Heuristiques, optimisation des algorithmes, comparaison des méthodes, expérimentation.

La résolution de CSP se heurte à une explosion combinatoire qui peut engendrer des temps de résolution très long. Le problème d'existence d'une solution est caractérisé formellement comme un problème NP-complet par la théorie de la complexité qui estime le nombre d'instructions à exécuter pour résoudre un réseau de contraintes (le pire) en fonction de sa taille. Cf. section 1.

On sait donc que l'on ne peut pas résoudre en temps polynomial toutes les instances du problème. Cependant on souhaite comparer l'efficacité de nos algorithmes en pratique (par opposition à la théorie). Pour cela, on construit des bancs d'essais composés d'instances spécifiques sur lesquels les différents programmes sont testés. Cf. section 2.

1. Rappel sur la complexité de problèmes

La théorie de la complexité dans sa forme la plus simple s'intéresse à classifier les problèmes de décision, c'est à dire les problèmes pour lesquels les deux réponses possibles sont *oui* ou *non* (typiquement : *existe-t-il une solution à tel problème ?* Un CSP, un problème de coloriage de graph, etc.), dans des classes imbriquées. Deux classes sont principalement considérées :

- la classe P des problèmes polynomiaux (on dit aussi *traitables*), ceux pour lesquels il existe un algorithme (exécutable sur une machine séquentielle classique) qui calcule la réponse en temps polynomial en la taille de la donnée du problème quelque soit l'instance (i.e. la donnée) considérée ;
- la classe NP des problèmes polynomiaux non-déterministes, ceux pour lesquels il existe un algorithme, exécutable sur une machine capable d'exécuter en parallèle un nombre infini d'alternatives, qui calcule la réponse en temps polynomial en la taille de la donnée du problème quelque soit l'instance considérée ; cela peut se voir comme l'existence d'un nombre infini de « cas » (des solutions potentielles) à tester mais vérifier si chacun de chacun chaque cas peut être vérifié en temps polynomial.

Clairement tout problème dans la classe P est donc également dans la classe NP, $P \subseteq NP$. Pour certains problèmes de NP on ne trouve pas d'algorithme polynomial. Les problèmes les plus difficiles de la classe NP sont appelés NP-complets. Un problème est NP-complet[1] s'il est dans NP et qu'il est au moins aussi difficile à résoudre que n'importe quel autre problème de NP (i.e. n'importe quel autre problème de NP peut être polynomialement transformé en ce problème, on parle de réduction) : cela définit donc une classe de problème tous équivalents telle que si on trouve un jour un algorithme polynomial pour l'un de ces problèmes, tous rentrent dans la classe P et on aurait P=NP. Il est conjecturé que P≠NP.

Dans le contexte des CSP, le problème de décision qui nous intéresse est : *« étant donné un réseau de contraintes N, existe t'il une assignation des variables de ce réseau qui satisfait les contraintes (i.e. une assignation localement consistante complète) ? »*

La taille d'un réseau *N=(X,D,C)*, instance d'un problème de CSP, est dépendante de :

- *n* le nombre de variables de *X*
- *d* la taille maximale d'un domaine de *D*
- *c* le nombre de contraintes dans *C*

[1] En pratique, grâce au théorème de Cook (1971) qui prouve que SAT est NP-complet, il suffit d'exhiber une réduction depuis un autre problème NP-complet pour prouver qu'un problème est NP-complet.

- k l'arité maximale d'une contrainte de C
- t le nombre maximal de tuples d'une contrainte de C

On ne connaît pas d'algorithme polynomial pour résoudre ce problème quelque soit l'instance en entrée. Il en existe des polynomiaux pour certaines catégories de réseaux (par exemple ceux dont le graphe de contraintes a une structure arborescente ou quasi-arborescente). On ne peut donc pas classer ce problème dans P sans pour autant être sûr qu'il n'y est pas ! Ce problème est par contre clairement dans NP ; on peut vérifier en temps polynomial qu'une assignation complète satisfait les contraintes. Ce problème est NP-complet. On peut par exemple exhiber une réduction depuis SAT.

