Rapport de stage Master 2 Recherche Informatique Fondamentale Université de Nice Sophia Antipolis

Palmieri Anthony

3 décembre 2015

Table des matières

1 Introduction

En 1946 apparaissait le premier ordinateur au sens que l'on entend aujourd'hui. Celui-ci avait pour but d'effectuer des calculs balistiques pour l'armée américaine. L'informatique a toujours été un outil très important pour résoudre des problèmes. C'est pourquoi les Constraint Satisfaction Problems (CSP) se sont développés. Les CSP sont des méthodes génériques de modélisation pour résoudre des problèmes. Le terme CSP peut faire référence à plusieurs techniques telles que la programmation linéaire (ou programmation linéaire en nombre entier si notre problème contient des variables avec un domaine discret), la programmation par contraintes et les problèmes de satisfaction booléen. Notre contexte nous amènera à nous intéresser plus particulièrement à la programmation par contraintes.

La résolution de problèmes avec un algorithme dédié ou un modèle (e.g un CSP), met au jour le fait qu'il en existe avec un temps de résolution exponentiel. Ce qui veut dire que si on dispose de données en entrée de taille n, alors le temps de résolution $\approx e^{h(n)}$. Avec h(n) une fonction quelconque dépendante de n. Ces problèmes sont difficiles et bien souvent inexploitables en pratique même, pour des instances de tailles modérée. Ces problèmes s'appellent NP-Complet [1]. Les heuristiques sont étudiés pour faire face à cette difficulté. Ce sont des méthodes de calculs qui permettent, en règle générale, d'obtenir rapidement une solution réalisable mais pas nécessairement optimale. Néanmoins il arrive, que le temps de réponse d'un heuristique soit considérable (i.e le temps est quasi infini).

Prenons par exemple le problème de coloriage dans un graphe. Nous disposons en entrée d'un graphe. L'objectif est de déterminer quel est le nombre de couleurs minimales nécessaires pour le colorier tel que :

— tous les sommets aient une couleur.
— deux sommets adjacents (i.e reliés par une arête) n'aient pas la même couleur.

Nous pouvons voir un exemple de graphe colorié sur la Figure 1.

Plusieurs heuristiques permettent de résoudre le problème de coloration d'un graphe. Selon la structure et certaines caractéristiques de celui-ci, l'heuristique choisi sera plus ou moins efficace [2]. On s'aperçoit aussi que l'on ne sait pas déterminer à l'avance quel sera le meilleure algorithme à utiliser. Ce problème de sélection est une question récurrente en informatique. Elle a été formalisée pour la première fois par Rice [3]. Depuis cette formulation plusieurs types de méthodes, sous différents noms et dans différents domaines ont tenté de répondre à cette question. Les techniques d'apprentissage sont les plus utilisées, quelles soient en amont de la résolution(offline) ou à la volée(online). Ces techniques tentent d'apprendre les caractéristiques d'un ou plusieurs problèmes afin de prédire quel sera le meilleur algorithme pour maximiser la qualité des solutions tout en minimisant le temps de résolution.

Cependant la plupart de ces approches nécessitent un apprentissage. Il peut être effectué à priori, mais nécessite beaucoup de données et notamment des données représentatives. Par ailleurs, il est impossible d'obtenir des données représentant efficacement tous les problèmes. Il existe une autre approche d'ap-

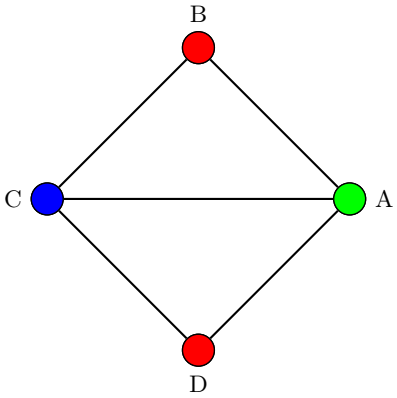


FIGURE 1 : Exemple d'une coloration possible d'un graphe avec 3 couleurs

prentissage qui est effectuée à la volée, mais, bien souvent celle-ci perd beaucoup de temps face aux meilleurs algorithmes. Comment peut on améliorer les performances de sélection d'algorithmes dans le contexte de la programmation par contraintes, tout en minimisant le temps d'apprentissage ?

Nous voyons dans une première partie les concepts préliminaires, dans une seconde partie nous faisons l'état de l'art des méthodes existantes. Nous introduisons ensuite dans une troisième partie une nouvelle méthode d'estimation du temps de résolution ainsi que certains des paramètres d'un CSP et dans une quatrième partie nous présenterons une méthode de sélection d'algorithme.

2 Concepts préliminaires

2.1 Programmation par contraintes

La programmation par contraintes est une technique de résolution de problèmes. Celle-ci s'effectue en deux étapes : la modélisation et la résolution du modèle par un solveur.

Pour modéliser un problème nous avons à notre disposition plusieurs outils :

— les variables qui peuvent représenter différentes entités, par exemple la couleur associée d'un graphe lorsque l'on cherche à le colorier. De façon générale une variable représente un élément pour lequel on cherche à déterminer une ou plusieurs valeurs. Une variable est définie sur un domaine, c'est à dire l'ensemble de ses valeurs possibles. Elles peuvent être entières (i.e le domaine est composé seulement d'entiers) ou flottantes.

— les contraintes qui permettent d'exprimer des propriétés entre des variables. Dis autrement, une contrainte restreint la liste des valeurs possibles des variables impliquées dans la contrainte.

Il existe 3 types de contraintes :

— les contraintes prédéfinies du solver (e.g. contraintes arithmétiques, de cardinalité,...)

— les contraintes données en extension, autrement dit par l'ensemble des combinaisons autorisées ou interdites
— les contraintes correspondant à des combinaisons de contraintes utilisant les opérateurs logiques ET, OU, XOR, NOT appelées parfois meta-contraintes. En outre, l'utilisateur peut définir ses propres contraintes, en établissant la sémantique de la contrainte, ainsi que l'algorithme de filtrage associé.

Un modèle est caractérisé par un ensemble de variables et un ensemble de contraintes. Une solution consiste à trouver une affectation valide de toutes les variables. En d'autres termes, il faut trouver une assignation de chaque variable telle que toutes les contraintes soient satisfaites.

2.1.1 Notations :

Un modèle en programmation par contraintes est représenté par :
— Un ensemble de variables $X = \{x_1, ..., x_n\}$.
— Un ensemble de domaines $D = \{D(x_1), ..., D(x_n)\}$ où $D(x_i)$ représente l'ensemble fini des valeurs possibles pour la variable x_i.
— Un ensemble C de contraintes.

La modélisation d'un problème n'est qu'une première étape. La seconde consiste à le résoudre. Pour cela un solveur de programmation par contraintes dispose de 3 grands mécanismes : le filtrage, la propagation et le mécanisme de recherche de solution.

2.1.2 Le filtrage :

Chaque contrainte dispose d'un algorithme de filtrage. Celui-ci permet de supprimer les valeurs qui ne sont pas consistantes avec la contrainte. Si une valeur est détectée inconsistante, alors elle est supprimée du domaine de la variable.

2.1.3 La propagation :

Lorsqu'une suppression est effectuée, on appel les algorithmes de filtrage des autres contraintes impliquant cette variable pour éventuellement déduire d'autres suppressions. C'est ce que l'on appelle le mécanisme de propagation.

2.1.4 Le mécanisme de recherche de solution :

Bien souvent, pour résoudre un problème les mécanismes de filtrage et de propagation ne suffisent pas. On doit essayer les valeurs restantes des variables. Ces tentatives sont structurées à travers un arbre de recherche. L'ensemble des branches explorées et restantes à explorer forment ce que l'on appelle l'espace de

recherche. Un arbre de recherche peut être représenté de manière binaire (voir Figure 2) ou d-aire (avec d la taille du domaine de la variable étudiée). Dans un arbre de recherche une arête symbolise une décision. Celle-ci peut représenter une affectation (e.g $x_i = 1$) ou une interdiction(e.g $x_i \neq 1$). A titre de rappel, une affectation ou une interdiction modifie le domaine d'une variable, le mécanisme de propagation est donc appelé.

Nous allons voir un exemple de résolution d'un problème afin de pouvoir illustrer les concepts précédents (l'arbre de recherche correspondant est visible sur la Figure 2). Considérons un modèle contenant deux variables x_0 et x_1. On a $D(x_0) = \{1, 2\}$ et $D(x_1) = \{2, 3\}$. La valeur de chaque variable doit être différente et leurs sommes doivent être supérieure ou égal à 5. Ce qui correspond aux contraintes suivantes :

— AllDiff(x_0, x_1)
— $x_0 + x_1 \geqslant 5$

Lorsqu'un solveur résout un modèle, il y a toujours une phase de propagation initiale. Son but est de vérifier si toutes les valeurs sont consistantes avec les contraintes. Dans notre modèle simpliste, cette étape ne supprime aucune valeur. Elles sont toutes consistantes, rien ne peut être déduit. Une valeur doit donc être essayée. Par exemple nous affectons x_0 à 1. Suite à cette décision, aucune déduction ne peut être faite, nous continuons donc l'exploration de notre espace de recherche. Nous essayons d'affecter x_1 à 2. Celle-ci conduit à un échec, la contrainte $x_0 + x_1 \geqslant 5$ est violée. La décision d'affecter x_1 à 2 est remise en cause [1]. Une nouvelle contrainte temporaire (valable uniquement pour le sous-arbre correspondant à l'affectation x_0) est introduite : $x_1 \neq 2$. Le domaine de la variable x_1 est modifié, la seule valeur restante pour x_1 est 3. Malheureusement cette affectation viole à nouveau la contrainte $x_0 + x_1 \geqslant 5$. Toutes les décisions jusqu'à $x_0 = 1$ doivent être remises en cause. Puisque l'exploration du sous arbre correspondant à $x_0 = 1$ est terminé. L'étude de l'espace de recherche se poursuit avec le sous arbre correspondant à $x_0 \neq 1$. En ajoutant la contrainte $x_0 \neq 1$, x_0 est affecté avec 2, celle-ci étant l'unique valeur restante de son domaine. Le filtrage de la contrainte de différence, permet de supprimer 2 du domaine de x_1. La seule valeur possible restante pour x_1 étant 3, x_1 vaut 3. Les affectations de x_1 à 2 et de x_3 à 3 ne violent aucune contraintes. la solution trouvée est donc retournée (noeud vert dans l'arbre). Toutes les valeurs possibles ont été essayées, l'exploration de l'espace de recherche est terminée et complète.

Cet exemple simpliste expose les mécanismes intervenant dans la résolution d'un problème par un solveur. Cependant, il en existe un autre ayant une influence considérable sur le temps de résolution d'un problème, ce sont les stratégies de choix de variables et de valeurs (voir tableau 1). Une stratégie permet

1. Le terme backtrack peut être utilisé, c'est celui que l'on rencontre le plus dans la littérature.

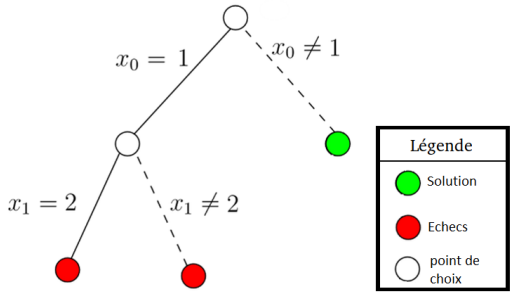


FIGURE 2 : Exemple d'un arbre de recherche binaire.

de définir la prochaine variable et de la prochaine valeur à sélectionner. Le choix d'une stratégie n'est pas fait par le solveur, il n'existe aucune méthode automatique pour le faire. La stratégie employée doit être explicitement définie par l'utilisateur lorsqu'il construit un modèle. Il existe diverses stratégies classées en 2 catégories : les stratégies statiques qui correspondent à celles dont l'ordre et la valeur instanciée des variables sont connus avant la résolution. On peut citer par exemple la stratégie qui sélectionne la variable la plus contrainte (c'est à dire celle qui est présente dans le plus de contraintes), et qui lui affecte la plus petite valeur de son domaine. Par abus de langage, on pourra appeler cette stratégie *most_ constrained_ indomain_ min*. La deuxième catégorie de stratégie est celle des stratégies dynamiques : l'ordre des variables est choisie dynamiquement durant l'exploration de l'espace de recherche. La stratégie qui sélectionne la variable avec le plus petit domaine (*first_fail*) est dynamique. La prochaine variable sélectionnée est inconnue car les décisions précédentes ont une grand influence sur les domaines courant des variables.

