Un bandit manchot pour combiner CHB et VSIDS

Mohamed Sami Cherif, Djamal Habet, Cyril Terrioux

▶ To cite this version:

Mohamed Sami Cherif, Djamal Habet, Cyril Terrioux. Un bandit manchot pour combiner CHB et VSIDS. Actes des 16èmes Journées Francophones de Programmation par Contraintes (JFPC), Jun 2021, Nice, France. hal-03270931

HAL Id: hal-03270931
https://amu.hal.science/hal-03270931

Submitted on 25 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Actes JFPC 2021

Un bandit manchot pour combiner CHB et VSIDS*

Mohamed Sami Cherif† **Djamal Habet** **Cyril Terrioux**

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

Résumé

Les solveurs *Conflict Driven Clause Learning* (CDCL) sont efficaces pour résoudre des instances structurées avec un grand nombre de variables et de clauses. Un composant important de ces solveurs est l'heuristique de branchement qui sélectionne la prochaine variable de décision. Dans cet article, on propose d'utiliser un bandit manchot pour combiner deux heuristiques de l'état de l'art pour SAT, à savoir *Variable State Independent Decaying Sum* (VSIDS) et *Conflict History-Based* (CHB). Le bandit évalue et choisit de manière adaptative une heuristique adéquate à chaque redémarrage. Une évaluation expérimentale est menée et montre que la combinaison de VSIDS et CHB avec un bandit est compétitive et surpasse les deux heuristiques.

Abstract

Conflict Driven Clause Learning (CDCL) solvers are known to be efficient on structured instances and manage to solve ones with a large number of variables and clauses. An important component in such solvers is the branching heuristic which picks the next variable to branch on. In this paper, we propose a Multi-Armed Bandit (MAB) framework to combine two state-of-the-art heuristics for SAT, namely the Variable State Independent Decaying Sum (VSIDS) and the Conflict History-Based (CHB) branching heuristic. The MAB takes advantage of the restart mechanism to evaluate and adaptively choose an adequate heuristic. We conduct an experimental evaluation which shows that the combination of VSIDS and CHB using MAB is competitive and even outperforms both heuristics.

1 Introduction

Résoudre le problème de Satisfiabilité (SAT) consiste à déterminer, étant donné une formule booléenne en Forme Normale Conjonctive (FNC), s'il existe une affectation des variables qui la satisfait. SAT est au cœur de nombreuses applications dans différents domaines et est utilisé pour modéliser une grande variété de problèmes académiques et industriels [25, 11, 18]. C'est aussi le premier problème de décision prouvé NP-complet [10]. Néanmoins, les solveurs modernes, appelés Conflict Driven Clause Learning (CDCL) [26], parviennent à résoudre des instances impliquant un grand nombre de variables et de clauses. Un composant important de ces solveurs est l'heuristique de branchement qui sélectionne la prochaine variable sur laquelle brancher. Variable State Independent Decaying Sum (VSIDS) [27] est l'heuristique dominante depuis son introduction il y a deux décennies. Récemment, Liang et al. ont conçu une nouvelle heuristique pour SAT, appelée Conflict History-Based (CHB) [22], et ont montré qu'elle est compétitive avec VSIDS. Ces dernières années, la majorité des solveurs CDCL présentés dans les compétitions SAT incorporent une variante d'une de ces deux heuristiques.

Des travaux récents ont montré l'intérêt de l'apprentissage automatique dans la conception d'heuristiques de recherche efficaces pour SAT [22, 23, 19] ainsi que pour d'autres problèmes de décision [33, 32, 28, 8]. L'un des principaux défis est de définir une heuristique avec des performances élevées sur n'importe quelle instance considérée. En effet, une heuristique peut très bien fonctionner sur une famille d'instances tout en échouant drastiquement sur une autre. Pour cela, on utilise l'apprentissage par renforcement sous la forme de bandit manchot afin de choisir une heuristique adéquate parmi CHB et VSIDS pour chaque instance. Le bandit évalue et choisit de manière adaptative une heuristique adéquate à chaque redémarrage. L'évaluation est réalisée grâce à une fonction de récompense, qui doit estimer l'efficacité d'une heuristique en se basant sur les informations acquises lors des exécutions entre les redémarrages. On a choisi une fonction de récompense

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet DEMOGRAPH (ANR-16-CE40-0028).

