IA : TP1

Rambal Julien , M1-IASD

October 17, 2023

1 Introduction :

Dans ce travail nous allons nous intéresser à la région dans laquelle le réseau de contraintes contient autant d'instance positive que d'instance négative . Cette région a été définie par cf .. comme la "muddy region" car une grande de performance était observer pour résoudre les réseaux de contraintes appartenant a cette dernière .

1.1 Organisation :

Tout d'abord nous allons présenter l'algorithme de génération du réseau de contraintes Ensuite nous présenterons la méthode de sampling utilisé pour les réseaux de contraintes . Finalement nous expliquerons les résultats du benchmark .

2 Generation :

2.1 Methode :

2.2 Algorithme urbcsp :

2.2.1 Présentation d'urbcsp

Variables :

- N : Nombre de variable
- D : Taille du domaine
- C : Nombre de contraintes
- T : Nombre de tuples interdit
- S : Seed
- I : Nombre de réseau (instance de problème)

Limite de chaque variable :

- N : ≥ 2 (au moins deux variables dans le réseau)
- D : ≥ 2 (au moins deux valeurs distinctes par variables)
- C : $\geq 1 \leq \frac{N*(N-1)}{2}$
- T : $\geq 2 \leq (D * D) - 1$
- S : $\geq 0 \leq 2^{63}$ (Long.MIN_VALUE et Long.MAX_VALUE)
- I : ≥ 1

urbcsp génère I graphe non orienté possédant N sommets connectés par C arêtes de poids $\overline{T}$. La probabilité de sélection de chaque tuple et chaque arête est paramétré par C et T ainsi nous pouvons moduler la densité du graphe avec C et le "degré de satisfiabilité" de ses contraintes avec T .

2.2.2 Explication d'urbcsp

La sélection des C contraintes se fait en 2 étapes tout d'abord, on génère l'ensemble des contraintes binaires permis (card = ensembles des arêtes dans G : $\frac{N*(N-1)}{2}$)puis on y sélectionne N contraintes :

```
// ...
// Creation des N * (N - 1)/2 couples de variables/contraintes/ar tes possible
// avec N card de V

for (c=0; c {
/* Choose a random number between c and PossibleCTs - 1, inclusive. */
r = c + (int) (ran2(Seed) * (PossibleCTs - c));
// ...
}

// ...
```

Il en va de même pour la sélection des T tuples interdits :

```
// ...
// Creation des D*D tuples possible avec D card de V[i]

      for (t=0; t<T; t++)
  {
    /* Choose a random number between t and PossibleNGs - 1, inclusive.*/
    r =  t + (int) (ran2(Seed) * (PossibleNGs - t));

// ...
```

2.3 Resultats :

Le graphe généré par urbcsp est un CSP binaire paramétrée par (N,D,C,T,S) qui est stocké dans un fichier texte de la forme :

Header :

```
nbVar
nbDom
```

Content :

```
nbContrainte
portee
nbTuple
[Tuples]
```

3 Sampling :

3.1 Methode :

3.1.1 Présentation :

Dans ce TP nous fixerons les valeurs des paramètres N , D aux valeurs N=20 et D=10 comme ce qui a été proposé par P.Posser 1994 .
En revanche pour les autres paramètres C et T suivront deux lois binomiales et S sera généré par le Mersenne Twister (librairie : org.apache.commons.math3) .
Ceci fait sens car pour l'instant nous n'étudions que l'impact des paramètres C et T sur "la dureté" du problème/réseau .

Variables :

- v1 : variable aléatoire correspondant au nombre de relation dans le graphe (contraintes)
- v2 : variable aléatoire correspondant au poids soustrait à chaque relation (tuples interdits)
- p1 : probabilité qu'une relation existe
- p2 : probabilité qu'un tuple soit interdit
- n1 : nombre de relation possible
- n2 : nombre de tuple interdits possible

La variable aléatoire v1 suit une loi binomiale de paramètre : (p1,n1) $\mathbf{B}(p1, n1)$. Pour mimer le comportement d'une variable nous utilisons le *MersenneTwister* qui est l'un des RandomGenerator les plus utilisés . La distribution, quant à elle, est simuler par la fonction *BinomialDistribution* :
NbRelations :

```
// RandomGenerator generator = new MersenneTwister(); // partager par bin_p2
    et bin_p1

BinomialDistribution bin_p1 = new
    BinomialDistribution(generator,(nbVar*(nbVar-1))/2, p1);

nbRelations = bin_p1.sample();

if(nbRelations==0)nbRelations++;
```

Nous avons repris cette implémentation pour la variable aléatoire v2 à ceci près que $v2 \to \mathbf{B}(p2, n2)$.

