Projet CSP
Génération de benchmarks et évaluation de méthode

Yanis Bouallouche
Chamy Kaci

Novembre 2022

Table des matières

1 Partie A **3**

2 Partie B **4**
2.1 Le script de résolution 6
2.2 Mise en évidence de la transition de phase 6

3 Partie C **7**

1 Partie A

benchSatisf.txt

Pour générer le jeu d'essai `benchSatisf.txt`, il nous suffit d'exécuter le programme `urbcsp.c` comme suit :

```
./urbcsp 10 15 10 80 3
```

Listing 1 – benchSatisf.txt

benchInsat.txt

Pour générer le jeu d'essai `benchInsat.txt`, il nous suffit d'exécuter le programme `urbcsp.c` comme suit :

```
./urbcsp 10 15 10 20 3
```

Listing 2 – benchInsat.txt

Modification de Expe.java

Nous allons modifier le programme `Expe.java` pour pouvoir calaculer le nombre de réseaux ayant une solution pour chaque jeux d'esssai. On commence par `benchSatisf.txt` :

```java
    public static void main(String[] args) throws Exception{

        String ficName = "benchSatisf.txt";
        BufferedReader readFile = new BufferedReader(new FileReader(ficName));
        int nbRes=3;
        int nbResWithSol=0;
        int nbrResSol=0;
        for(int nb=1 ; nb<=nbRes; nb++) {
            Model model=lireReseau(readFile);
            if(model==null) {
                System.out.println("Problème de lecture de fichier !\n");
            }
            System.out.println("Réseau lu "+nb+"\n");
            if(model.getSolver().solve()) {
                nbrResSol++;
                System.out.println("Réseau solvable\n\n");
            }
            else System.out.println("Réseau sans solution\n\n");

        }
        System.out.println("============================================================");
        System.out.println("Le nombre de réseau ayant une solution est : "+nbrResSol);
        return ;
    }
}
```

FIGURE 1 – Expe.java : Calcul de benchSatisf

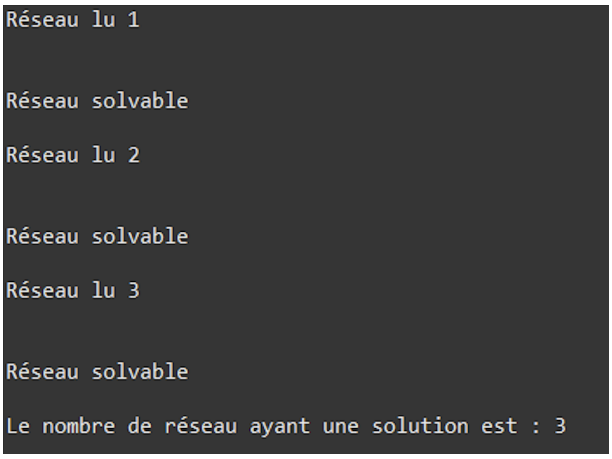

FIGURE 2 – Résultat de benchSatisf

Pour le fichier `benchInsat.txt` le programme `Expe.java` sera le même.

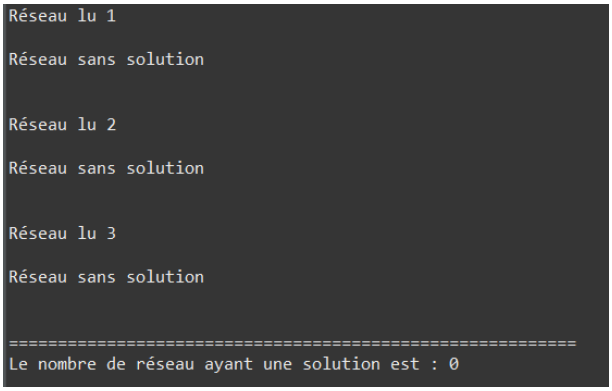

FIGURE 3 – Résulat benchInsat.txt

On remarque que tout les réseaux du jeu d'essai `benchSatisf.txt` ont des solutions, contrairement aux réseaux de benchInsat.txt qu n'ont aucune solution.

2 Partie B

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

Premièrement, on a commencé par modifier le fichier `Expe.java` pour qu'il puisse calculer la fonction % de réseau ayant au moins une solution pour un benchmark dont les paramètres sont spécifiés. Pour cela , on a juste initialisé

une variable nbResWithSol à 0, puis on itère sur le nombre de réseaux et dès qu'une solution est trouvée, on incrémente notre variable. Une fois l'itération terminée, on calcule le rapport (nbResWithSol *100)/nombres de réseaux.