†Papier doctorant : Mohamed Sami Cherif est auteur principal.

qui estime la capacité d'une heuristique à atteindre des conflits rapidement et efficacement. Le bandit utilise la stratégie Upper Confidence Bound (UCB) [4] pour sélectionner le bras adéquat à chaque redémarrage. L'évaluation expérimentale menée montre que le bandit réalise un gain substantiel, principalement en termes d'instances satisfiables, par rapport à VSIDS et CHB.

Cet article est organisé de la manière suivante. Un aperçu des algorithmes CDCL est donné dans la section 2. Les heuristiques VSIDS et CHB ainsi que le problème du bandit manchot sont décrits dans la section 3. L'idée proposée est détaillée dans la section 4 et évaluée expérimentalement dans la section 5. Enfin, on conclut et on discute les perspectives de ce travail dans la section 6.

2 Préliminaires

Soit X un ensemble des variables propositionnelles. Un littéral est une variable $x \in X$ ou sa négation $\overline{x}$. Une clause est une disjonction de littéraux. Une formule en Forme Normale Conjonctive (FNC) est une conjonction de clauses. Une affectation $I : X \to \{vrai, faux\}$ associe à chaque variable une valeur booléenne et peut être représentée comme un ensemble de littéraux. un littéral est satisfait par une affectation I si $l \in I$, sinon il est falsifié par I. Une clause est satisfaite par une affectation I si au moins un de ses littéraux est satisfait par I, sinon elle est falsifiée par I. Une formule FNC est satisfiable s'il existe une affectation qui satisfait toutes ses clauses, sinon elle est insatisfiable. La résolution du problème de Satisfiabilité (SAT) consiste à déterminer si une formule FNC donnée est satisfiable.

Bien que SAT soit NP-complet [10], les solveurs CDCL (Conflict Driven Clause Learning) sont efficaces et parviennent à résoudre des instances impliquant un grand nombre de variables et de clauses. Ces solveurs reposent sur des heuristiques de branchement puissantes ainsi que sur plusieurs techniques de résolution, à savoir la propagation unitaire, l'apprentissage de clauses et les redémarrages, entre autres. À chaque étape, la propagation unitaire est appliquée pour simplifier la formule en propageant les littéraux dans les clauses unitaires. Ensuite, une heuristique de branchement sélectionne une variable en fonction des informations acquises tout au long de la recherche. Plus important encore, lorsqu'un conflit est détecté, c'est-à-dire qu'une clause est falsifiée par l'affectation en cours, les étapes de l'algorithme sont retracées et les clauses impliquées dans le conflit sont résolues jusqu'au premier point d'implication unitaire (FUIP) dans le graphe d'implication [26]. La clause produite par ce processus est apprise, c'est-à-dire ajoutée à la formule. Cela permet d'éviter de revisiter un sous-espace de l'arborescence de recherche déjà exploré. Initialement introduits pour traiter les phénomènes à queue lourde (heavy-tailed) dans SAT [13], les redémarrages sont également un composant important dans les solveurs CDCL. Au début de chaque redémarrage, les paramètres du solveur et ses structures de données sont réinitialisés afin de lancer l'exploration ailleurs dans l'espace de recherche. Il existe deux stratégies principales de redémarrage, à savoir les redémarrages géométriques [31] et Luby [24]. La plupart des solveurs CDCL modernes utilisent les redémarrages Luby car ils sont plus performants [14].