2. Comparaison expérimentale

Afin d'évaluer les différentes méthodes, algorithmes, heuristiques, structures de données, programmes proposés pour la résolution de CSP, on utilise des bancs d'essais composés d'instances spécifiques sur lesquels les différents programmes sont testés. Ces « benchmarks » peuvent être bâtis à partir de cas réels (par exemple un ensemble de problèmes d'emploi du temps) ou construits par tirages aléatoires de réseaux. Dans ce dernier cas, il est intéressant de sélectionner des instances à résoudre à différents niveaux de difficulté.

Intuitivement, un réseau peu contraint, qui a donc beaucoup de solutions, est un réseau « facile » à résoudre car on a de grandes chances de rapidement trouver une solution. De même un réseau très contraint, qui n'a pas de solution, est un réseau facile à résoudre car on va vite exhiber une inconsistance. Dans ces deux cas, les méthodes devraient donc rapidement prouver l'existence/l'absence de solution.

Entre ces deux extrêmes, il existe des réseaux « difficiles » à résoudre car au fur et à mesure de l'extension des assignations, on conserve autant de chance d'avoir une solution que de ne pas en avoir ; ce sera donc a priori plus long de trouver une solution (ou de vérifier qu'il n'y en a pas).

2.1. Transition de phase

En physique, on désigne par la notion de transition de phase la transformation physique (brutale) d'un système d'un état vers un autre, induite par la variation d'un paramètre de contrôle (par exemple le passage de l'état solide à l'état liquide en fonction d'une variation de température).

Dans les années 90, ce phénomène de transition de phase a été mis en évidence pour de nombreux problèmes informatiques de décision : le changement d'état observé est le passage de l'état réponse *oui* à l'état réponse *non* en fonction d'un paramètre faisant varier la difficulté du problème (pour notre cas, le passage de réseaux peu contraint à des réseaux très contraint).

En pratique, pour mettre en évidence ce phénomène :

1. On choisit un paramètre p qui permet de faire évoluer la difficulté des réseaux et on fixe les valeurs d'observation de ce paramètre (l'échelle de précision de la difficulté).
2. On génère de manière aléatoire b réseaux pour chaque valeur choisie de ce paramètre.
3. On exécute l'algorithme de résolution (backtrack ou autre) sur chacun des problèmes générés afin de classifier le réseau dans la catégorie « oui » des réseaux ayant une solution, ou « non » des réseaux n'ayant pas de solution.
4. On calcule pour chaque valeur choisie du paramètre p le nombre de réseaux classifiés « oui » par rapport au nombre b de réseaux générés pour cette valeur et on exprime ce rapport en pourcentage (le % de réseau ayant une solution).

5. On représente alors sur un graphique ce % de réseau ayant une solution en fonction des différentes valeurs de p.

Si le paramètre p est bien choisi, on doit pouvoir observer un graphique du type de celui de la figure 1.