Stratégie / Problème	$Most\_constrained\_indomain\_min$	$first\_fail\_indomain\_max$	$anti\_first\_fail\_indomain\_max$
Règle de golomb	15 h	1 j	4 j
Tank attack puzzle	9j	12h	2j
Sport scheduling	50m	1h	20m

TABLE 1 : Illustration des différences de performances entre les stratégies

2.2 Méthodes parallèles en programmation par contraintes

Pour abstraire et simplifier la réalité, le terme worker dénotera une unité de calcul. En règle générale cette unité correspondra à un coeur. Un algorithme exécuté par un unique worker à la fois est appelé un algorithme séquentiel. Si plusieurs workers exécutent un algorithme alors c'est un algorithme parallèle. L'étude des algorithmes séquentiels s'effectue depuis l'antiquité, avec par

exemple l'algorithme d'Euclide. L'accélération d'un programme en améliorant un algorithme séquentiel devient plus difficile, c'est pourquoi, d'autres méthodes sont utilisées telle que la parallélisation des programmes. L'idée consiste à ajouter des workers afin d'accélérer le programme. L'efficacité d'un programme parallèle se mesure par l'accélération obtenue par rapport au meilleur algorithme séquentiel.

$$\text{Accélération} = \frac{T_{\text{séquentiel}}}{T_{\text{Parallèle}}}$$

Une parallélisation parfaite d'un programme produit une accélération équivalente aux nombre de workers utilisés.

Les méthodes habituelles de parallélisation de problèmes en programmation par contraintes sont basées sur le partage dynamique du travail (Work Stealing). Le principe de cette méthode est simple : lorsqu'un worker a fini son travail, il en demande à un autre. Si ce dernier dispose d'assez de travail, alors il le coupe en deux et lui en donne un morceau. Cette méthode a été implémentée dans de nombreux solveurs (Comet [4] ou ILOG [5] par exemple), et de différentes manières [6,7,8,9]. Cependant celle-ci rencontre de nombreuses difficultés : elle ne passe pas bien à l'échelle, certains problèmes résolus en séquentiel ne le sont plus en parallèle. Il y a un problème de terminaison. À la fin de la résolution, les workers passent la plupart de leur temps à découper leur problème et à communiquer. Cela vient en partie du fait que le travail restant est trop petit pour être partagé. Lorsqu'un worker reçoit un travail trop petit, celui-ci est résolu quasi instantanément, entraînant une autre demande de travail et l'arrêt d'un nouveau worker pour lui en fournir. Ce problème ne peut pas être résolu pour l'instant, car l'évaluation la taille d'un sous arbre de recherche pose problème. Donc estimer la taille, le temps de résolution et d'autres paramètres d'un sous problème (i.e un sous arbre) est difficile.

La méthode Embarassingly Parallel Search (EPS) [10,11] voit les choses différemment, plutôt que de tenter d'avoir des sous-problèmes équivalents pour avoir une charge de travail équilibrée (et donc pour un partage dynamique du travail, réduire les communication), l'idée est de décomposer le problème en un grand nombre de sous-problèmes par worker (e,g 40 sous-problèmes par worker), afin d'équilibrer la charge de travail entre les workers. Pour cela, EPS comporte deux parties : la décomposition du problème initial en un grand nombre de sous problèmes et leur résolution. Les sous-problèmes peuvent être vus comme le sous arbre commençant à un point de choix (i.e un noeud dans l'arbre). Tous les sous-problèmes générés doivent être consistant avec la propagation. Pour effectuer la décomposition, une Depth-bounded Depth First Search est effectuée. C'est à dire une DFS bornée par une profondeur maximale. La profondeur d'arrêt pour la génération des sous-problèmes est estimée par le produit cartésien des domaines des variables que l'on va affecter (points de choix). Lorsque le nombre de sous problèmes requis est atteint, ils sont placés dans une queue afin d'être résolus dans la seconde phase de la méthode. Lorsqu'un worker aura besoin de travail, il ira chercher un sous-problème dans la queue. Il est facilement démontrable que

le temps d'attente maximum d'un worker est borné par le temps du plus long sous problème.

EPS est une méthode simple et efficace. Il a été démontré à travers des expérimentations que cette méthode passait très bien à l'échelle (testée jusqu'à 512 workers sur le centre de calculs de Nice).

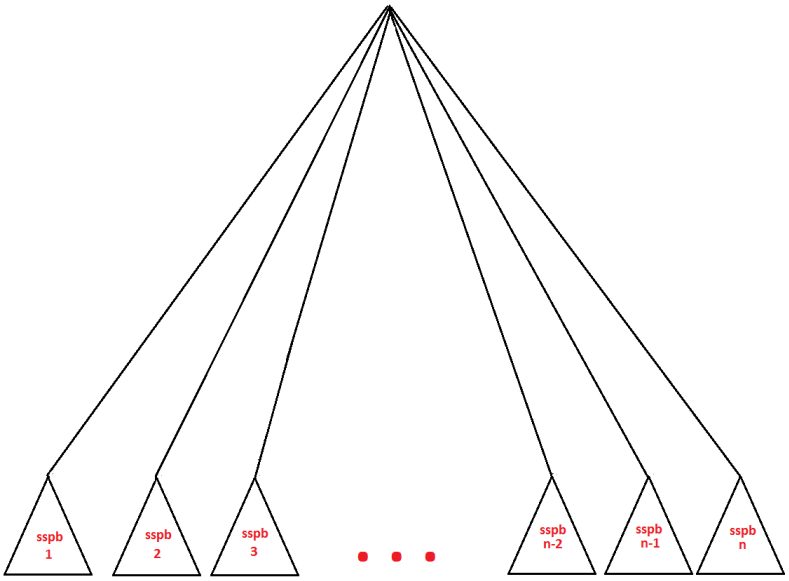


FIGURE 3 : EPS : décomposition de l'espace de recherche.

2.3 Théorie de l'échantillonnage

On appelle population la totalité des individus (ou unité statistique) sur lesquels des méthodes et des techniques de présentation, de description et d'inférence statistiques sont appliquées. Une population peut être finie ou infinie. En règle générale, il est impossible de l'analyser complètement pour des raisons de coûts et/ou de temps. Une technique permettant d'évaluer les caractéristiques de la population sans pour autant devoir la considérer entièrement doit être utilisée. Pour cela, un échantillonnage est effectué, il permet d'obtenir un sous-ensemble de la population étudiée appelé échantillon. Un bon échantillon (« sans biais [2] »), doit être représentatif de la population dont il est issu (notamment en terme de moyenne, variance ...). Bien entendu, les paramètres varient d'un échantillon à l'autre, il y a des fluctuations d'échantillonnage.

2. un biais est une démarche ou un procédé qui engendre des erreurs dans les résultats d'une étude.

Différentes méthodes adaptées à divers contextes existent pour effectuer un échantillonnage. Nous pouvons citer par exemple l'échantillonnage : par grappe, systématique, stratifié ou aléatoire. Nous allons nous intéresser plus particulièrement à l'échantillonnage aléatoire. En effet, celui-ci concerne surtout les cas où l'on cherche à estimer des paramètres d'une population dont les caractéristiques sont totalement inconnues (e.g proportion et représentativité). Dans un échantillonnage aléatoire, tous les échantillons possibles de même taille ont la même probabilité d'être sélectionnés. De même tous les éléments de la population ont la même chance d'appartenir à l'échantillon. Un échantillon obtenu par cette méthode peut être considéré comme le résultat d'une expérience aléatoire et donc qui peut être caractérisé par une variable aléatoire. Cette variable possède bien évidemment une distribution de probabilité, une moyenne, une variance ...

A titre de rappel[3], en général l'estimation d'une moyenne est effectuée à l'aide, de l'estimateur empirique de la moyenne qui est sans biais. Son calcul s'effectue avec la formule suivante :

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

De même l'estimation la variance utilise l'estimateur sans biais suivant :

$$\widehat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{1} (x_i - \overline{x})^2$$

L'analyse d'un échantillon nous conduit à analyser la distribution de sa moyenne. Deux cas sont à distinguer : les petits et les grands échantillons. En pratique un échantillon avec au moins 30 observations est considéré comme un grand échantillon. Il est important de noter que d'après la loi des grands nombres, les caractéristiques d'un échantillon aléatoire se rapprochent d'autant plus des caractéristiques statistiques de la population que la taille de l'échantillon augmente. En d'autres termes, l'estimation des paramètres converge vers les vraies valeurs de la population à mesure que le nombre d'observations augmente. La taille de l'échantillon à considérer pour approcher les caractéristiques de la population ne dépend que faiblement, voire pas du tout, de la taille de celle-ci. Par exemple, si l'on souhaite effectuer un sondage au Luxembourg ou aux États-Unis, pour obtenir des précisions égales, les échantillons seront de même taille. Pour notre étude, seul le cas des grands échantillons nous intéressera ($n \geqslant 30$).

Les grands échantillons sont très intéressant car ils nous permettent d'appliquer le Théorème de la Limite Centrale (appelé TLC ou TCL en anglais). Ce théorème nous dit que la distribution d'un grand échantillon peut toujours être ramené à une loi normale centrée réduite. Cette propriété remarquable, élargit considérablement le champs d'application des tests que nous allons présenter. La restriction imposée par l'hypothèse de normalité sous-jacente de X est levée.

3. Plus d'informations concernant les estimateurs et leur propriétés mathématiques peuvent être trouvées sur `http://www.math.u-bordeaux1.fr/~mchabano/Agreg/ProbaAgreg1213-COURS2-Stat1.pdf`

2.3.1 Théorème de la limite centrale (TLC) :

Soient $X_1, X_2, ..., X_n$ des variables aléatoire (v.a) indépendantes de même loi de répartition d'espérance μ et de variance σ^2.
Alors la v.a : $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ a :

- une moyenne égale à : $\mathbb{E}(\bar{X}) = \mu$
- une variance égale à $V(\bar{X}) = \frac{\sigma^2}{n}$

Concernant les distributions :

- si X $\sim \mathcal{N}(\mu, \sigma)$ alors la loi exacte de $\bar{X}$ est $\mathcal{N}(\mu, \frac{\sigma}{\sqrt{n}})$,
- si X suit une loi quelconque, en vertu du théorème central limite, lorsque $n \to +\infty$, alors $\bar{X}$ tend vers la loi normale. En pratique dès que n $\geqslant 30$, l'approximation devient effective. Dès que n est suffisamment élevé la quantité Z avec :

$$Z = \frac{\overline{X}-\mu}{\frac{\sigma}{\sqrt{n}}} \sim N(0,1) \quad \textbf{quelle que soit la loi sous-jacente de X.}$$

2.4 Tests statistiques

Il est important de noter que les tests statistiques procurent des techniques et des outils permettant de bien prendre en compte la variabilité (variance) des observations. Un test statistique consiste à vérifier une information hypothétique. C'est pour cela que l'on peut parfois entendre parler de tests d'hypothèses. L'information hypothétique à tester concerne la population à laquelle on s'intéresse. On peut par exemple vérifier la distribution d'une variable, la valeur d'une estimation d'un paramètre d'une population (e.g moyenne, variance...) à laquelle elle serait égale. Mais aussi, un intervalle de valeurs auquel appartiendrait la valeur d'une statistique, l'indépendance statistique de deux variables.

Un test peut aussi être utilisé, pour tenter d'évaluer l'impact positif (ou négatif) d'une action entreprise pour modifier la valeur d'une statistique.

Nous allons présenter un test d'hypothèse afin d'introduire le schéma d'utilisation des tests ainsi que tous les concepts nécessaires.

2.4.1 Tests d'hypothèses : exemple introductif

Nous venons de développer un nouveau solveur de programmation par contraintes. Nous voulons savoir si ce dernier est plus efficace globalement que son concurrent. Pour cela, nous avons à notre disposition une base de données contenant toutes les instances résolues (avec toutes les caractéristiques nécessaires pour reproduire les expériences) par le solveur concurrent. Comme nous ne pouvons pas nous comparer à toutes les instances, nous prélevons un échantillon de manière aléatoire par classe de problèmes. Nous analysons un échantillon du solveur concurrent au hasard dans une classe (voir tableau 2). La moyenne des problèmes résolus dans cette classe est de 600 secondes. La variance de la population quant à elle est inconnue.

569	361	555	747	914	507	333	436	886	638	85	897	264	159	798
188	193	437	692	58	608	850	851	552	452	730	97	745	951	36

TABLE 2 : Echantillon des temps de résolution pour résoudre un problème

La question que nous pouvons nous poser est de savoir si cet échantillon est représentatif des problèmes résolus en terme de temps de résolution moyen. Nous allons réaliser un premier test afin de comparer la moyenne observée de l'échantillon et la moyenne théorique. Bien évidemment juger de la qualité d'un solveur dépend de plus de paramètres que de la moyenne des temps de résolution, les problèmes utilisent beaucoup de concepts et d'algorithmes différents. D'autres paramètres doivent être pris en compte pour évaluer la valeur d'un solveur.