3 Travaux connexes

3.1 Heuristiques de branchement pour SAT

L'heuristique de branchement est l'un des composants les plus importants des solveurs CDCL modernes et a un impact direct sur leur efficacité. Une heuristique de branchement peut être considérée comme une fonction qui classe les variables à l'aide d'un score mise à jour tout au long de la recherche. Dans cette section, on décrit deux des principales heuristiques de branchement de l'état de l'art.

3.1.1 VSIDS

Variable State Independent Decaying Sum (VSIDS) [27] est l'heuristique la plus utilisée depuis son introduction il y a environ deux décennies. Cette heuristique maintient un score pour chaque variable, appelé activité et initialement fixé à 0. Lorsqu'un conflit survient, l'activité de certaines variables est augmentée de 1 (bump). De plus, les activités des variables sont réduites (decay) périodiquement, généralement après chaque conflit. Plus précisément, les activités des variables sont multipliées par un facteur appartenant à $]0, 1[$. Il existe plusieurs variantes de VSIDS. Par exemple, MiniSat [12] augmente les activités des variables apparaissant dans la clause apprise tandis que Chaff [27] le fait pour toutes les variables impliquées dans le conflit, c'est-à-dire les variables résolues y compris celles de la clause apprise.

3.1.2 CHB

L'heuristique de branchement CHB (Conflict History-Based) a été récemment introduite dans [22]. Cette heuristique, basée sur Exponential Recency Weighted Average (ERWA) [29], favorise les variables impliquées dans les conflits récents comme dans VSIDS. CHB maintient un score (ou une activité) $Q(x)$ pour chaque variable x, initialement fixé à 0. Le score $Q(x)$ est mis

à jour lorsqu'une variable x est branchée, propagée ou affirmée à l'aide de la formule d'ERWA comme suit :

$$Q(x) = (1 - \alpha) \times Q(x) + \alpha \times r(x)$$

Le paramètre $0 < \alpha < 1$ est la taille du pas, initialement fixée à $0,4$ et diminuée de 10^{-6} après chaque conflit jusqu'à un minimum de $0,06$. $r(x)$ est la valeur de récompense de la variable x qui peut diminuer ou augmenter la probabilité de brancher sur x. Des récompenses plus élevées sont accordées aux variables impliquées dans des conflits récents selon la formule suivante :

$$r(x) = \frac{multiplier}{Conflicts - lastConflict(x) + 1}$$

$Conflicts$ indique le nombre de conflits survenus depuis le début de la recherche. $lastConflict(x)$ est mis à jour à la valeur actuelle de $Conflicts$ chaque fois que x est présent dans les clauses analysées durant un conflit. $multiplier$ est fixé à 1,0 lorsque le branchement, la propagation ou l'affirmation de la variable qui a déclenché la mise à jour du score conduit à un conflit, sinon il est réglé à 0,9. L'idée est de donner des récompenses supplémentaires pour les variables qui produisent un conflit.

3.2 Bandit Manchot

Le bandit manchot est un problème d'apprentissage par renforcement constitué d'un agent et d'un ensemble de bras candidats parmi lesquels l'agent doit choisir tout en maximisant le gain attendu. L'agent s'appuie sur des informations sous forme de récompenses données à chaque bras et collectées à travers une séquence d'essais. Un bandit est toujours confronté à un dilemme important qui est le compromis entre l'exploitation et l'exploration. En effet, l'agent doit explorer les bras sous-utilisés assez souvent pour avoir un retour d'information solide tout en exploitant les bons candidats qui ont les meilleures récompenses. Le premier modèle du bandit manchot, appelé bandit manchot stochastique, a été introduit dans [20]. Ensuite, différentes politiques ont été conçues pour le bandit manchot telles que la stratégie ϵ-Greedy [29], qui effectue une exploration aléatoire, Thompson Sampling [30] et la famille Upper Confidence Bound (UCB) [1, 4], qui permettent une exploration plus intelligente.