Après exécution `Expe.java` sur `benchsatisf.txt`, on voit que tous les réseaux ont une solution et le pourcentage est à 100%, ce qui est correct.

ET de même pour `benchInsat.txt`, aprés l'exécution de `Expe.java` on voit qu'aucun des trois réseaux n'a de solutions et donc le pourcentage est à 0%.

```
int nbResWithSol = 0;
......code .......
int  pourcentage = (nbResWithSol*100)/nbRes;
```

Listing 3 – % de réseau avec solution

```
Réseau lu 3 :

 Model[Expe]

[ 10 vars -- 10 cstrs ]
satisfaction  : undefined
== variables ==
x[0] = {0..14}
x[1] = {0..14}
x[2] = {0..14}
x[3] = {0..14}
x[4] = {0..14}
x[5] = {0..14}
x[6] = {0..14}
x[7] = {0..14}
x[8] = {0..14}
x[9] = {0..14}
== constraints ==
TABLE ([PropBinAC3bitrm(x[1], x[6])])
TABLE ([PropBinAC3bitrm(x[1], x[7])])
TABLE ([PropBinAC3bitrm(x[0], x[7])])
TABLE ([PropBinAC3bitrm(x[2], x[6])])
TABLE ([PropBinAC3bitrm(x[0], x[1])])
TABLE ([PropBinAC3bitrm(x[7], x[8])])
TABLE ([PropBinAC3bitrm(x[1], x[3])])
TABLE ([PropBinAC3bitrm(x[0], x[8])])
TABLE ([PropBinAC3bitrm(x[0], x[3])])
TABLE ([PropBinAC3bitrm(x[0], x[9])])



nbrSolution  : 1

nombre de reseau ayant une solution :3
pourcentage:100%

************************************
```

FIGURE 4 – console fonction % benchSatisf.txt

2.1 Le script de résolution

```
Start=150
End=300
for ((i=$Start ; i<=$End ; i=$i+1))
do
        ./urbcsp 30 20 160 $i 15 > csp/csp$i.txt
done
```

Listing 4 – Script benchmark

Le script précédent permet de générer les fichiers csp$i.txt de 15 réseaux, 30 variables, 160 contraintes. Il y a 20 valeurs dans le domaine de chaque variable et le nombre de tuples dans chaque contrainte varie entre 150 à 300 avec des pas de +1. Chaque réseau auras donc une densité de 36%

2.2 Mise en évidence de la transition de phase

Pour pouvoir mettre en évidence la transition de phase, nous avons effectué plusieurs essais en changeant les paramètres. Nous sommes donc arrivés à trouver deux combinaisons de paramètres intéressantes.

- Le script vas génerer des fichiers ayant les caractéristiques suivantes :
  - Le nombre de variables : 30
  - La taille des domaines : 20
  - Le nombre de contraintes : 160 ce qui fixeras la densité de chaque réseaux à 36
  - Le nombre de tuples dans une contrainte variera pour chaque fichier, allant de 150 à 300 pas pas de +1
  - On vas générer 15 réseaux pour chaque fichier

  Voici donc la courbe solvable/dureté :

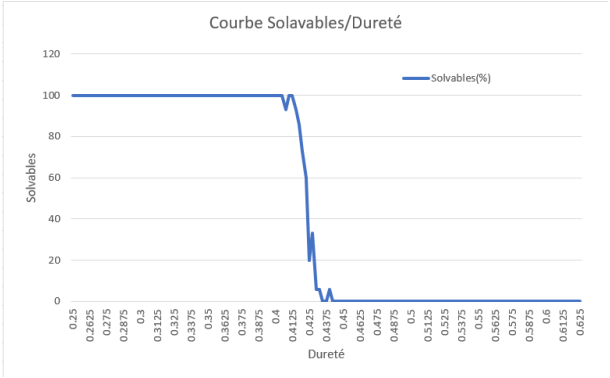

FIGURE 5 – Transition de phase 1

- Pour la deuxième combinaisons de paramètres, le script vas génerer des fichiers avec les caractéristiques suivantes :

— Le nombre de variables : 35
— La taille des domaines : 20
— Le nombre de contraintes : 204 ce qui fixeras la densité de chaque réseau à 34
— Le nombre de tuples dans une contrainte variera pour chaque fichier, allant de 180 à 310 par pas de +3
— On vas générer 20 réseaux pour chaque fichier