NbTuples :

```
// RandomGenerator generator =  new MersenneTwister(); // partager par bin_p2
    et bin_p1

BinomialDistribution bin_p2 = new
    BinomialDistribution(generator,nbDom*nbDom-1, 1-p2);

nbTuples =bin_p2.sample();

if(nbTuples==0)nbTuples++;
```

Les échantillons des distribution de v1 et v2 seront utilisés pour générer un réseau de contrainte grâce à l'algorithme fournis : urbcsp .

3.1.2 Génération des samples :

Le pseudo code pour générer une instance d'un problème P paramétré par le couple (p1,p2) :

```

urbcsp(20,10,nbRelations(p1),nbTuples(p2),Seed,1); // avec Seed un long s lectionn
```

Ainsi pour générer un ensemble d'instance il nous suffit de faire varier p1 et p2 , dans un premier temps nous déciderons d'incrémenter p1 par 0.1 unité [0.1 : 1] , p2 par 0.01 unité [0.01 : 1[et finalement par couple (p1,p2) nous générerons 10 réseaux distincts , ainsi nous avons un total de $10 * 99 * 10 = 9900$ samples

Variables :

- step_p1 : incrément de p1 (0.1)
- step_p2 : incrément de p2 (0.01)
- nbSamples : nombre de samples par couple (p1,p2) (10)

```
for (;p1<=1;p1+=step_p1) {
//choix de p1
    for (;p1<=1;p1+=step_p1) {
        //choix de p2
            for (int i = 1; i <= nbSamples; i++) { // avec i le numero du
                Reseau
                    urbcsp(20,10,nbRelations(p1),nbTuples(p2),Seed,1);
                    }
    }
}
```

3.1.3 Résultats :

Les graphes générés par urbcsp sont stockés dans des fichier texte de noms :

```
./samples/Test_Var20_Dom10/R[nombreRelations]/numero_[i]_nbTuples_[nbTuples]
```

Et dont le contenu est formatté comme vue précédemment (algo urbcsp) .

3.2 Résolution des réseaux :

3.2.1 Présentation :

Pour résoudre les réseaux de contraintes précédemment générées par urbcsp nous utiliserons la librairie choco-solver proposé par C.Prud'homme et son équipe .

Une instance d'un problème quelconque sera représenté par la classe AbstractProblem_bis et implémenté par Expe :

Variables :

Hérité de AbstractProblem_bis :

- ficName : nom du fichier d'input (String , format : lettre+ .txt)
- timeLimit : temps limite pour la recherche de solutions (String ,format : (d)+m)
- nodeLimit : nombre maximal de nœuds développé par solution (Integer)
- model : entre autre une structure $< D , V , C , R >$ avec V les symboles de variables , D les domaines de ces variables , R les symboles de relation (contraintes) et C les symboles de constantes . Mais aussi une assignation/environement qui associe à chaque V une valeur dans D . (Model)
- csvStat : les statistiques de la recherche des assignations satisfiant le modèle (String)

Propre à Expe :

- var : tableau des variables entières décrivant le problème

Methodes :

Hérité de AbstractProblem_bis :

- setUp : initialise les paramètres ficName , timeLimit , nodeLimit .
- buildModel : construit le modèle grâce au fichier d'input et la fonction lireReseau fournis .
- configureSearch : fixe entre autres la stratégie et les limites (noeud développé max , temps max ,ect) de la recherche
- solve : effectue la recherche et update l'attribut csvStat
- execute : appelle setUp , buildModel , configureSearch et Solve et retourne csvStat

Propre à Expe :

- tocsv : retourne un String formatté comme suit ,

```
[solutionCount]%d;[TimeCount]%.3f(en ms);
[NodeCount]%d;[BacktrackCount]%d;
[BackJumpCount]%d;[FailCOunt]%d;
[RestartCount]%d;[MaxDepth]%d;
```

3.2.2 Algorithme :

```
ArrayList<String> csv_tab ; // tableau stockant les statistiques csv des
recherches de tous les [1:nbSamples] r seaux des [0.01:1[ p2 de chaque p1

//...
// urbcsp(20,10,nbRelations(p1),nbTuples(p2),Seed,1) > ficname ;

String tmp_stre= new Expe().execute_bis(bius);
csv_tab.add(subNamecsv);

//...
}//fin for loop p2
//ecrit dans output.txt csv_tab

//...
```