La première chose à réaliser lorsque l'on souhaite effectuer un test est de calculer les paramètres de l'échantillon avec les estimateurs sans biais précédemment énoncés. Pour rappel, nous avons sélectionné 30 problèmes (voir Tableau 2). On estime donc la moyenne :

$$\overline{x} = \frac{569 + 361 + 555 + ... + 745 + 951 + 36}{30} = 519.63$$

et son écart type (racine carré de la variance) :

$$\widehat{\sigma} = 289.04$$

La seconde étape consiste à formuler les hypothèses. Celle que nous voulons vérifier sera appelée hypothèse nulle et est notée H_0. Pour en revenir à notre exemple nous souhaitons vérifier si la moyenne observée est égale statistiquement à la moyenne théorique (la moyenne théorique de la population totale étant de 600 secondes). Nous posons alors :

$$H_0 : \mu = 600$$

où μ représente la moyenne du temps de résolution d'un problème. Nous rassemblerons d'autre part, l'ensemble des hypothèses alternatives sous H_1 :

$$H_1 : \mu \neq 600$$

.

Nous parlerons de tester H_0 contre les alternatives bilatérales H_1 (sous H_1, μ peut être inférieur ou supérieur à 600).

Un test bilatéral (voir Figure 4) s'applique quand on cherche une différence entre deux estimations, ou entre une estimation et une valeur donnée sans se préoccuper du signe ou du sens de la différence. Dans ce cas, la zone de rejet de l'hypothèse principale se fait de part et d'autre de la distribution de référence.

Un test peut aussi être unilatéral à gauche ou à droite. Celui-ci s'applique quand on cherche à savoir si une estimation est supérieure (ou inférieure) à une autre, ou à une valeur donnée. La zone de rejet de l'hypothèse principale est

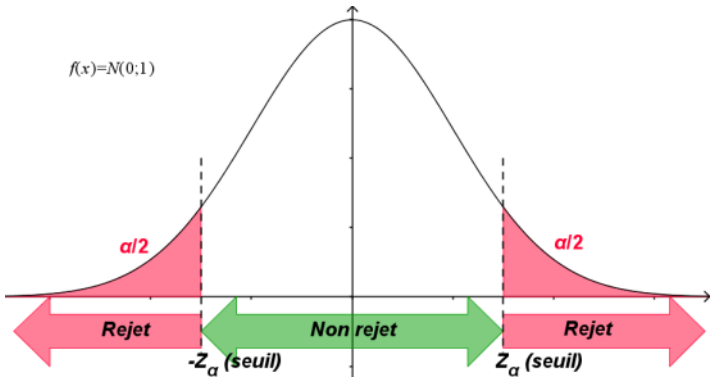


FIGURE 4 : Illustration d'un test bilatéral

alors située d'un seul côté de la distribution de probabilité de référence (voir Figure 5). Une zone de rejet est définie par rapport à un niveau de signification nominal α (appelé aussi niveau de signification réel [4]). Ce niveau de signification définit la probabilité (le risque) de faire une erreur. Le niveau de signification est la probabilité de rejeter l'hypothèse nulle à tort. Deux sortes d'erreurs sont possibles :

- Les erreurs de première espèce consistant à rejeter l'hypothèse nulle alors qu'elle est vraie. Ces erreurs se produisent si la valeur de la statistique de test tombe dans la région de rejet alors que l'hypothèse H_0 est vraie.
- Les erreurs de seconde espèce consistent à ne pas rejetter l'hypothèse nulle alors qu'elle est fausse. Ce type d'erreur se produit si la valeur de la statistique de test tombe dans la région de non rejet (ou d'acceptation) alors que H_0 est fausse (i.e H_1 est vrai).

Pour la suite de cet exemple, nous allons prendre $\alpha = 5\%$.

Une fois les hypothèses de test posées, nous devons choisir la statistique du test. C'est en comparant la valeur de cette statistique observée dans l'échantillon à sa valeur sous l'hypothèse H_0 que nous pourrons prendre une décision (i.e donner la conclusion du test). Il existe de nombreuses statistiques adaptées aux différentes situations. La statistique de Student est la plus adaptée à notre contexte et à notre objectif de comparaison de moyenne observée et théorique. Les conditions d'utilisation de cette statistique sont respectées à savoir une distribution normale de la moyenne, acquise en vertu du TLC.
Le calcul de la statistique de student est donné par la formule suivante :

$$t = \frac{\bar{x} - \mu}{\sigma/\sqrt{n}}$$

.

4. On pourra voir le terme p-value dans la littérature.

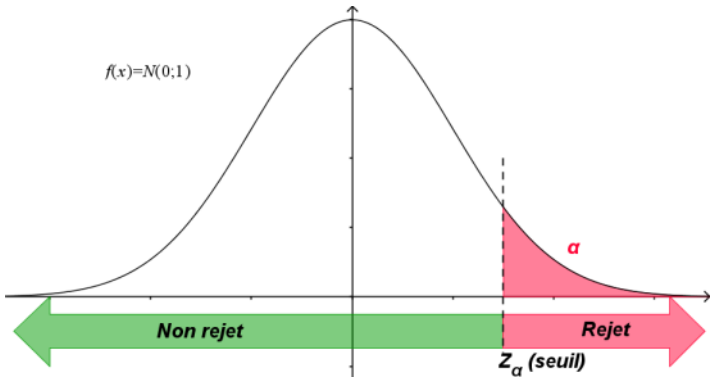


FIGURE 5 : Illustration d'un test unilatéral droite

avec $\overline{x}$ la moyenne observée (i.e celle de l'échantillon), μ la moyenne théorique, et σ l'écart type. Si celui-ci est inconnu, il peut être remplacé par son estimateur $\widehat{\sigma}$. Dans notre cas, le calcul de la statistique nous donne :

$$|t| = |\frac{519.63 - 600}{289.04/\sqrt{30}}| = 1.523$$

Pour savoir si la différence est significative, il faut tout dabord lire dans la table de student(voir Annexe A), la valeur critique correspondant au risque $\alpha = 5\%$ pour un degré de liberté (d.d.l). La valeur du degré de liberté est défini par :

$$d.d.l = n - 1$$

où n le nombre d'éléments dans l'échantillon. Dans notre cas, le degré de liberté vaut 29. Pour $1 - \alpha = 95\%$, la valeur critique lu est de 2.045.

Si la valeur absolue de t ($|t|$) est supérieure à la valeur critique, alors la différence est significative donc on rejette l'hypothèse H_0. De même pour un cas unilatéral gauche si $t_{\text{calculé}} < -t_{lu}$ ou si $t_{\text{calculé}} > t_{lu}$ pour un test unilatéral droit, on rejette H_0.

Dans notre cas, comme le t calculé est inférieur à la valeur critique, on accepte H_0. La valeur de la moyenne de notre échantillon est statistiquement égale à la moyenne théorique. Donc, en terme de temps moyen de résolution, les problèmes tirés au hasard sont représentatif de la population. Nous savons maintenant que nous disposons d'un échantillon représentatif. Nous allons résoudre chaque problème avec notre nouveau solveur afin de pouvoir les comparer avec son concurrent à l'aide d'un autre test statistique.

Le Tableau 3 montre les problèmes résolus avec notre nouveau solveur. Globalement définir quel est le meilleur solveur est un choix difficile. Par exemple, pour le problème 13 (ils ne sont pas numérotés, les nombres correspondent à la 13e case en partant de la gauche) le solveur concurrent est plus rapide de 267 secondes alors que pour le 14 il est plus lent de 176 secondes. Comment peut on évaluer si les différences sont significatives et quels sont les bons critères pour

756	481	525	877	1091	532	490	483	878	722	196	917	422	10	914
104	319	521	715	104	797	971	653	500	416	718	15	660	925	77

TABLE 3 : Temps de résolution de notre solveur sur l'échantillon.

comparer nos deux échantillons ? Pour cela, nous allons à nouveau réaliser un test de student.

2.4.2 Test de comparaison de deux échantillons

Ces tests permettent de comparer des résultats obtenus pour une variable, sur deux groupes d'observations, en vue de déterminer s'ils sont significativement différents d'un groupe à l'autre. Il peut s'agir, par exemple, d'un test de deux packagings ou de deux messages publicitaires, en vue d'évaluer la version la plus appréciée par les personnes interrogées. Les tests paramétriques de comparaison les plus fréquents sont les tests de différence entre deux moyennes ou entre deux pourcentages. Le premier s'applique sur des variables numériques. Il peut porter sur des échantillons indépendants ou appariés. A titre d'exemple, si on fait goûter une boisson à un groupe de femmes et à un groupe d'hommes pour voir s'il y a une différence d'appréciation selon le sexe, on réalise là un test sur des échantillons indépendants. En revanche, si on fait goûter deux boissons différentes à un même groupe d'individus, pour voir s'il y a une préférence significative pour l'une des deux, il s'agit d'une mesure sur des échantillons appariés.

Dans le premier cas, le test compare la moyenne pour le 1er et pour le 2ème groupe puis cherche à évaluer si cette différence est significativement différente de 0. Si tel est le cas, on peut considérer que les hommes n'apprécient pas la boisson de la même manière que les femmes. Pour savoir quel groupe l'apprécie le plus, il n'est pas forcément besoin de choisir que le test se fasse de manière unilatérale puisqu'il suffit de jeter un coup d'oeil sur les moyennes.

Dans le deuxième cas, le test consiste à calculer les différences entre les 2 notes données par chaque individu aux produits testés. Ensuite le test calcule la moyenne de ces différences puis essaie de voir si cette moyenne est significativement différente de 0. Si tel est le cas, on peut conclure que les produits sont notés de manière différente. Là aussi, l'appréciation du meilleur produit peut se faire par l'examen de la moyenne de chacun des deux ou alors, en demandant au départ un test unilatéral.

Nous n'allons présenter que le cas pour des échantillons appariés.

2.4.3 Test de comparaison des moyennes pour échantillons appariés :

L'objectif de l'appariement est de réduire la variabilité due aux observations. Prenons un exemple simple pour expliciter l'idée. Un industriel affirme que son additif pour essence permet de réduire la consommation des automobiles. Pour

vérifier cette assertion, nous choisissons au hasard n_1 véhicules, nous leur faisons emprunter un parcours routier, nous notons la consommation de chaque véhicule. Puis nous extrayons un second échantillon de n_2 observations, nous rajoutons l'additif dans le réservoir, sur le même parcours routier, nous mesurons les consommations. Pour tester la réduction de la consommation, nous confrontons les deux moyennes observées x_1 et x_2. Nous sommes dans un schéma de test sur échantillons indépendants dans ce cas. En y regardant de plus près, on se rend compte qu'il y a des éléments non maîtrisés dans notre expérimentation. Avec un peu de (mal)chance, il se peut que les petites berlines soient majoritaires dans le premier échantillon, les grosses berlines dans le second. Cela faussera totalement les résultats, laissant à penser que l'additif a un effet néfaste sur les consommations. Le principe de l'appariement est d'écarter ce risque en créant des paires d'observations. Dans notre exemple, nous choisissons en effet n véhicules au hasard dans la population : nous leur faisons faire le trajet normalement une première fois, puis nous rajoutons l'additif dans réservoir et leur faisons parcourir le même chemin. L'écart entre les consommations sera un bon indicateur des prétendus bénéfices introduits par l'additif. Ce schéma "avant-après" est la forme la plus populaire de l'appariement. Elle permet de réduire le risque de second espèce du test c.-à-d. nous augmentons la puissance du test. L'appariement est en réalité plus large que le seul schéma "avant-après". Il est efficace à partir du moment où nous réunissons les deux conditions suivantes : les individus dans chaque paire se ressemblent le plus possible, ou appartiennent à une même entité statistique (un ménage, des jumeaux, etc.) ; les paires d'observations sont très différentes les unes des autres. Nous allons voir comment nous pouvons utiliser un test pour échantillons appariés afin de comparer nos deux solveurs.

2.4.4 Test d'hypothèses sur des échantillons appariés, statistique du test et région critique

Revenons maintenant à nos comparaisons de solveurs. Nous allons effectuer un test sur des échantillons appariés. Nous pouvons considérer que ces deux échantillons sont appariés puisque seul le solveur change. Nous gardons les mêmes conditions de résolution (machine, problème résolu ...). Nous considérons maintenant nos deux échantillons contenant les 30 problèmes résolus, que l'on peut regrouper par paire. Nous formons alors une nouvelle variable aléatoire D dont les valeurs d_i sont obtenues par différences des paires de valeurs :

$$d_i = x_{1i} - x_{2i}$$

Pour x_1 le solveur concurrent et x_2 le notre, les différences obtenues sont les suivantes :

X étant gaussienne, D l'est également. Nous savons de plus que $\mathbb{E}(D) = \mu_D = \mu_1 - \mu_2$. Le test de comparaison de moyennes pour échantillons appariés s'écrit dès lors (pour un test unilatéral droit) :

$$H_0 : \mu_D \leqslant 0$$

$$H_1 : \mu_D > 0$$

-187	-120	30	-130	-177	-25	-157	-47	8	-84	-111	-20	-158	149	-116
84	-126	-84	-23	-46	-189	-121	198	52	36	12	82	85	26	-41

TABLE 4 : Différence des temps de résolution pour chaque problème.