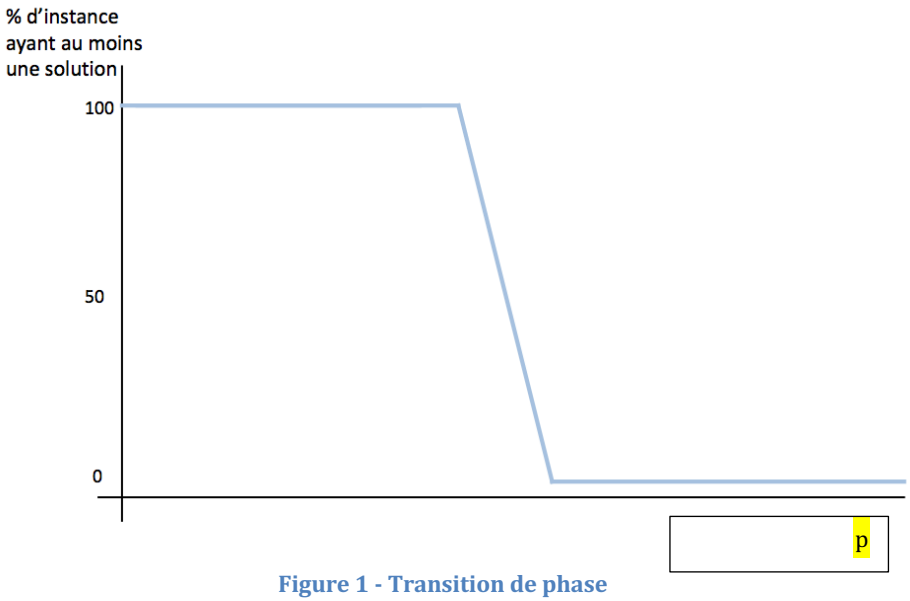

Figure 1 - Transition de phase

Pour les réseaux de contraintes, deux paramètres sont généralement considérés :

- la densité D du réseau : le rapport entre le nombre de contraintes c et le nombre total de contraintes[2] possibles pour le nombre n de variables et l'arité k choisis. On a donc : $D = \frac{c}{\binom{n}{k}}$. Pour une arité binaire, i.e. $k=2$, on aura donc $D = \frac{2c}{n^2-n}$.
- la dureté T des contraintes du réseau (que l'on imposera être la même pour chaque contrainte du réseau) ; la dureté d'une contrainte étant définie comme le rapport entre le nombre de tuples interdits de la contrainte et le nombre total de tuples pouvant appartenir à la contrainte (c'est-à-dire la taille du produit cartésien des domaines des variables de la contrainte). Pour un réseau de contraintes binaires dont le nombre de tuples choisis est t et dont les domaines sont de taille d, $T = \frac{d^2-t}{d^2}$.

2.1. Mesure de temps

Après avoir généré un benchmark et identifié le niveau de difficulté des réseaux générés, ceux de part et d'autre de la transition étant faciles alors qu'au cœur de la transition de phase se trouve les plus durs, on peut mesurer la performance d'un programme de résolution sur l'ensemble des niveaux de difficulté. On peut de même comparer deux programmes différents et en particulier mesurer l'apport d'une heuristique ou d'une optimisation par rapport à la version de base.

La façon la plus classique, d'évaluer la performance d'un programme consiste à mesurer le temps passé à l'exécution de ce programme. On s'attachera à mesurer le temps CPU du processus de résolution, c'est à dire la durée pendant laquelle un microprocesseur a effectivement exécuté le processus, plutôt que le temps utilisateur pour éviter d'être impacté par les autres processus de la machine et par les attentes dues aux entrées/sorties du programme.

[2] On ne considère qu'une contrainte pour un ensemble de variables donnée.

Une manière très simple de faire ça sous Linux consiste à utiliser la commande *time* qui prend en paramètre la commande à exécuter (cf. *man time*). Cette commande sans option spécifique exécute le programme en paramètre et retourne différentes mesures de temps d'exécution de ce programme :

- le temps réel (real) : différence de temps entre le moment où l'exécution de la commande *time* a démarré et le moment où elle s'est terminée ;
- le temps utilisateur (user) : correspondant au temps CPU passé à exécuter le programme passé en paramètre en mode (hors appel aux fonctions systèmes);
- le temps système (sys) : correspondant au temps CPU passé à exécuter les appels systèmes du programme passé en paramètre.

Le temps CPU total, c'est à dire le temps durant lequel un microprocesseur a été affecté à l'exécution de votre programme est la somme entre le temps utilisateur et le temps système.