Ces dernières années, il y a eu un afflux d'intérêt pour l'application des techniques d'apprentissage par renforcement et, en particulier, celles basées sur le bandit manchot dans le contexte de SAT. Par exemple, CHB [22] et LRB [23] (une extension de CHB) sont des heuristiques basées sur ERWA [29] qui est utilisée dans le cadre des problèmes bandit manchot non stationnaires pour estimer les récompenses moyennes de chaque bras. De plus, une nouvelle approche, appelée Bandit Ensemble for parallel SAT Solving (BESS), a été conçue dans [21] pour contrôler la topologie de coopération dans des solveurs SAT parallèles. Des approches similaires sont également utilisées dans le contexte du problèmes de satisfaction de contraintes (CSP) pour choisir une heuristique de branchement parmi un ensemble de candidats à chaque nœud de l'arbre de recherche [33] ou à chaque redémarrage [32]. Enfin, des stratégies de perturbation simples pilotées par des bandits ont également été introduites et évaluées dans [28] pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages.

4 Bandit Manchot pour SAT

Soit $A = \{a_1, \ldots, a_K\}$ l'ensemble des bras du bandit contenant différentes heuristiques candidates. Le bandit sélectionne une heuristique a_i où $i \in \{1 \ldots K\}$ à chaque redémarrage de l'algorithme conformément à la politique Upper Confidence Bound (UCB) [4]. Pour choisir un bras, UCB s'appuie sur une fonction de récompense calculée à chaque exécution pour estimer les performances des heuristiques de branchement. La fonction de récompense joue un rôle important dans le cadre proposé et a un impact direct sur son efficacité. Pour cela, on a choisi une fonction de récompense qui estime la capacité d'une heuristique à atteindre des conflits rapidement et efficacement. Si t désigne l'exécution en cours, la récompense du bras $a \in A$ est calculée comme suit :

$$r_t(a) = \frac{log_2(decisions_t)}{decidedVars_t}$$

$decisions_t$ et $decidedVars_t$ désignent respectivement le nombre de décisions et le nombre de variables décidées, c'est-à-dire les variables sur lesquelles l'algorithme a branché au moins une fois au cours de l'exécution t. Par conséquent, plus les conflits sont rencontrés tôt dans l'arbre de recherche et moins il y a de variables instanciées, plus la valeur de récompense attribuée sera élevée pour l'heuristique correspondante. $r_t(a)$ est clairement dans $[0, 1]$ puisque $decisions_t \leq 2^{decidedVars_t}$. Cette fonction de récompense est adaptée de la mesure de sous-arbre explorée (explored sub-tree) introduite dans [28].

L'algorithme UCB1 [4] est utilisé pour sélectionner l'heuristique de branchement suivante dans l'ensemble des heuristiques candidates A. Les paramètres suivants sont conservés pour chaque bras candidat $a \in A$:

- $n_t(a)$: le nombre de fois où le bras a est sélectionné pendant les t premières exécutions
- $\widehat{r}_t(a)$: la moyenne empirique des récompenses du bras a sur les t premières exécutions

UCB1 sélectionne donc le bras $a \in A$ qui maximise $UCB(a)$ défini ci-dessous. Le terme de gauche de $UCB(a)$ vise à mettre l'accent sur les bras qui ont reçu les récompenses les plus élevées. À l'inverse, le terme de droite assure l'exploration des bras sous-utilisés. Le paramètre c peut aider à équilibrer de manière appropriée les phases d'exploitation et d'exploration.