Il s'agit ni plus ni moins que d'un test de conformité de la moyenne à un standard à partir d'un échantillon. A travers ces hypothèses, on cherche à vérifier si le solveur concurrent est plus efficace. Notons $\bar{d}$ la moyenne empirique, avec

$$\bar{d} = \frac{1}{n} \sum_{i=1}^{n} d_i$$

Dans notre cas, $\bar{d} = -40$

L'estimation de la variance de D à partir d'un échantillon s'écrit :

$$V(D) = \sigma_D^2 = \sigma_{X_1 - X_2}^2 = \sigma_1^2 + \sigma_2^2 - 2 \times COV(X1, X2)$$

.

Les échantillons sont appariés, les variables aléatoires ne sont donc pas indépendantes. La variance ne peut pas se résumer à la somme des variances des variables individuelles. Il faut prendre en compte la covariance entre les deux observations.

Comme nous devons estimer les variances, nous passons directement par l'estimation $\widehat{\sigma}_D^2$ à partir des observations d_i avec :

$$\widehat{\sigma}_D^2 = \frac{1}{n-1} \sum_{i=1}^{n} (d_i - \bar{d})$$

Dans notre cas, $\widehat{\sigma}_D = 101.89$

Sous H_0, la statistique du test s'écrit :

$$t = \frac{\bar{d}}{\widehat{\sigma}_D / \sqrt{n}}$$

Elle suit une loi de Student à (n - 1) degrés de liberté.

On calcule le t correspondant :

$$t = 2.15$$

Nous allons vérifier notre hypothèse pour un risque $\alpha = 5\%$. Comme nous sommes dans le cas d'un test unilatéral, les choses sont un peu différentes. Pour le cas d'un test bilatéral, nous avons de part et d'autre un risque $\frac{\alpha}{2}$, ce qui n'est plus le cas pour un test unilatéral, nous devons regarder la même table mais cette fois ci en calculant Nous rejetons l'hypothèse nulle si

$$t_{\text{calculé}} > t_{lu}$$

Nous lisons la valeur 1.699 (voir 7) qui correspond à un risque $\alpha = 5\%$ et à un d.d.l = 29. Comme $t_{\text{calculé}} > t_{lu}$ l'hypothèse H_0 est rejetée. Mais on se rend compte que si le test inverse est effectué c'est à dire que hypothèse posée est que notre solveur est plus efficace que le concurrent, nous refuserons encore l'hypothèse nulle. Aucune déduction ne peut être faite à l'aide d'un test statistique avec une marge d'erreur assez faible. Les deux solveurs sont statistiquement équivalent avec un risque d'erreur de 5%. Nous venons de voir à travers un exemple, des cas d'utilisation de tests statistiques plus particulièrement celui de student.

3 Estimation des paramètres d'un arbre de recherche et du temps de résolution d'un problème

Dans un arbre de recherche, plusieurs paramètres peuvent être analysés, cela peut être le nombre de noeuds, d'échecs et même d'une manière plus générale, le temps d'exécution de l'algorithme. Dans cette partie, nous allons voir les principales méthodes utilisées pour estimer la taille d'un arbre de recherche et le temps d'exécution, car malheureusement à l'heure actuelle aucune méthode ne permet d'estimer le nombre d'échecs dans un arbre de recherche. Nous introduisons ensuite une nouvelle méthode permettant d'estimer tous ces paramètres d'un arbre de recherche ainsi que le temps de résolution d'un problème en programmation par contraintes.

3.1 État de l'art

L'évaluation du temps d'exécution d'un algorithme est une question assez importante. Tout programmeur s'est déjà au moins une fois posé cette question lors du lancement d'un de ses algorithmes. Cependant, peu de travaux ont été effectués sur ce sujet. Seul Hutter et al. [12] ont proposé une méthode offline basée sur des prédictions obtenues grâce à des Machines Learning (ML). En utilisant un algorithme offline, un modèle est construit à partir d'un ensemble d'apprentissage. Cependant, cela sous-entend que l'ensemble d'apprentissage est représentatif de tous les problèmes existants (même pour un seul problème ce n'est pas évident surtout s'il n'a jamais été résolu par le passé) et que le modèle sait caractériser tous les problèmes. Concernant l'estimation de la taille d'un arbre de recherche, la plupart des méthodes d'estimation se fondent sur l'algorithme de Knuth [13] qui échantillonne une sous-partie de l'arbre. L'idée de cet algorithme est d'effectuer une marche aléatoire afin d'estimer le nombre de noeuds. Néanmoins, l'estimation d'un arbre de recherche par cet algorithme suppose que celui-ci est équilibré ce qui rarissime en pratique. Chen [14] a amélioré cet estimateur en effectuant un échantillonnage stratifié. Plus récemment, Kilby et al. [15] ont proposé deux méthodes online. La première est basée sur la pondération des échantillons en fonction de leur chronologie de visite. La seconde, quant à elle, utilise une DFS de la gauche vers la droite dans l'arbre. Pour effectuer une estimation, cette méthode suppose que la partie droite non

explorée de l'arbre de recherche est similaire à la partie gauche explorée. L'inconvénient principal de cet méthode vient du fait que pour obtenir une estimation fiable une certaine proportion de l'espace de recherche doit être parcourue. Par exemple dans [15] cette proportion correspond à 10% de l'espace de recherche selon leurs benchmarks. Ensuite, selon la méthode utilisée, les calculs effectués lors de l'estimation sont difficilement réutilisables pour résoudre le problème, ce qui potentiellement coûte cher, surtout si 10% du temps total de la résolution y est consacré.

3.2 Méthode pour les problèmes de satisfaction

Dans cette sous-partie, nous proposons une méthode permettant d'estimer les divers paramètres d'un arbre de recherche ainsi que le temps de résolution d'un problème en tentant de combler les défauts des méthodes précédentes. Notre nouvelle méthode repose essentiellement sur la théorie de l'échantillonnage (voir section 2.3). Le but est d'estimer les paramètres d'une population. Le principe est donc de voir un problème comme une population. Pour cela, le problème est décomposé en un grand nombre de sous-problèmes grâce à la décomposition d'EPS (chaque sous problème correspond à un individu). Ensuite l'estimation est effectuée en résolvant un certain nombre de sous-problèmes selectionné grâce à un échantillonnage aléatoire. La qualité de l'estimation dépendra du nombre de sous-problèmes résolus (voir Annexe B). Les paramètres observés de cette population peuvent être estimés grâce au calcul de l'estimateur de la moyenne empirique à partir des observations de l'échantillon. La propriété remarquable de cette méthode est acquise grâce à la loi des grands nombres (voir section 2.3). En effet, d'après cette loi, la qualité de l'estimation ne dépend que très peu de la taille initiale de la population et donc pour obtenir une estimation fiable, seul le nombre de sous-problèmes évalués comptera (environ 1 000 en pratique). Par exemple lorsqu'un modèle très difficile doit être estimé, pour éviter de trop gros calculs, une grande décomposition sera avantageuse. L'augmentation du nombre de sous-problèmes diminuera le temps d'évalation de chaque sous problème et donc permettra de diminuer le temps d'estimation. L'idée est donc de décomposer le problème initial en un très grand nombre de sous-problèmes (e.g 10 000 000) afin de réduire le temps d'exécution lié à l'estimation tout en conservant une bonne fiabilité.

3.3 Contexte théorique et application

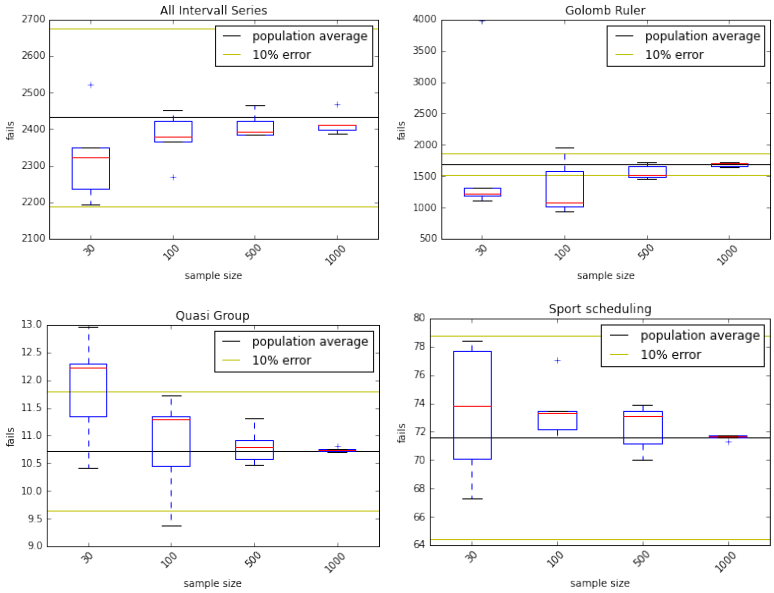
Le contexte rigide d'application de la théorie de l'échantillonnage et certains théorèmes tel que celui de la limite centrale et de la loi des grands nombres obligent d'avoir des variables aléatoires iid (même si des version plus faible de ces théorèmes existent). Cependant dans notre cas, nous allons voir que nous disposons bien de variable indépendantes et identiquement distribuées. Il est important de faire le parallèle avec la théorie des sondages dans l'analyse de notre méthode. En effet cela revient à effectuer par exemple l'analyse de la taille des individus d'une population. Pour en revenir à notre méthode, le nombre de sous

problème généré est fini (on posera sa taille égale à N). L'échantillonnage aléatoire sans remise de taille n (avec n < N) n'est autre que l'équiprobabilité des A_N^n échantillons possibles. Si la sous-population d'intérêt A est de cardinal K, donc en proportion $p = \frac{K}{N}$, la proportion ou fréquence de A dans l'échantillon s'écrit $F = \frac{X}{n}$ avec X hypergéométrique de paramètres N, n et p. La loi hypergéométrique peut être approchée par une loi binomiale de paramètres n et p dès que le taux de sondage $\frac{n}{N}$ est négligeable, dans la pratique inférieur à 1/10). Les deux lois binomiale et hypergéométrique peuvent être approchées par une loi normale dès que n est suffisamment grand (dans la pratique supérieur à 30). Ces dernières conditions sont respectés lors de l'application de notre méthode.

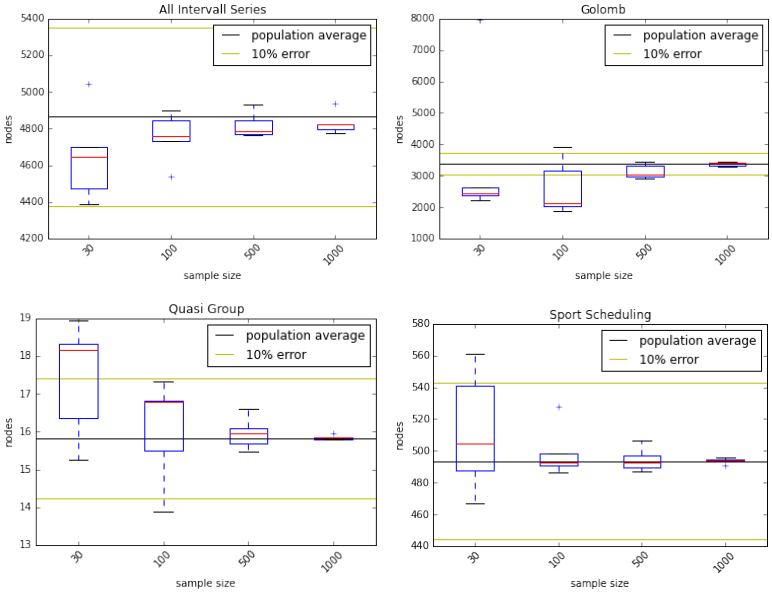
3.4 Validation expérimentale

Pour effectuer ces expérimentations, nous avons résolu plusieurs problèmes provenant de la CPSLIB[16] et de `http://www.hakank.org`. Nous avons ensuite décomposé notre problème en 20 000 sous-problèmes, effectué un échantillonnage et comparé les résultats de nos estimations (de diverses tailles) face à la valeur réelle des paramètres. Dans les graphiques ci-dessous, nous pouvons observer la qualité de notre estimateur, notamment la faible variance observée dans nos estimations et la convergence des estimateurs.