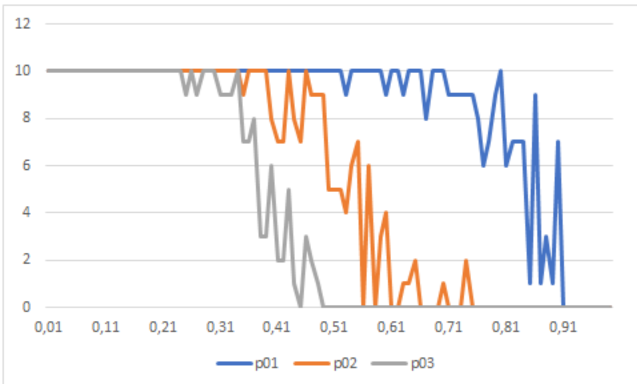
3.2.3 Résultats :

Output.txt est un fichier écrit dans chaque répertoire R[NbRelations] qui récapitule les statistiques des réseaux de ce répertoire .
Il est formatté comme suit :

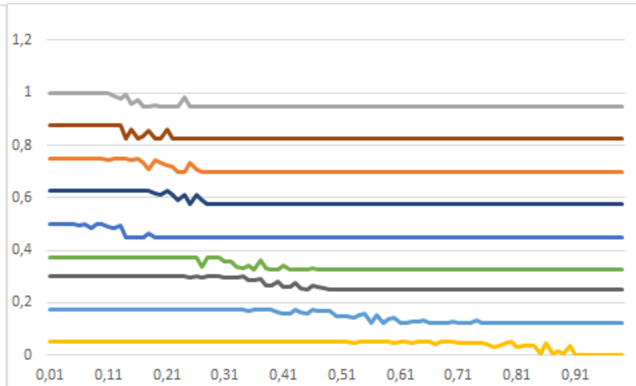
```

//header :
timeLimit;nodeLimit;nbVar;nbDom;nbSamples;p1;nbConstraint;p2;nbTuples;numeroReseau;
    nbSolution;resTime;nbDevNode;nbBacktrack;nbBackJump;nbFails;nbRestart;maxDepth

//content :
// valeurs correspondantes (1 ligne pour chaque r seau)
```

3.3 Résultats :

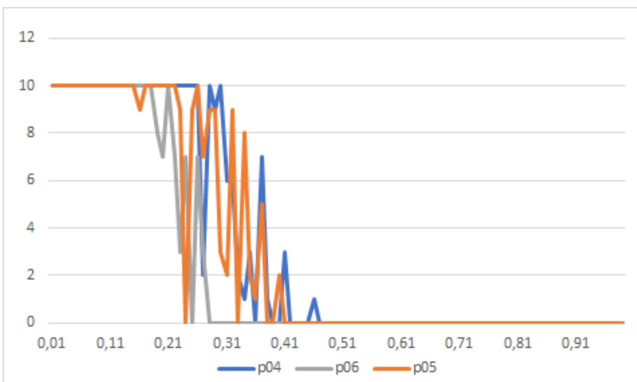
3.3.1 Schématiser les fichiers d'Output :



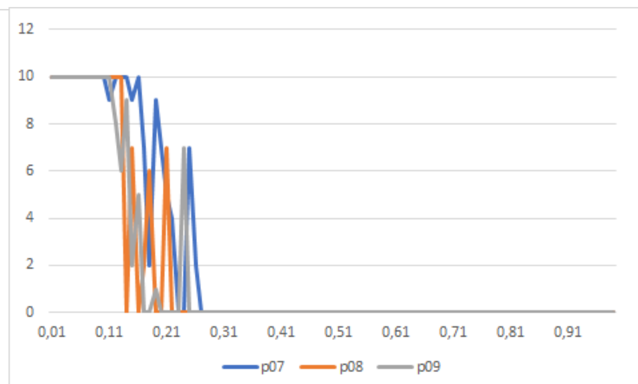
((a))



((b))



((c))



((d))

Figure 1: a) p1 $<$ 0.3 , b) Fusion c) et d) p1 $>$ 0.3
varSelector FirstFail , valSelector IntDomainMin et decision : IntEqDecision , proposé par Julien

3.3.2 Observation :

Nous observons que sur tous les critères précédemment évoqués la région de plus grande complexité pour p1 = 0.5 (en temps et non en nombre de solutions comme vue plus haut) se situe en $p2 \in]0.37 : 0.45[$. Ceci peut être expliqué car dans cette région les instances insatisfiables et satisfiables se mélangent :

La première observation est que l'algorithme/stratégie de recherche utiliser ne change pas significativement l'emplacement de la transition de phase .