Le problème de cette technique est qu'elle prend en compte le temps de chargement du programme et ne permet pas de mesurer une partie spécifique de votre programme. Une autre solution consiste à utiliser des fonctions spécifiques de votre langage de programmation. Pour mesurer le temps CPU total en java, on peut utiliser la fonction `getCurrentThreadCpuTime` qui retourne le nombre de nanosecondes pendant lequel le thread courant a été exécuté par un micro-processeur (donc a priori le ramasse-miettes (*garbage collector*) de java n'est pas compté). Les fonctions *`nanoTime`* et `getCurrentThreadUserTime` permettent d'obtenir respectivement le temps réel et le temps utilisateur. Exemple :

```
import java.lang.management.ManagementFactory;
import java.lang.management.ThreadMXBean;
   //   ...
ThreadMXBean thread = ManagementFactory.getThreadMXBean();
   //   ...
long startTime = System.nanoTime();
long startCpuTime = thread.getCurrentThreadCpuTime();
long startUserTime = thread.getCurrentThreadUserTime();
   // ... the code being measured ...
long userTime = thread.getCurrentThreadUserTime() - startUSerTime;
long cpuTime = thread.getCurrentThreadCpuTime() - startCpuTime;
long sysTime = cpuTime – userTime
long realTime = System.nanoTime() - startTime;
```

Attention d'une exécution à l'autre sur le même programme et les mêmes données, il peut y avoir de grandes différences de temps dues à certains mécanismes de caches, à la chauffe des composants, à des paramètres d'optimisation de la JVM… Par ailleurs la précision des outils en mode exclusif (mono-user) est de l'ordre de 1 ms. Pour éviter, ces différents problèmes on essaiera de :

- s'attacher à mesurer des temps supérieurs à 10 ms ;
- « chauffer » la JVM en faisant une dizaine d'exécutions avant de lancer celles sur laquelle la mesure est faite ;
- faire plusieurs exécutions (5 à 10) d'un même programme sur les mêmes données et retenir la moyenne robuste (c'est à dire la moyenne après avoir enlever les plus petites et plus grandes valeur) des temps d'exécution.

Dans de nombreux cas, on peut se heurter à des temps d'exécution trop long sur certaines instances. Pour éviter d'attendre trop longtemps (voire indéfiniment le résultat d'une exécution), une solution consiste à ajouter une limite de durée à l'exécution, un time-out (TO). Dès qu'un TO est déclenché, il faut alors se méfier des moyennes de temps calculé (qui doivent être considérées en TO) ; dans certains cas on peut même ne pas

connaître le résultat d'une instance (i.e. déterminer si le réseau possède ou non une solution) car toutes ses exécutions sont en TO.

Pour des problèmes d'exploration d'espaces de recherche, on peut aussi remplacer la mesure du temps par une mesure du nombre de nœuds explorés de l'espace de recherche et compléter par une estimation du temps moyen passé à explorer un nœud.

2.3. Présentation des résultats

Il s'agit non seulement de décrire les enseignements tirés de l'expérimentation réalisée mais également de bien préciser ce qui a été expérimenté afin d'assurer la reproductibilité de l'expérimentation. On précisera :

Les résultats :

- Donner les conclusions de l'expérimentation : telle méthode est plus rapide que telle autre, telle heuristique est plus appropriée dans tel cas …
- Fournir différents graphiques présentant les temps d'exécution permettant d'appuyer vos conclusions. Typiquement, on s'attachera à calculer la moyenne des temps d'exécution de chaque réseau d'une même classe de difficultés. Des outils tels gnuplot (cf. https://doc.ubuntu-fr.org/gnuplot) qui permettent de facilement tracer des graphiques. Si vous avez correctement identifié la transition de phase, vous devriez avoir un pic de temps de calcul au niveau de la transition de phase (cf. figure 2).

Le benchmark :

- Fournir un accès aux données,
- Expliquer le format des données,
- Préciser éventuellement les résultats attendus sur chaque donnée (par exemple le réseau est-il solvable, nombre de solutions…)
- Préciser comment le benchmark a été construit, quelles sont ses caractéristiques (on pourra ici adjoindre une figure illustrant la transition de phase).