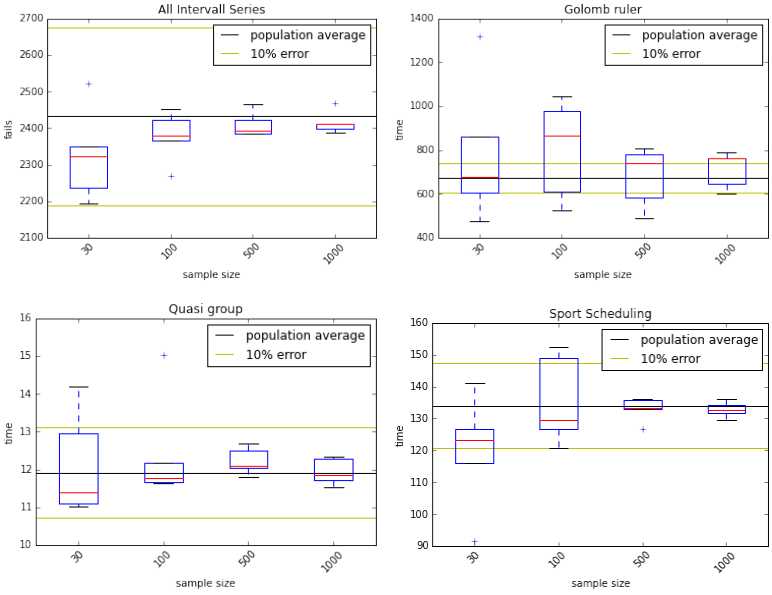
3.4.1 Estimation du nombre d'échecs



3.4.2 Estimation du nombre de noeuds



3.4.3 Estimation du temps de résolution



3.5 Avantages et limitations

Par rapport aux méthodes présentées dans l'état de l'art, notre méthode diffère en plusieurs points. Tout d'abord, à la suite d'une estimation, la résolution déjà entreprise peut-être conservée très facilement afin d'éviter de refaire les mêmes calculs. Le dernier point qui est l'un des plus importants est que la qualité de l'estimation ne dépend que très peu de la taille initiale de la population (voir section 2.3). Par conséquent, lorsqu'un problème très difficile est rencontré, par exemple avec un temps de résolution d'une semaine, nous n'allons pas dépenser un jour complet pour effectuer l'estimation. Simplement, le problème est décomposé en beaucoup plus de sous-problèmes afin d'éviter de trop gros calculs. Le nombre requis de sous-problèmes résolus restera sensiblement équivalent (voir annexe B). Néanmoins, notre méthode possède une limitation. En effet, elle n'est valide que pour les problèmes n'ayant pas d'objectif. Par exemple, si un algorithme de type séparation et évaluation est utilisé pour résoudre un problème, alors pour un sous problème donné, le nombre de noeuds varie selon la valeur courante de l'objectif et donc l'estimateur n'est plus valable dans ce cas (il en est de même pour tous les autres paramètres).

3.6 Méthode pour les problèmes d'optimisation

Nous venons de voir une méthode permettant d'estimer les paramètres d'un arbre de recherche d'un problème de satisfaction. Cependant il existe une autre catégorie de problème : ceux d'optimisation. Ces derniers possèdent une fonction objectif rendant la résolution différente. En effet, la valeur courante de l'objectif (i.e la meilleure solution trouvée jusqu'à un instant donnée) influe sur la manière d'explorer un arbre de recherche. Pour résoudre un problème d'optimisation, un algorithme de type séparation évaluation est utilisé. Avec celui-ci seul les branches pouvant améliorer l'objectif ou possédant des solutions équivalentes sont explorées sont explorées. D'un point de vue statistiques, nos réalisations issue de l'exploration d'un sous problème ne sont plus indépendantes les unes des autres. La manière d'explorer un sous problème dépendra de l'objectif courant et donc des sous problèmes précédemment évalués.

En appliquant directement notre estimateur par exemple sur le problème de la règle de Golomb, on s'aperçoit que les paramètres sont surestimés (voir Figure 6). En effet, l'évolution de l'objectif permet d'éviter d'évaluer certaines branches. En appliquant directement notre estimateur, nous évaluons en faite les paramètres d'un arbre complètement développé, c-a-d en considérant les branches coupées par l'algorithme de séparation et évaluation (i.e le problème d'optimisation a été transformé en problème de satisfaction).

En observant la variation de l'objectif ci dessous (voir Figure 7) et d'une manière plus globale, il est important de remarquer que l'objectif varie beaucoup au début de la résolution. De manière plus intuitive cela peut être traduit comme le fait que les solutions les moins bonne sont plus facile à découvrir (cela peut s'expliquer par leur nombre plus important). Alors que les meilleurs solutions quant à elles sont généralement plus difficile à trouver car plus rare.

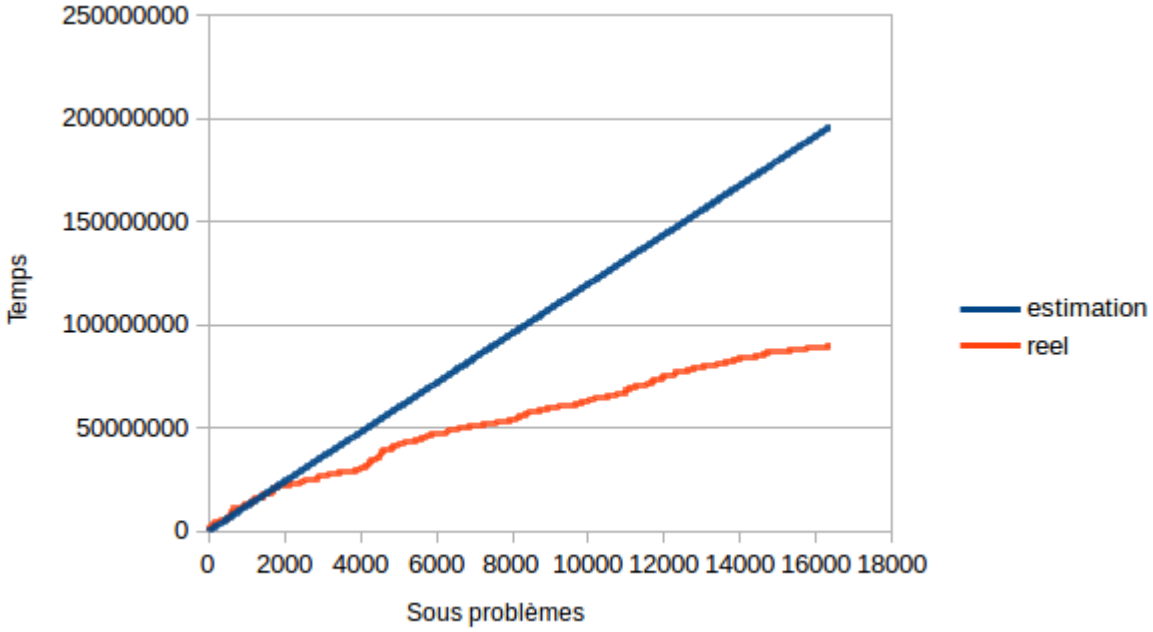


FIGURE 6 : Illustration del'estimation du temps de résolution pour un problème de Golomb avec la méthode pour les problèmes de satisfaction.

—

En faite, les problèmes d'optimisations peuvent être classés en deux catégories selon la variation de leur objectif :

- ceux avec peu de solutions : leur objectif ne varie que très peu. Ils se rapprochent fortement d'un problème de satisfaction.
- ceux où la variation de l'objectif est importante (e.g la règle de Golomb).

Ce sont ces derniers qui posent des problèmes pour effectuer notre estimation. L'idée pour palier ce problème est de se ramener à un problème de satisfaction. Pour adapter notre méthode aux problèmes d'optimisation, notre échantillonnage va se faire en deux passes. La première consiste à évaluer nos sous-problèmes sélectionnés une première fois comme pour un problème habituel de satisfaction. Lors de cette première phase notre objectif évolue (assez rapidement ou non selon le type de problèmes). Pour estimer donc nos paramètres de notre arbre de recherche nous allons les approximer en estimant un problème de satisfaction se rapprochant le plus possible du problème d'optimisation. Cela correspond à notre deuxième passe. Pour cela nous recalculons tous les sous problèmes avec la meilleur valeur de l'objectif. L'idée est donc d'équilibrer les gains et les pertes du à la variation de la valeur de l'objectif (voir Figure 8).

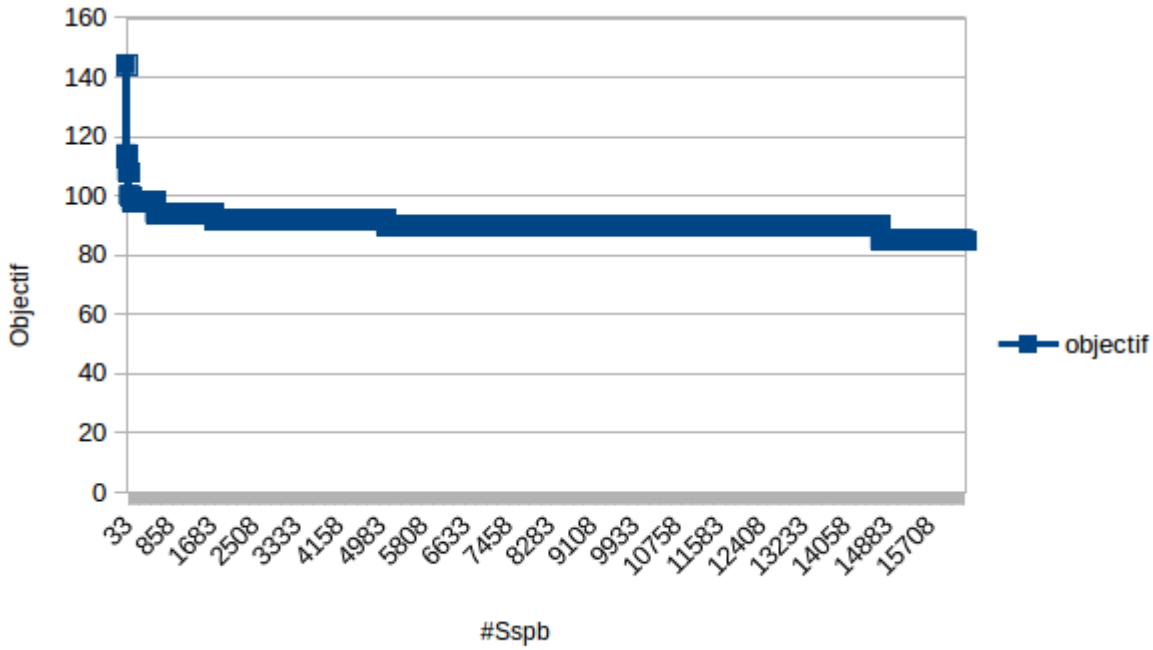


FIGURE 7 : Évolution de la valeur de l'objectif dans le problème de la règle de Golomb.

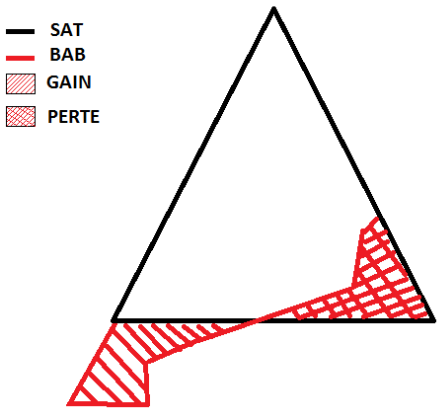


FIGURE 8 : Approximation d'un problème d'optimisation par un problème de satisfaction.

Cette approximation permet de conserver l'indépendance de nos variables aléatoire. Néanmoins, cette estimation n'est pas parfaite et introduit un biais. Celui-ci correspond à la différence entre l'arbre développé par l'algorithme de type séparation évaluation utilisé pour le problème global et un algorithme de type backtrack pour le problème de satisfaction approchant notre problème d'optimisation. Mais nous allons voir dans notre validation expérimentale qu'en règle général il est inférieur à celui crée en utilisant la méthode pour les problèmes de satisfaction.

3.7 Validation expérimentale

Pour notre validation expérimentale, nous nous sommes servi de modèles disponible sur CSPLIB, Minizinc et hakank. Nous allons comparer donc 3 temps : le temps estimé avec la méthode habituelle, l'estimation effectuée grâce à l'approximation par un problème de satisfaction et le temps réel.

Problème	temps réel	estimation habituel	estimation par approximation
Sugiyama	1601,61	1563,93	1338,83
Golomb 12	5480,05	12836,43	9896,2
Minimum Feed Back Vertex Set	435,66	506,18	484,55
Mario	15,45	17,46	15,89
Java routing	132,1	201,87	130,71
Radiation	3,73	3,05	1,86
Rectangle Packing	1019,85	15368,56	80,06
money change	47,52	51,08	39,12
crew	255,25	245,98	245,98
dépôt placement	6533,96	12460,98	8915,93
dudney thea	63,19	50,99	50,99

TABLE 5 : Evaluation de l'estimation sur un problème d'optimisation

On peut remarquer que globalement notre estimateur est efficace. Bien évidemment il reste soumis aux mêmes conditions que les estimations habituelles faites dans le contexte d'un sondage (i.e intervalle de confiance etc..).