Voici donc la courbe solvable/dureté :

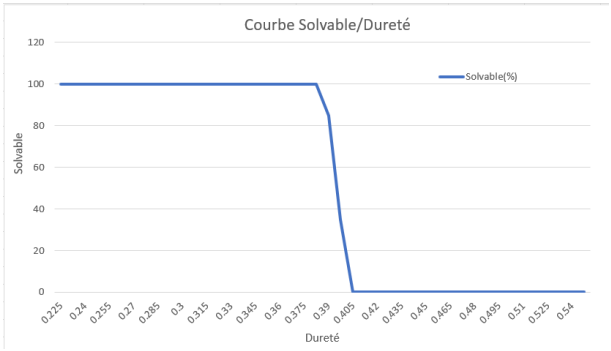

FIGURE 6 – Transition de phase 2

3 Partie C

Mise en place du TimeOut

Nous allons modifier le `Expe.java` pour calculer le temps d'exécution des tests de nos benchmarks. Pour cela nous allons garder les mêmes paramètres que ceux qui nous ont permis d'identifier la transition de phase.
Nous mettons en place un mécanisme de Time Out qui empêchera une attente trop longue, en particulier pandant que la transition de phase s'opère.

On définit une limite de temps et on teste en sortie si le solver s'est arrêté parce qu'il a trouvé une solution ou parce qu'il a atteint la limite impartie de temps.

```
model.getSolver().limitTime("10s");
.....code.....;
if(model.getSolver().solve()) {
.....code.....;
}else { if(model.getSolver().isStopCriterionMet()) {
      System.out.println("......");}
     }else { System.out.println("Pas de solutions");
     }
```

```
9
```

Listing 5 – TimeOut

Calcule des mesures d'évaluation.

On a modifié le `Expe.java` pour calculer nos meseures d'évaluation.

Les mesures choisies sont le temps et le nombre de noeuds moyen générés par l'arbre.

1. Temps

```
1
2
3  ThreadMXBean thread = ManagementFactory.getThreadMXBean();
4  long startCpuTime = thread.getCurrentThreadCpuTime();
5
6  ......code a meusurer......
7
8  long cpuTime = thread.getCurrentThreadCpuTime() -
       startCpuTime;
9
10
11
```

Listing 6 – Temps

Comme mentionner dans le document `Méthodologie d'évaluation des méthodes.pdf`, on s'attachera a mesuré le temps CPU du processus de résolution, afin d'éviter d'être impacté par les autres processus de la machine et par les attentes dues aux entrées/sorties du programme.

2. Nombre de noeuds

   Sur Choco, on dispose d'une méthode qui permet de récupérer le nombre de noeuds générer pandant la resolution d'un réseau. On vas donc récuperer le nombre de réseaux total générer lors de la résolution de tout les réseaux de chaque jeu d'essai, et ensuite calculer la moyenne.

```
1  long nombre_noeuds = 0;
2  BufferedReader readFile = new BufferedReader .....
3
4  int nbNod=;
5  for(int nb=1 ; nb<=nbRes; nb++) {
6  ..... code .......;
7  System.out.println(model.getSolver().getNodeCount());
8  nbNod += (model.getSolver().getNodeCount());
9  }
10 int noeudAvg=nbNod/nbRes;
11
```

```
12
```

Listing 7 – Calcul du nombre de noeuds moyen

la figure suivante nous montre le nombre de noeuds trouver par la methode `getNodeCount()`, on remarque que c'est le même que celui afficher par la methode `printStatistics()`

```
nbrSolution  : 1

nombre de noeuds 9


*** Bilan ***
** Choco 4.10.2 (2019-10) : Constraint Programming Solver
- Model[Expe] features:
        Variables : 10
        Constraints : 10
        Building time : 0,005s
        User-defined search strategy : no
        Complementary search strategy : no
- Complete search - 1 solution found.
        Model[Expe]
        Solutions: 1
        Building time : 0,005s
        Resolution time : 0,004s
        Nodes: 9 (2 300,3 n/s)
        Backtracks: 0
        Backjumps: 0
        Fails: 0
        Restarts: 0
nombre de reseau ayant une solution :3
pourcentage:100%

**************************************
```

FIGURE 7 – console statisitcs

Expliquer la méthodologie d'évaluation mise en place

Comme vous pouvez le voir sur le code, on a laissé tomber la gestion des timeouts car après avoir tracé les courbes, on obtient 0% de réseaux résolus, et ça est dû au timeout fixer à 10s .En effet , cela revient au choix des paramètres du jeu d'essai.