L'expérimentation (tout ce qui permet d'assurer sa reproductibilité) :

- Quels résultats et temps obtenus ?
- Qu'avez-vous mesuré (temps, nombre de nœuds, unité, moyen de mesure, TO) ?
- Quel programme (si possible le rendre disponible) ?
- Quelles données (si possible les rendre disponibles) ?
- Avec quels outils ?
- Quel matériel (Processeurs, Cache, RAM…) ?
- Quel système (nom et version) ?
- Quel compilateur a été utilisé pour fabriquer votre exécutable (nom et version) ?
- Quelles options de compilation ?
- Quelle conditions d'exécution (mono-utilisateur, multi-utilisateur, priorités, autres activités sur la machine…) ?
- Présence d'entrées/sorties dans le code exécuté ?

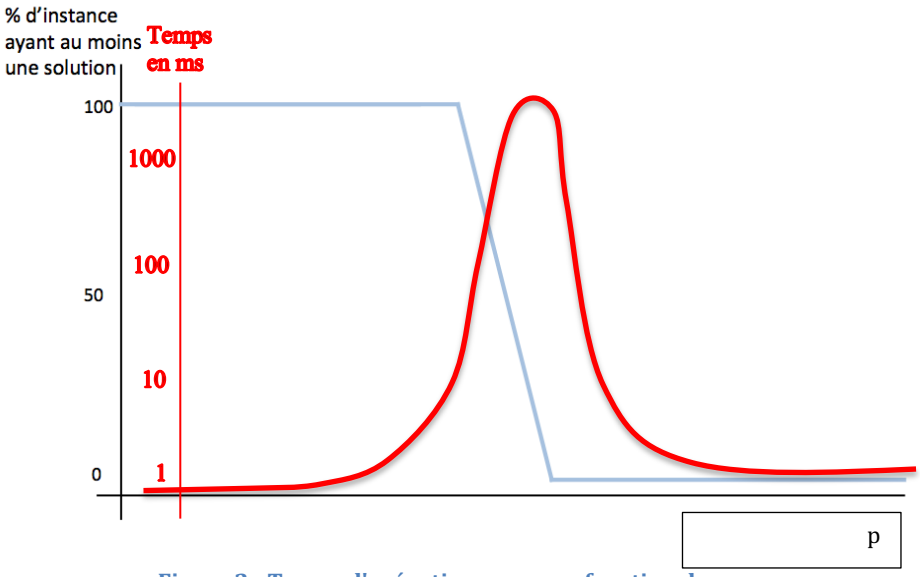

Figure 2 - Temps d'exécution en ms en fonction de p

3. Mise en application sur les CSP

Pour construire un benchmark pour vos tests d'algorithmes de résolution de CSP, vous pouvez utiliser le générateur de CSP binaire *urbcsp*.

Vous pouvez par exemple construire un benchmark ayant les caractéristiques suivantes :

- l'arité k=2
- le nombre de variables n=35
- la taille des domaines d=17
- le nombre de contraintes c=249, ce qui fixe la densité à 41,8%
- et faire varier t le nombre de tuples demandés par contraintes de 211 à 178 par pas de -3, ce qui fera varier la dureté de 27% à 38% par pas de 1%

Vous générez 10 instances pour chaque valeur de dureté soit 130 réseaux.

Pour chaque réseau vous faites 5 exécutions, éliminez les temps le plus rapide et le plus lent, et faites la moyenne des 3 temps restants.

Mettez en évidence la transition de phase observée sur votre benchmark en traçant la courbe % d'instances ayant une solution par rapport en fonction de la dureté.

Tracez alors le graphique temps moyen de calcul d'une instance d'un niveau de dureté donné par rapport au niveau de dureté. Vous devriez observer un pic au niveau de la transition de phase.

Pour chaque heuristique, algorithme à tester/comparer, recommencez les exécutions du nouveau programme sur l'ensemble des instances du benchmark et tracez le graphique temps moyen de calcul / dureté.

Comparez les graphiques obtenus pour définir les « meilleures » heuristiques, algorithmes, programmes.