3.8 Avantages et limitations

Nous venons de voir une méthode permettant d'estimer les paramètres d'un arbre de recherche changeant au court du temps en fonction de l'objectif (e.g nombre de noeuds de solutions, le temps de résolution). Nous rappelons que l'idée est d'approximer cet arbre par celui d'un problème de satisfaction s'en approchant. Cependant, cette approximation supprime un avantage conséquent de l'estimation des problèmes de satisfaction : la loi des grands nombres n'est plus applicable. En effet notre approximation dépend fortement de l'objectif courant, le biais est donc plus ou moins important selon notre choix de valeur

d'objectif pour déterminer le problème de satisfaction approchant. Mais cette information n'est pas une moyenne, on peut difficilement l'estimer.

4 Méthode d'élimination des stratégies

4.1 Etat de l'art

Comme nous l'avons dit dans l'introduction, la question de sélection d'algorithme a toujours été très importante, notamment depuis sa formalisation par Rice [3]. La méthode du portfolio est une des premières méthode générique tentant de répondre à cette question. Elle a été introduite dans [17] en faisant référence au problème de finance portant le même nom. Le principe de base du portfolio en finance est d'effectuer une diversification d'actifs permettant de mieux maîtriser le risque et donc la rentabilité globale du portefeuille. Également en informatique cela consiste à effectuer une diversification mais cette fois ci des algorithmes, avec pour but de maîtriser le risque dû au choix de l'algorithme. Dans [17], les auteurs créent un portfolio composé de plusieurs algorithmes randomisé de types Las Vegas [5]. Le principe est d'exécuter en parallèle et de manière indépendant les algorithmes présent dans le portfolio. Les résultats obtenus montrent l'intérêt de cette méthode tant au niveau performance qu'au niveau robustesse (i.e le risque dû à la sélection d'un algorithme est diminué). Dans [18] les auteurs confirment que la méthode du portfolio est efficace et proposent d'exécuter en parallèle ou par intermittence sur un même processeur des algorithmes de type Las Vegas. Ils montrent aussi que leur approche est plus efficace en exécutant plusieurs algorithmes à la fois plutôt qu'un seul tout en les combinant avec des restarts pour résoudre le problème des quasigroup completion [19] en utilisant des CSP. Un restart consiste à recommencer une exploration à partir d'un noeud plus haut dans l'arbre. Un critère d'arrêt est défini explicitement, cela peut être par exemple un nombre maximal d'échecs dans une branche d'un arbre. Ce mécanisme fait ressortir le fait que de petites explorations sont plus efficaces qu'une longue. En effet, lorsqu'une exploration est longue, celle-ci produit beaucoup. L'intérêt des restarts en programmation par contraintes est dû à la randomisation des algorithmes de recherche. Celle-ci se situe principalement dans les tie-break. Lors d'une exploration de l'arbre de recherche, il arrive que deux variables ou deux valeurs puissent être sélectionnable en même temps par une stratégie. Un critère doit être défini pour en choisir une. Ce critère s'appelle le tie-break. La randomisation d'un algorithme de recherche peut être effectuée en ajoutant de l'aléatoire dans le tie-break. Pour en revenir au portfolio, en règle générale, pour sélectionner un ou plusieurs algorithmes, deux approches s'opposent :

— celles offline, le choix du ou des algorithmes pour résoudre le problème est effectué avant que la résolution ne commence.

5. Les algorithmes de type Las Vegas sont des algorithmes probabilites retournant toujours une réponse exacte. Seul le temps de réponse peut varier d'une exécution à l'autre. En général, le but sera d'obtenir une bonne probabilité que le temps de résolution soit efficace.

— celles online, le ou les algorithmes à exécuter sont choisis durant la résolution du problème.

Ces dernières années beaucoup de méthodes offline ont vu le jour comme le souligne Kotthoff dans [20] [6].

Par exemple SatZilla qui est un des plus ancien solveur utilisant la méthode du Portfolio. Celui-ci a même relancé l'intérêt pour cette méthode en gagnant plusieurs fois la compétition SAT [21]. SatZilla est un solveur portfolio c'est à dire qu'il est composé d'une collection de solvers. Un contrôleur est chargé quand à lui de sélectionner l'algorithme le plus approprié pour résoudre efficacement une instance donnée. Pour SatZilla le contrôleur se base sur la prédiction du temps de résolution de chacun des solveurs dont il dispose en se basant sur des modèles de régression [22], construient à partir d'une grosse base de données contenant des instances déjà résolues (comme toutes les approches offline). Plus récemment dans SatZilla a été introduit un modèle de classification plus sensible aux coûts améliorant la prédiction [23]. Malitsky et al. [24] ont proposé une nouvelle manière pour sélectionner un solveur basé sur la sensibilité des coûts par regroupement hiérarchique (Cost-Sensitive Hierarchical Clustering). Leur solveur CSHC a gagné 2 médailles d'or en 2013 dans la compétition SAT. Le solveur CPHydra [25] est aussi un solveur portfolio mais pour résoudre des CSP. La sélection du solveur est faite avec un raisonnement par cas (Case Base Raisonning en anglais). Dans la même idée de solveur portfolio, AQME [26] résout les formules boolèennes quantifiées. Ce solveur est implémenté sur la base de la librairie de Data Mining Weka [7]. Diverses versions d'AQME ont évolués dans les compétitions : une utilisant des arbres de décision, une autre utilisant une régression logistique et enfin une dernière basée sur l'algorithme des k-neigbhour. Le solveur portolio ppfolio [27] quand à lui lance simplement tous les solveurs en parallèle, aucun apprentissage n'est nécessaire. Epstein et al. ont proposé Adaptive Constraint Engine [28](ACE). L'idée de cette méthode est d'unifier les décisions de plusieurs heuristiques afin de guider la procédure de recherche. Pour cela, chaque heuristique vote pour une possible variable et une possible valeur. Ensuite, un contrôleur choisi le couple (variable,valeur) le plus approprié selon des poids pré-calculés (offline) attribués à chaque heuristiques.

Un modèle bayésien utilisant un filtrage collaboratif est présenté par Stern et al. [29]. Roberts et al. [30] quand à eux ont opté pour un arbre de décision basé sur une multitude de caractéristiques. Pour chaque nouveau problème rencontré, ils ordonnancent chaque heuristique par probabilité de succès décroissante. Les heuristiques sont exécutés séquentiellement selon l'ordre établi par l'ordonnancement, jusqu'à ce qu'une solution soit trouvée. De manière analogue, Streeter et al. [31] utilisent des techniques d'optimisation pour produire un ordonnancement de solveur exécuté dans un ordre spécifique pour un temps donné afin de maximiser les chances de résolution du problème. Cependant, toutes les approches présentées ci dessus utilisent une approche offline avec un ensemble d'apprentissage

6. Beaucoup de références concernant la sélection d'algorithme peuvent être trouvées sur `https://larskotthoff.github.io/assurvey/`

7. `http://www.cs.waikato.ac.nz/ml/weka/`

choisi en amont de la résolution. Cette technique suppose que la connaissance (i.e le meilleur algorithme ou le modèle construit) est obtenue grâce à l'ensemble d'apprentissage et que donc cet ensemble peut généraliser toutes les instances de problèmes dans une même classe. Pour faire face à cette limite, des approches online sont étudiées. Dans la même idée, d'effectuer une politique d'ordonnancement Gaglio et al. [32] ont proposé une nouvelle méthode utilisant des algorithmes évolutionnaires. Celle-ci se base sur le fait qu'avec une méthode online, sélectionner le meilleur algorithme directement(i.e sans calcul) est impossible. Il est plutôt préférable d'allouer du temps à tous les candidats (i.e algorithmes) selon certaines règles de distribution. Ensuite selon la performance de chaque algorithme évaluée par sa fonction fitness, un modèle linéaire est construit. Cette étape constitue une itération. Tant que le temps alloué pour trouver l'algorithme n'est pas utilisé, des itérations sont effectuées en distribuant du temps selon les performances évaluées/prédient par les modèles linéaires. Dans [33], les auteurs se basent sur des prédictions de temps de résolution. Cette méthode s'appelle GambleTA. L'idée principale est d'allouer du temps à chacun des algorithmes sous la forme d'un problème de bandit manchot. D'autre approches utilisent des machines learning (ML). Par exemple, Samulowitz et al. [34] appliquent des techniques de ML pour produire une combinaison online(à la volée) d'heuristiques. Arbelaez et al. [35] utilisent une machine à vecteurs de support (SVM) avec un kernel Gaussien afin de sélectionner le meilleur heuristique tout au long de la recherche. Cette sélection est effectuée sur certains noeuds précis de l'arbre défini par l'utilisateur. Le problème des approches online que nous avons vu précédemment, était que soit elles perdent du temps en allouant des ressources à de mauvais algorithme ou soit tentaient de caractériser un problème, ce qui est difficile à faire. Une autre méthode intéressante a été proposé par Lobjois et Lemaître dans [36]. Ils estiment le temps d'exécution en prédisant le nombre de noeuds dans l'arbre de recherche à explorer ainsi que le temps passé par noeud grâce à un adaptation de l'algorithme de Knuth [13]. Cependant cette approche est limitée aux résolutions utilisant un algorithme de type séparation et évaluation (Branch and Bound) et partage les défaults de l'algorithme de Knuth (une grande variances entre les observations). Notre approche tente de tirer parti de ces observations en effectuant un apprentissage online sur le problème le plus représentatif : lui même.

4.2 Méthode

Nous venons de voir certaines limitations des autres méthodes de sélection d'algorithmes. Nous proposons ici une nouvelle méthode : l'estimation de la meilleure ou des meilleures stratégies pour une instance donnée. L'idée de celle-ci n'est pas d'apprendre des critères une fois pour toutes, mais bien d'estimer l'efficacité des stratégies. En utilisant les méthodes traditionnelles, la comparaison des stratégies n'est seulement possible lorsque l'une d'elles a terminé sa résolution. En effet, deux stratégies différentes vont étudier un espace de recherche différent car, cet espace dépend de l'ordre d'évaluation des variables et des valeurs. Lorsqu'un problème est résolu avec EPS, les choses sont différentes :

un grand nombre de sous-problèmes indépendant est généré pour être ensuite résolu. Chaque sous-problème correspond une partie de l'espace de recherche : le problème est résolu lorsque tous les sous-problèmes le sont eux aussi. Avec la théorie de l'échantillonnage (voir section 2.3), nous avons vu qu'il n'était pas nécessaire d'analyser entièrement une population pour connaître approximativement ses paramètres. L'idée est donc d'utiliser l'estimateur introduit précédemment pour ainsi mesurer l'efficacité des stratégies. De plus, en utilisant cet estimateur, nous bénéficions de tous ses avantages. En conséquence, en effectuant un échantillonnage aléatoire, notre estimation sera très précise et donc la stratégie choisie sera efficace. Néanmoins, en résolvant plusieurs sous-problèmes avec les pires stratégies, le temps de résolution est fortement impacté. Pour palier ce problème, l'échantillonnage est effectué sous contrôle, c'est-à-dire qu'un temps maximal est alloué pour chaque sous-problème. Cette allocation n'est pas faite n'importe comment, elle dépend du meilleur temps de résolution du sous-problème considéré. D'après nos expérimentations, arrêter la résolution du sous-problème lorsque celui-ci dépasse 4 fois celui du meilleur est efficace. Dans la même optique de réduire les coûts liés à l'évaluation des pires stratégies, l'échantillonnage est effectué en plusieurs étapes. La première consiste à les comparer sur un échantillon de 30 observations. Cette quantité peut paraître insuffisante, mais est suffisante pour distinguer les tendances globales des stratégies. À partir de cet échantillon, les plus mauvaises stratégies vont être éliminées grâce à des critères qualitatifs. Celles ayant été arrêtées dans $\frac{2}{3}$ des observations (i.e leur temps d'exécution est supérieur à 4 fois celui de la meilleure stratégie dans $\frac{2}{3}$ des cas) sont supprimées. Ensuite, nous continuons notre échantillonnage en augmentant à 100 nos observations. Le même schéma d'élimination que précédemment est effectué. Toujours dans le but d'obtenir des informations fiables, nous augmentons encore la taille de notre échantillon à 500 observations. Grâce à l'augmentation de la précision, nous pouvons maintenant utiliser les outils statistique à notre disposition tel que le test de student pour comparer l'efficacité des stratégies. Une variante du test de Student est utilisée : le test de comparaison sur des échantillons appariés(voir section 2.4). Celui-ci permet de déterminer pour une probabilité donnée si la la moyenne des différences entre deux échantillons (relative à chacune des stratégies) est statistiquement différente. Si la différence est significative, nous pourrons alors supprimer la plus mauvaise stratégie. La condition de normalité pour pouvoir appliquer un test de student est respectée en vertu du TLC. D'autre part, nos échantillons sont appariés puisque les conditions initiales de résolution de chacun des sous-problèmes sont les mêmes (i.e solveur et position dans l'arbre recherche par définition d'un sous-problème), seule la stratégie utilisée change. Il est important de noter qu'avant d'effectuer ce test, nous devons obtenir des informations non biaisées. En effet, lorsqu'une résolution est arrêtée, les données sont modifiées. Afin d'éviter ce biais, pour chaque stratégie restante, les sous-problèmes arrêtés sont relancés là où ils ont été interrompus. Avant d'éliminer une stratégie, il faut déterminer si le temps résultant de l'application en parallèle des stratégies(i.e la somme des temps minimaux de chaque sous-problème avec l'utilisation en parallèle des stratégies) et

donc coût de résolution systématique de plusieurs fois le même sous-problème est intéressant face à une résolution unique mais cette fois-ci en utilisant une stratégie. S'il ressort une meilleure efficacité de l'application de plusieurs stratégies, nous le la supprimons pas et la gardons pour vérifier cela avec un plus gros échantillon. À la suite de ces éliminations, il arrive que plusieurs stratégies soient encore en compétition. Pour éviter le plus possible une erreur qui potentiellement peut coûter (en terme de temps de résolution), nous augmentons une dernière fois la taille de notre échantillon à 1000 observations. Les méthodes d'éliminations précédemment énoncées sont à nouveau appliquées ainsi que la vérification de l'efficacité d'appliquer simultanément plusieurs stratégies. Pour finir, si plusieurs stratégies sont toujours présentes et qu'il n'existe aucun intérêt à les lancer en parallèle, nous choisissons alors la meilleure d'entre elles pour résoudre le problème.