En ce qui concerne le nombre d'exécutions, après plusieurs essais on a décidé de fair qu'une seule exécution par réseau. On va maintenant dessiner les courbes du temps et du nomdre de noeuds moyens pour les deux combinaisons

de paramètres vues un peu plus haut.
Combinaison 1 :

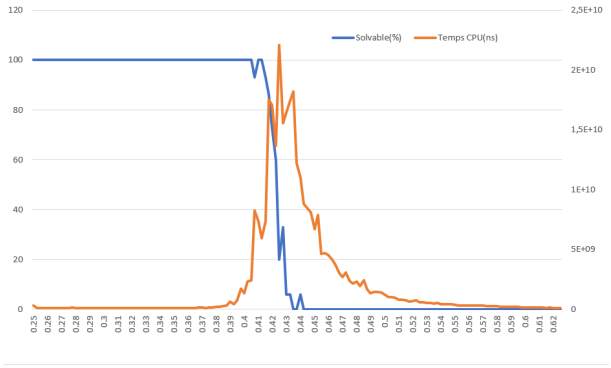

FIGURE 8 – Courbe pourcentage/dureté & TempsCPU/dureté

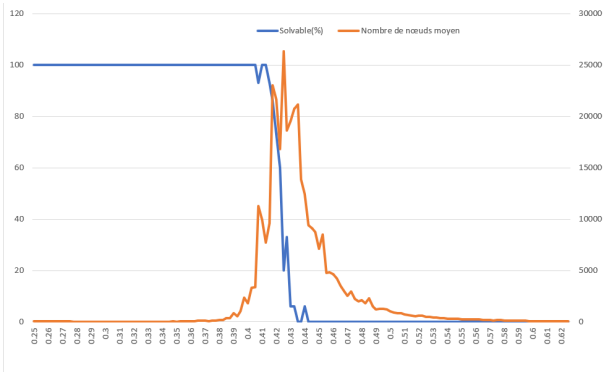

FIGURE 9 – Courbe pourcentage/dureté & Nb noeud moyen/dureté

Combinaison 2 :

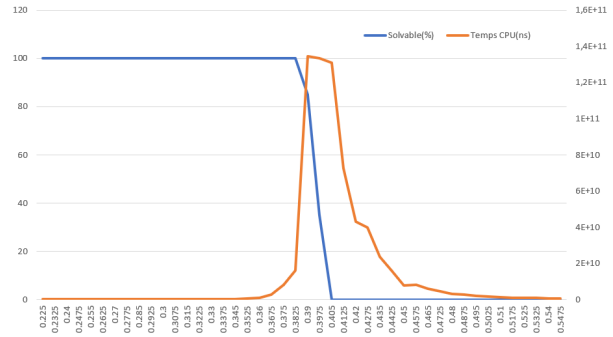

FIGURE 10 – Courbe pourcentage/dureté & TempsCPU/dureté

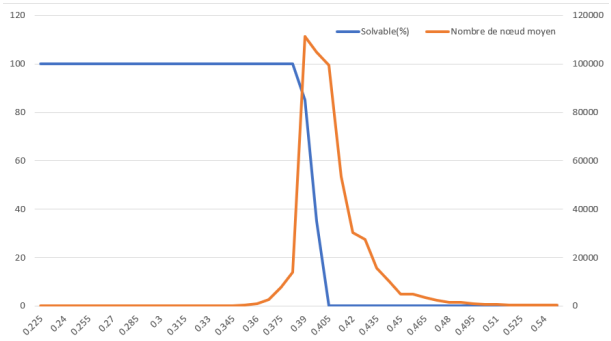

FIGURE 11 – Courbe pourcentage/dureté & Nb noeud moyen/dureté

Analyse des résultats obtenus

On remarque pour les deux cas, que le pic du temps de résolution et du nombre de noeuds moyen se trouve en plein milieu de la phase de transition. On peut en déduire que l'existence de solutions pour les benchmarks en par et d'autres de la transition, est facilement vérifiable, contraint au benchmark qui se situe au coeur de la transition de phase, où le programme prend plus de temps à répondre et développe beaucoup plus l'arbre de recherche.