Les réseaux de cette région sont en moyenne à peine satisfiables (dans notre recherche naive les critères d'arrêt arrête la recherche avant d'atteindre la solution en question ou de prouver l'insatisfiabilité du réseau) .

Nous observons que pour un certains nombres de relations il existe dans cette région des instances avec un grand nombre de solutions et d'autres avec un nombre de solution nul .

Une autre observation est que la zone se décale vers la gauche (vers l'origine) plus le nombre de relations (densité du graphe) est important

Finalement , au vue des résultats , nous pouvons supposer que l'instance de "difficulté " maximale est souvent une instance à peine satisfiable . Sa position partage le graphique en 2 parties : majoritairement satisfiable vs majoritairement insatisfiable .

3.3.3 Formules :

D'après le papier "Scaling Effects in the CSP Phase Transition" , nous pouvons estimer cette zone critique avec la formule :

$\hat{p}_{2crit} = 1 - m^{\frac{-n}{E(C)}}$, $E(C) = p_1 n(n-1)/2$

avec m la taille des domaines , n le nombre de variable et p_1 : p1 .

Ce qui donne environ 0.384 resp pour $\hat{p}_{2crit}$ avec $p_1 = 0.5$ .Ceci semble confirmer notre approximation de p2 ($\in$]0.37 : 0.45[ ) bien que notre soit un peu décaler vers la droite .

La moyenne des solutions d'un réseau quant a elle est estimé par :

$E(N) = m^n(1-p_2)^{fracp_1.n(n-1)2}$

Dans notre cas la moyenne du nombre de solutions pour p2=0.384 est d'environ 1.023 ,ce qui une nouvelle fois confirme notre intuition .

Remarque :
Ces estimations sont moins fiable pour des réseaux peu denses .

3.4 Conclusion :

Le graphe se partage en 2 parties proches de l'origine les solutions sont nombreuses car le réseau est peu contraints , à l'opposé , le réseau est très contraints et il n'y a plus de solution . Ces deux opposés ont des difficultés similaires, car l'un ne prune presque rien et l'autre presque tout ainsi on arrive facilement à prouver la satisfiabilité resp l'insatisifiabilité .

La topologie des graphes générés entraînent l'apparition d'outlier . Par exemple : un noeud déconnecté (se produit souvent quand p1 faible) , et un noeud hyper connecté (se produit souvent quand p2 faible) peuvent entrainer des cout moindre tandis que pour un graphe régulier le "pruning/assignation" seront plus difficile .

4 Ouverture :

Il serait pertinent d'augmenter le nombre de sample dans la zone de transition de phase pour mieux constater les variations .

⟶ Paramétré le nombre de sample en fonction de la variabilité des temps de calcul (gradient) serait une option .
Nous avons constater que certains échantillon "font" apparaitre la transition de phase en amont ou en aval de notre prédiction à cause des débuts d'apparition des "instances complexes" .
⟶ Une solution serait d'utiliser la médiane et non la moyenne .
Finalement il serait également interessant d'évaluer l'impact du nombre de variable sur la complexité des instances générés .

References

[1] *Scaling Effect in the CSP phase Transition* , Ian P. Gent, Ewan MacIntyre, et al. , https://link.springer.com/chapter/10.1007/3-540-60299-2_5

[2] *Generating Random Solutions for Constraint Satisfaction Problems* , Rina Dechter Kalev Kask, et al. , https://www.researchgate.net/publication/2837162_Generating_Random_Solutions_for_Constraint_Satisfaction_Problems

[3] *An emperical study oh phase transition in binary constraint satisfaction problem* , Patrick Posser , https://www.sciencedirect.com/science/article/pii/0004370295000488

[4] *Random Uniform CSP Generators* , Christian Bessiere , https://www.lirmm.fr/~bessiere/generator.html

[5] *Introduction à l'IA* ,Michel Leclère ,https://www.lirmm.fr/~leclere/

[6] *Constraint Processing* , Rina Dechter , https://www.sciencedirect.com/book/9781558608900/constraint-processing

[7] *Handbook of Constraint Programming* , F.Rossi , P.Van Beek , T.Walsh , https://www.elsevier.com/books/handbook-of-constraint-programming/rossi/978-0-444-52726-4

[8] *Principles of Constraint Programming* , Krzysztof Apt , https://www.cambridge.org/core/books/principles-of-constraint-programming/C008FB32571F66C3EE0EEEBDE1F98A7D