4.3 Extension aux problèmes d'optimisation

Nous venons de voir une méthode permettant d'estimer la meilleure stratégie à appliquer pour un problème de satisfaction en programmation par contraintes. Celle-ci se base sur l'estimateur introduit dans la section 3. Nous avons aussi introduit une variante pour les problèmes d'optimisation. Une première approche pour effectuer un choix entre diverses stratégie serait d'utiliser l'estimateur biaisé. Seulement utilisé cette estimateur coûte plus en temps de calcul que celui des problèmes de satisfaction et n'est pas forcément nécessaire dans notre contexte. En effet dans notre cas, seul les variations entre stratégies nous intéresse. Une estimation fidèle n'est pas utile, seul une classification des stratégie est nécessaire. Nous allons donc garder le même raisonnement que précédemment pour comparer les stratégies. Ce raisonnement reste toujours possible grâce à l'existence de variantes du théorème centrale limite plus faible. Plus particulièrement qui affaiblie la condition d'indépendance des variables aléatoire. Ce qui correspond exactement à notre cas d'application pour pouvoir utiliser les test de Student.

4.4 Expérimentations

Dans cette sous-partie, une validation expérimentalement notre méthode de sélection d'algorithmes. Nos expérimentations ont été effectuées avec plusieurs machines parallèles où nous avons pu valider notre méthode avec 6, 40 et 500 coeurs. Cependant, par souci de simplification, la comparaison se fera sur le travail effectué (i.e la somme des temps de résolutions des sous-problèmes). Nos benchmarks, ont été réalisés avec diverses statégies orthogonales. Les stratégies de choix de variable utilisées sont weighted degree [37], activité [38], max regret , most constrained et first fail. L'idée des stratégies Weighted Degree et activity est de s'attaquer aux zones difficiles du problème. Pour cela durant l'exploration de l'espace de recherche, des calculs sont effectués. Dans Weighted Degree, chaque variable possède un compteur. Lorsqu'une contrainte est violée, les variables présentes dans la contrainte voient leur compteur incrémenté. À chaque

niveau de l'arbre, la variable la plus présente dans les contraintes violée (i.e celle avec la plus grande valeur) est sélectionnée. La méthode de sélection pour la stratégie activity, est un peu différente, au lieu de calculer le nombre de fois ou une contrainte est violée, un calcul mesure l'impact d'une affectation durant l'exploration de l'arbre de recherche. Les variables avec le plus gros impact seront sélectionnées. La stratégie max regret consiste à choisir la prochaine instanciation de façon à minimiser un risque estimé [39]. Pour finir, most constrained sélectionne la variable la plus contrainte (i.e qui est présente dans le plus de contrainte) et first fail choisis la variable avec le plus petit domaine. À cela, on associe des stratégies de choix de valeurs telles que la plus petite ou la plus grande valeur du domaine.

Pour réaliser ces expérimentations, plusieurs types de problèmes et de différentes catégories ont été pris sur CPSLIB[16] et sur `http://www.hakank.org`. Dans le tableau 6, la première colonne indique pour chacun des différents problèmes résolus le travail effectué lorsque seulement la meilleure stratégie est utilisée. Cette colonne correspond au meilleur temps possible. Dans une seconde colonne le temps moyen réalisé par notre méthode, dans une troisième le pire des cas observé pour notre méthode et dans une dernière l'espérance du temps de résolution obtenu par la sélection aléatoire d'une stratégie.

Temps / Problème	meilleure stratégie	Ω(méthode)	O(méthode)	$\mathbb{E}$(choix aléatoire)
All intervall series	1 565 823	1 743 208	1 749 246	4 035 169
Golomb ruler	11 024 831	12 977 935	13 208 311	21 549 156
Lam's	1 624 550	1 761 823	1 766 814	20 444 811
Pair divide the sum	142 179	155 030	160 480	464 237
Permutation	1 098 757	1 335 005	1 363 184	1 551 749
Quasi group	195 360	217 317	218 183	27 633 595
Sport scheduling	2 196 651	2 575 996	2 592 349	6 716 762
Market split	29 570 692	32 053 011	32 218 456	38 162 843
Tank attack puzzle	1 778 463	2 344 677	3 149 234	897 060 324

TABLE 6 : Évaluation de notre méthode de sélection de stratégies : pour les problèmes de satisfaction.

À travers ce tableau, l'efficacité de la méthode est clairement visible, puisque nos pertes face à la meilleure stratégie sont minimes. Par exemple, pour le problème du Quasi group, nous avons pu observer une perte de 20% face à la meilleure stratégie dans le pire des cas. D'autre part, le risque dû à l'évaluation des mauvaises stratégies est efficacement contrôlé grâce à notre mécanisme d'échantillonnage graduel. À travers nos expérimentions, il ressort que dans 95% des cas, la stratégie optimale est sélectionnée. Dans le cas contraire, cette dernière n'est pas mauvaise et est même compétitive, les pertes ne sont pas importantes.

Pour terminer avec la présentation de la méthode, nous pouvons remarquer qu'un de ses très gros avantages vient du fait que l'ensemble d'apprentissage(ici

notre échantillon) n'a pas de problème de représentativité du problème comparé aux méthodes offline, puisque c'est le problème lui-même. Le deuxième point qui n'est pas des moindres est que nous pouvons adapter le nombre de sous-problèmes évalués et générés afin de ne pas comparer les stratégies sur une trop grosse proportion du problème initial. De plus cette méthode se base sur EPS qui est une méthode parallèle. De ce fait, notre méthode est parfaitement parallélisable et possède toutes les bonnes propriétés d'EPS, avec par exemple un bon passage à l'échelle. Cependant, notre méthode possède un inconvénient lié à son estimateur. En effet, celui-ci est incapable d'effectuer une estimation sur les arbres explorés avec un algorithme de type séparation et évaluation.

5 Recherche d'une première solution

5.1 Amélioration de la recherche d'une première solution en décomposant plus le problème

Pour résoudre un problème EPS le découpe en une multitude de sous problèmes. Pour effectuer la résolution, tous les sous problèmes vont être résolus les uns à la suite des autres. L'idée est de se demander si l'augmentation du nombre de sous problèmes générés améliore la recherche d'une première solution. Avant de rentrer dans un cadre mathématique, nous allons expliquer l'intuition de cet idée. Prenons une décomposition quelconque d'un problème (e.g Figure **??**). Dans les sous problèmes générés, certains ont des solutions et d'autres non. Cependant, lorsqu'un sous problème possède deux solution, cela veut dire que la décomposition n'est pas optimale pour rechercher une première solution. L'idée est donc de continuer la décomposition pour espérer séparer les solutions (e.g Figure 10) et donc se donner plus de chance de trouver rapidement une première solution. Une solution est dite découvrable si elle est unique dans le sous problèmes. Si deux solutions sont dans le même sous-problème alors l'une d'elle n'est pas découvrable car on s'arrête à la première solution.

5.2 Preuves :

Évaluer l'efficacité de la recherche d'une première solution passe par la caractérisation de la probabilité du premier succès (i.e le premier sous problème pris contenant une solution). Pour simplifier les choses nous allons considérer un tirage avec remise (donc nos variables sont indépendantes et identiquement distribué puisque c'est une répétition d'épreuves de Bernouilli). La recherche du premier succès correspond à une loi géométrique (voir annexe C). Celle-ci permet de déterminer le rang du premier succès et donc le nombre de sous problèmes évalués.

Soit (a) et (b) deux décompositions similaire dans leur manière de découper. Seul le nombre de sous problèmes engendré diffère.

— (a) On a n sous problèmes de durée k.
— (b) on a $\frac{n}{i}$ sous problèmes de durée i×k $\forall i \in \mathbb{N}^*$.

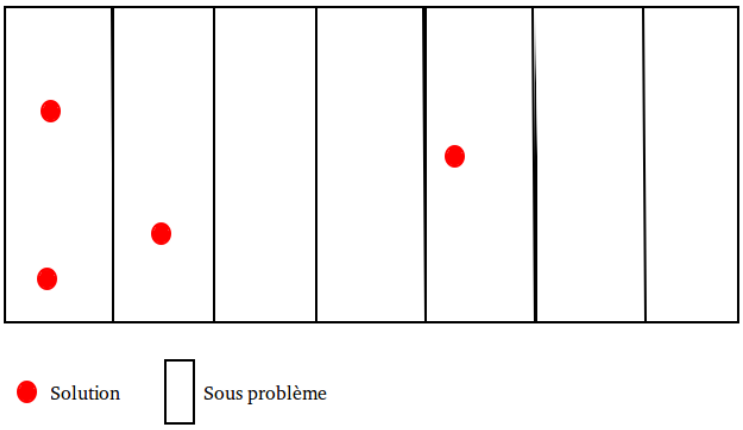


FIGURE 9 : Une première décomposition

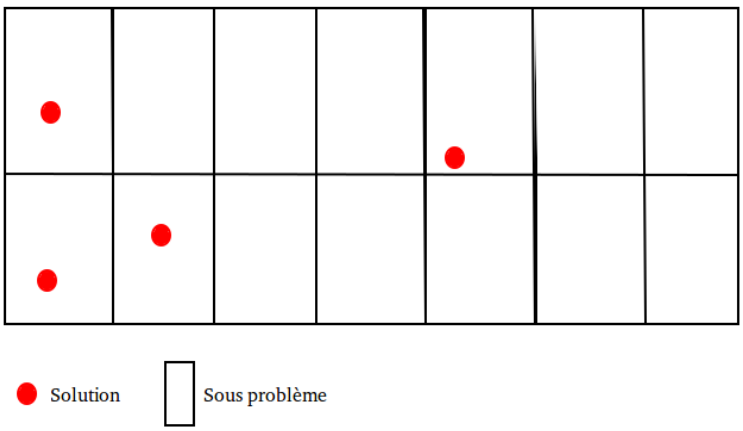


FIGURE 10 : Idée intuitive de l'effet de l'amélioration de la recherche d'une première solution en décomposant plus le problème initial.

Les hypothèses sont les suivantes :
— le temps de résolution est pris sur la résolution complète du sous problème (pour simplifier les choses).

5.2.1 Théorème :

lorsque la décomposition ne met pas au jour de nouvelles solutions découvrable alors le temps moyen de résolution est le même.

Preuve :

Que ce soit dans (a) ou dans (b) il y a p solutions "découvrable" et P solutions en tout, avec $p \leqslant P$.

Pour (a) la probabilité de succès est $\frac{p}{n}$, quand à (b) est de $\frac{i \times p}{n}$.
L'espérance d'une loi géométrique est $\frac{1}{prob(succes)}$. Donc pour (a) et (b) il faut résoudre respectivement en moyenne : $\frac{n}{p}$ et $\frac{n}{i \times p}$ problèmes.

Alors le temps moyen est résolution est $\frac{n}{p} \times k$ pour (a) et $\frac{n}{i \times p} \times i \times k$ on a donc T(a)=T(b).

5.2.2 Théorème :

Lorsque décomposer amène à avoir plus de solution découvrable, alors le temps moyen pour trouver une première solution est diminué.

Preuve :

Il y a respectivement dans (a) et (b) p_1 et p_2 solutions avec $P \geqslant p_1 \geqslant p_2$ car augmenter le nombre de sous problème ne peut que seulement augmenter le nombre de sous problèmes contenant une solution découvrable.

