni

Fichiers fournis :

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 ?

Après modification du code pour parser les trois fichiers, on voit que tous les modèles de benchSatisf ont une solution, tandis que tous ceux de benchInsat n'ont aucune solution.

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

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

1. On ajoute un compteur avant de lire chaque réseau, on l'incrémente à chaque fois qu'un réseau trouve une solution, et on compare ce compteur au nombre total de réseau.
2. On réalise un script en bash pour générer les différents benchmarks

```
for ((i=211; i>178; i-=3))
do
./urbcsp 35 17 249 $i 50 > "../TP_Choco/reseaux/bench1/csp_$i.txt"
done
```

3. En prenant différentes densités de réseau avant de faire varier la dureté, on remarque que plus la densité augmente, plus le pourcentage de réseaux ayant une solution diminue, mais les temps d'exécution semblent être les plus longs pour des duretés variant avec une densité fixée aux alentours de 50%.
4. Pour la suite, on fixe donc la densité à 50%, et on va uniquement faire varier la dureté des réseaux en faisant varier le nombre de tuples autorisés de 211 à 181.
5. On peut donc représenter sur une courbe l'évolution du pourcentage de réseaux qui ont au moins une solution en fonction de la dureté des réseaux (voir figure 5 page 6)

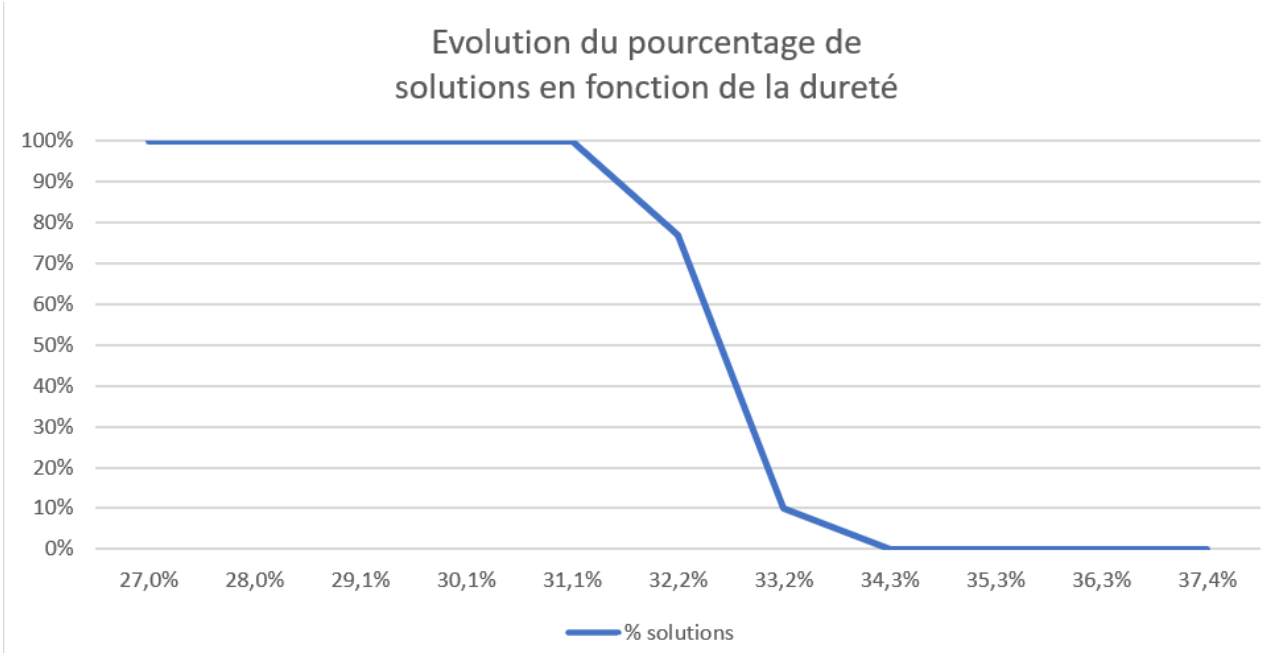

Figure 5: Graphique témoignant de la transition de phase

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

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.

1. On ajoute dans le code un compteur de TimeOut pour ne pas compter les réseaux qui ont timeout dans la moyenne :

```
int nbTO = 0; // compteur de TimeOut
for(int nb=1 ; nb<=nbRes; nb++) { // pour chaque reseau
    ...
    solver.limitTime("30s");
    if (solver.solve()) {...}
    else if (solver.isStopCriterionMet()){ // si le modele TimeOut
        nbTO++;
    }
    else {...}
}
```

Ici, on a choisi la valeur de 30 secondes pour le Time out, car 10 secondes paraissait trop peu, beaucoup de jeux de test n'arrivaient pas à la fin, et 1 minute paraissait trop long, et prenait beaucoup trop de temps selon le benchmark testé.

2. On ajoute une mesure du temps avec la fonction nanoTime(), puis on fait la moyenne de chaque réseau lors de l'écriture dans le fichier. On prend soin d'enlever les réseaux qui ont TimeOut de la moyenne finale pour ne pas fausser les résultats. On obtient donc les données présentes à la figure 6 page 7.