Pour (a) la probabilité de succès est $\frac{p_1}{n}$, quand à (b) celle-ci est de $\frac{i \times p_2}{n}$.
L'espérance d'une loi géométrique est $\frac{1}{prob(succes)}$. Donc pour (a) et (b) il faut résoudre respectivement en moyenne : $\frac{n}{p_1}$ et $\frac{n}{i \times p_2}$ problèmes.

Alors le temps moyen est résolution est $\frac{n}{p_1} \times k$ pour (a) et $\frac{n}{i \times p_2} \times i \times k$. Or comme $p_1 \geqslant p_2$ on a $\frac{n}{p_1} \times k \leqslant \frac{n}{p_2} \times k$. Donc le temps moyen pour trouver une solution sera toujours au moins inférieur.

6 Conclusion et perspectives

Dans ce rapport, nous avons pu voir deux contributions majeures : une méthode d'estimation des paramètres d'un arbre de recherche ainsi que de prévision du temps de résolution d'un problème en programmation par contraintes. Mais aussi nous avons vu une méthode de sélection automatique de stratégie selon sa performance. Nos méthodes sont beaucoup plus avantageuses que celles existantes, notamment concernant notre méthode de sélection de stratégies qui s'effectue online sans pour autant perdre beaucoup de temps face à la meilleure stratégie. Concernant notre estimateur, sa force n'est plus à prouver grâce à toute

la théorie basée sur l'échantillonnage, plus particulièrement la loi des grands nombres, qui nous permet d'éliminer de trop lourd calcul et de ne plus être dépendant de la taille de l'arbre pour estimer correctement sa taille. Ce travail reste encore à approfondir, car il n'est pas fait pour des algorithmes de type séparation évaluation. Néanmoins de nombreuses perspectives sont envisageables, cela peut aller de l'étude des caractéristiques des sous-problèmes grâce à un échantillonnage, d'ajout de statistique pour améliorer la vitesse de résolution d'un problème, la création d'une stratégie dynamique en fonction des caractéristiques observés,l'amélioration de la recherche d'une première solution.

A Table de Student

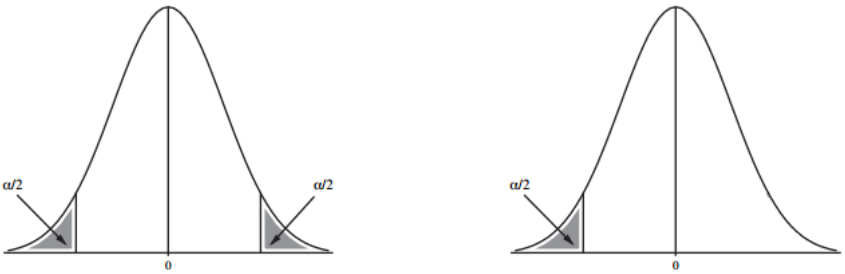


FIGURE 11 : Choix de la lecture de la table pour un test bilatérale ou unilatérale.

α	80.0%	66.6%	50.0%	40.0%	25.0%	20.0%	10.0%	5%	2%	1%	0.2%
$\frac{\alpha}{2}$	40.0%	33.3%	25.0%	20.0%	12.5%	10.0%	5.0%	2.5%	1.0%	0.5%	0.1%
d.d.l											
1	0.325	0.577	1.000	1.376	2.414	3.078	6.314	12.706	31.821	63.657	318.31
2	0.289	0.500	0.816	1.061	1.604	1.886	2.920	4.303	6.965	9.925	22.327
3	0.277	0.476	0.765	0.978	1.423	1.638	2.353	3.182	4.541	5.841	10.215
4	0.271	0.464	0.741	0.941	1.344	1.533	2.132	2.776	3.747	4.604	7.173
5	0.267	0.457	0.727	0.920	1.301	1.476	2.015	2.571	3.365	4.032	5.893
6	0.265	0.453	0.718	0.906	1.273	1.440	1.943	2.447	3.143	3.707	5.208
7	0.263	0.449	0.711	0.896	1.254	1.415	1.895	2.365	2.998	3.499	4.785
8	0.262	0.447	0.706	0.889	1.240	1.397	1.860	2.306	2.896	3.355	4.501
9	0.261	0.445	0.703	0.883	1.230	1.383	1.833	2.262	2.821	3.250	4.297
10	0.260	0.444	0.700	0.879	1.221	1.372	1.812	2.228	2.764	3.169	4.144
11	0.260	0.443	0.697	0.876	1.214	1.363	1.796	2.201	2.718	3.106	4.025
12	0.259	0.442	0.695	0.873	1.209	1.356	1.782	2.179	2.681	3.055	3.930
13	0.259	0.441	0.694	0.870	1.204	1.350	1.771	2.160	2.650	3.012	3.852
14	0.258	0.440	0.692	0.868	1.200	1.345	1.761	2.145	2.624	2.977	3.787
15	0.258	0.439	0.691	0.866	1.197	1.341	1.753	2.131	2.602	2.947	3.733
20	0.257	0.437	0.687	0.860	1.185	1.325	1.725	2.086	2.528	2.845	3.552
25	0.256	0.436	0.684	0.856	1.178	1.316	1.708	2.060	2.485	2.787	3.450
29	0.256	0.435	0.683	0.854	1.174	1.311	1.699	2.045	2.462	2.756	3.396
30	0.256	0.435	0.683	0.854	1.173	1.310	1.697	2.042	2.457	2.750	3.385
60	0.254	0.433	0.679	0.848	1.162	1.296	1.671	2.000	2.390	2.660	3.232
∞	0.253	0.431	0.674	0.842	1.150	1.282	1.645	1.960	2.326	2.576	3.090

TABLE 7 : Table de student, on lira le risque α pour le cas bilatéral et $\frac{\alpha}{2}$ pour le cas unilatéral

B Relation entre taille d'un échantillon, indice de confiance et marge d'erreur

Pour déterminer la taille de léchantillon on utilise cette formule

$$n = \frac{z^2 p(1-p)}{d^2}$$

ce qui correspond dans le pire des cas (p=0.5) à :

$$n = \frac{z^2}{4 \times d^2}$$

avec n la taille de léchantillon, z le niveau de confiance selon la loi normale centrée réduite (pour un niveau de confiance de 95%, z = 1.96, pour un niveau de confiance de 99%, z = 2.575) et d = marge derreur tolérée. Par exemple, pour calculer une proportion avec un niveau de confiance de 95% et une marge derreur à 5% nous obtenons donc :

$$n = \frac{1.96^2}{4 \times 0.05^2} = 384.16$$

C Loi géométrique

Source Wikipédia

La loi géométrique est une loi de probabilité apparaissant dans de nombreuses applications. La loi géométrique de paramètre p ($0 < p < 1$) correspond au modèle suivant : On considère une épreuve de Bernoulli dont la probabilité de succès est p et celle d'échec q = 1 - p. On renouvelle cette épreuve de manière indépendante jusqu'au premier succès. On appelle X la variable aléatoire donnant le rang du premier succès. Les valeurs de X sont les entiers naturels non nuls 1, 2, 3, ... La probabilité que X = k est alors, pour$k \in \mathbb{N}^*$ $p(k) = q^{k-1}p$. On dit que X suit une loi géométrique de paramètre p.

Références

1. Cook, S.A. : The complexity of theorem-proving procedures. In : Proceedings of the third annual ACM symposium on Theory of computing, ACM (1971) 151–158
2. Klotz, W. : Graph coloring algorithms. Mathematics Report **5**(2002) (2002) 1–9
3. Rice, J.R. : The algorithm selection problem. (1975)
4. Michel, L., See, A., Van Hentenryck, P. : Transparent parallelization of constraint programming. INFORMS Journal on Computing **21**(3) (2009) 363–382

5. Perron, L. : Search procedures and parallelism in constraint programming. In : Principles and Practice of Constraint Programming–CP99, Springer (1999) 346–360
6. Jaffar, J., Santosa, A.E., Yap, R.H., Zhu, K.Q. : Scalable distributed depth-first search with greedy work stealing. In : Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, IEEE (2004) 98–103
7. Chu, G., Schulte, C., Stuckey, P.J. : Confidence-based work stealing in parallel constraint programming. In : Principles and Practice of Constraint Programming-CP 2009. Springer (2009) 226–241
8. Zoeteweij, P., Arbab, F. : A component-based parallel constraint solver. In : Coordination Models and Languages, Springer (2004) 307–322
9. Schulte, C. : Parallel search made simple. In : Proceedings of TRICS : Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP. (2000) 41–57
10. Régin, J.C., Rezgui, M., Malapert, A. : Embarrassingly parallel search. In : Principles and Practice of Constraint Programming, Springer (2013) 596–610
11. Régin, J.C., Rezgui, M., Malapert, A. : Improvement of the embarrassingly parallel search for data centers. In : Principles and Practice of Constraint Programming, Springer (2014) 622–635
12. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K. : Algorithm runtime prediction : Methods & evaluation. Artificial Intelligence **206** (2014) 79–111
13. Knuth, D.E. : Estimating the efficiency of backtrack programs. Mathematics of computation **29**(129) (1975) 122–136
14. Chen, P.C. : Heuristic sampling : A method for predicting the performance of tree searching programs. SIAM Journal on Computing **21**(2) (1992) 295–315
15. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T. : Estimating search tree size. In : Proc. of the 21st National Conf. of Artificial Intelligence, AAAI, Menlo Park. (2006)
16. : CSPLib : A problem library for constraints. `http://www.csplib.org` (1999)
17. Huberman, B.A., Lukose, R.M., Hogg, T. : An economics approach to hard computational problems. Science **275**(5296) (1997) 51–54
18. Gomes, C.P., Selman, B. : Algorithm portfolios. Artificial Intelligence **126**(1) (2001) 43–62
19. Walsh, T. : CSPLib problem 003 : Quasigroup existence. `http://www.csplib.org/Problems/prob003`
20. Kotthoff, L. : Algorithm selection for combinatorial search problems : A survey. arXiv preprint arXiv :1210.7959 (2012)
21. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K. : Detailed satzilla results from the data analysis track of the 2011 sat competition. In : 14th International Conference on Theory and Applications of Satisfiability Testing. (2011)
22. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K. : Satzilla-07 : the design and analysis of an algorithm portfolio for sat. In : Principles and Practice of Constraint Programming–CP 2007. Springer (2007) 712–727
23. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K. : Satzilla2009 : an automatic algorithm portfolio for sat. SAT **4** (2009) 53–55

24. Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M. : Algorithm portfolios based on cost-sensitive hierarchical clustering. In : Proceedings of the Twenty-Third international joint conference on Artificial Intelligence, AAAI Press (2013) 608–614
25. OMahony, E., Hebrard, E., Holland, A., Nugent, C., OSullivan, B. : Using case-based reasoning in an algorithm portfolio for constraint solving. In : Irish Conference on Artificial Intelligence and Cognitive Science. (2008) 210–216
26. Pulina, L., Tacchella, A. : A multi-engine solver for quantified boolean formulas. In : Principles and Practice of Constraint Programming–CP 2007. Springer (2007) 574–589
27. Roussel, O. : Description of ppfolio 2012. Proc. SAT Challenge (2012) 46
28. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B. : The adaptive constraint engine. In : Principles and Practice of Constraint Programming-CP 2002, Springer (2002) 525–540
29. Stern, D.H., Samulowitz, H., Herbrich, R., Graepel, T., Pulina, L., Tacchella, A. : Collaborative expert portfolio management. In : AAAI. (2010) 179–184
30. Roberts, M., Howe, A. : Directing a portfolio with learning. In : AAAI 2006 Workshop on Learning for Search. (2006) 129–135
31. Streeter, M., Golovin, D., Smith, S.F. : Combining multiple heuristics online. In : Proceedings of the national conference on artificial intelligence. Volume 22., Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999 (2007) 1197
32. Gagliolo, M., Zhumatiy, V., Schmidhuber, J. : Adaptive online time allocation to search algorithms. In : Machine Learning : ECML 2004. Springer (2004) 134–143
33. Gagliolo, M., Schmidhuber, J. : Learning dynamic algorithm portfolios. Annals of Mathematics and Artificial Intelligence **47**(3-4) (2006) 295–328
34. Samulowitz, H., Memisevic, R. : Learning to solve qbf. In : AAAI. Volume 7. (2007) 255–260
35. Arbelaez, A., Hamadi, Y., Sebag, M. : Online heuristic selection in constraint programming. (2009)
36. Lobjois, L., Lemaître, M., et al. : Branch and bound algorithm selection by performance prediction. In : AAAI/IAAI. (1998) 353–358
37. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L. : Boosting systematic search by weighting constraints. In : ECAI. Volume 16. (2004) 146
38. Michel, L., Van Hentenryck, P. : Activity-based search for black-box constraint programming solvers. In : Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems. Springer (2012) 228–243
39. Caseau, Y., Laburthe, F. : Solving small tsps with constraints. In : ICLP. Volume 97., Citeseer (1997) 104