Durete	% solutions	temps moyen (s)
31,10%	100%	0,2
31,50%	100%	0,9
31,80%	100%	1,8
32,20%	100%	3,8
32,50%	100%	3,5
32,90%	90%	8,3
33,20%	60%	8,9
33,60%	22%	13,7
33,90%	0%	9,2
34,30%	0%	5,7
34,60%	0%	5,3
34,90%	0%	5

Figure 6: Données brutes de la transition de phase et du pic du temps d'exécution

3. A l'aide des données précédemment générées, on peut représenter l'évolution du temps d'exécution en fonction de la dureté (voir figure 7 ci-dessous)

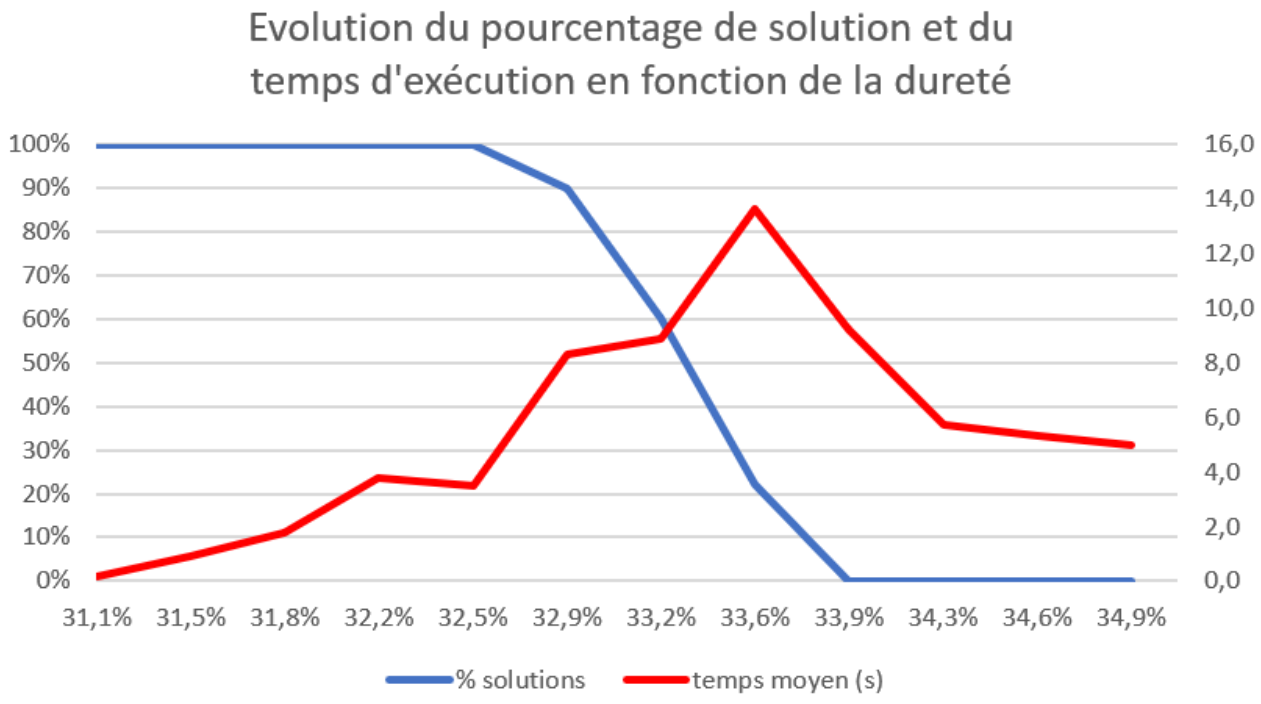

Figure 7: Graphique de la transition de phase accompagnée du temps d'exécution

4. Pour la méthodologie d'évaluation, nous avons lancé le programme sur notre jeu d'essais avec la densité fixée à 40%, et dont la dureté variait de 31% à 35% (avec un nombre de tuples de 188 à 202). La valeur du TimeOut à été définie à 30s.

5. On peut voir sur la courbe obtenue qu'on observe effectivement une transition de phase à partir de 32,5% de dureté. Le temps pris par la fonction de résolution atteint également son pic au milieu de cette transition de phase, qui se situe lorsque la dureté est à 33,5%.
   On peut donc en conclure que c'est lorsque le réseau est à un niveau de dureté "moyen" (pas trop simple, pour ne pas directement trouver une solution, ni trop dur, pour ne pas éliminer directement toutes les solutions), c'est là où la résolution du problème prendra le plus de temps.

Annexes

Il y a donc dans le fichier zip 2 dossiers : TP1 et Projet

1. TP1

Ce dossier contient un dossier *tp-ia-choco*, contenant le projet JAVA ainsi que le sujet du TP1. Dans ce dossier, on a après le chemin suivant : *./src/main/java/*, 3 dossiers : Zebres, AutreSolutionReines, BonneSolutionReines. Ils correspondent respectivement à la première partie du TP1, la solution des nReines demandé par le sujet et une autre solution pour résoudre le problème des nReines.

Ainsi, dans le dossier Zèbres, nous avons 2 fichiers qui ont chacun une représentation différente de leurs contraintes : Extension, Intension.

De la même manière, les deux dossiers sur le problème des nReines ont chacun 2 autres dossiers correspondant aux contraintes définies en intention et en extension.

2. Projet

Ce dossier comporte le sujet du projet, et 4 autres dossiers :

- *code* : Dossier du projet JAVA,
- *generation* : Comporte les fichier fournis et les generations des fichiers dans les dossiers bench1 et bench2,
- *reseaux* : Dossier pour contenir les fichiers créés,
- *resultats* : Fichier excel qui comporte les résultats.