$$UCB(a) = \widehat{r}_t(a) + c.\sqrt{\frac{ln(t)}{n_t(a)}}$$

Enfin, une stratégie pour le bandit manchot est évaluée par son regret cumulé espéré, c'est-à-dire la différence entre la valeur cumulée espérée de la récompense si le meilleur bras est utilisé à chaque redémarrage et sa valeur cumulée réelle pour toutes les exécutions. Le regret cumulé espéré R_T, où T est le nombre total d'exécutions et $a_t \in A$ désigne le bras choisi à l'exécution t, est formellement défini ci-dessous. En particulier, UCB1 garantit un regret cumulé en $O(\sqrt{K.T.\ln T})$.

$$R_T = \max_{a \in A} \sum_{t=1}^{T} \mathbf{E}[r_t(a)] - \sum_{t=1}^{T} \mathbf{E}[r_t(a_t)]$$

5 Évaluation Expérimentale

5.1 Protocole Expérimental

Dans cette section, on décrit l'évaluation expérimentale de la performance du bandit manchot pour combiner VSIDS et CHB. On considère les instances du Main Track des trois dernières compétitions SAT, totalisant 1 200 instances. Pour les tests, on utilise le solveur Kissat [7] qui a remporté la première place du Main Track de la Competition SAT 2020. Il faut noter que ce solveur est une réimplémentation condensée et améliorée du solveur référence et compétitif CaDiCaL [6, 7] en langage C. Des données fournies par A. Bierre et M. Heule [1] montrent que Kissat est très compétitif et surpasse les vainqueurs des compétitions antérieures sur les benchmarks de 2020 et 2019.

Pour le bandit manchot, on fixe $K = 2$ et on considère les heuristiques de l'état de l'art introduites dans la section 3, à savoir VSIDS et CHB. Ce choix sera discuté à la fin de la section 5.4. On maintient la variante VSIDS déjà implémentée dans Kissat qui est similaire à Chaff où toutes les variables analysées sont bumpées après chaque conflit [27]. On augmente également le solveur avec l'heuristique CHB comme spécifié dans [22] sauf la mise à jour des scores qu'on réalise seulement pour les variables au dernier niveau de décision après la propagation unitaire. De plus, on a implémenté le bandit manchot, qu'on dénotera MAB (pour Multi-Armed Bandit), comme spécifié dans la section 4 avec c fixé hors ligne à 2 après une recherche linéaire de la meilleure valeur. Les récompenses dans UCB sont initialisées en lançant chaque heuristique une fois lors des premiers redémarrages. Il est important de noter que les seuls composants modifiés du solveur sont le composant de décision et le composant de redémarrage, c'est-à-dire que tous les autres composants ainsi que les paramètres par défaut du solveur restent inchangés. Les expériences sont réalisées sur des serveurs Dell PowerEdge M620 avec des processeurs Intel Xeon Silver E5-2609 sous Ubuntu 18.04. Enfin, nous utilisons une limite de temps de 5 000 s pour chaque instance.

5.2 MAB vs (VSIDS,CHB)

Dans le tableau 1, on présente les résultats en termes du nombre d'instances résolues pour CHB, VSIDS et MAB. On inclut également les résultats du Virtual Best Solver (VBS) entre VSIDS et CHB. Avant de discuter des résultats, on rappelle que « l'amélioration des solveurs SAT est souvent un monde cruel. Pour donner une idée, améliorer un solveur en résolvant au moins dix instances supplémentaires (sur un ensemble fixe de benchmarks d'une compétition) est généralement considéré comme une nouvelle fonctionnalité critique. En général, le gagnant d'une compétition est choisi sur la base de quelques instances supplémentaires résolus » [2]. Les résultats indiquent clairement que MAB surpasse les deux heuristiques. En effet, MAB parvient à résoudre 30 instances supplémentaires au total (+3,9%) par rapport à la meilleure heuristique. De plus, bien que les résultats globaux obtenus par VSIDS et CHB soient comparables, ils peuvent avoir des comportements différents sur les benchmarks individuels. Poutant, MAB parvient à capturer le comportement de la meilleure heuristique et même à le surpasser pour chaque benchmark individuel. Les résultats obtenus par MAB sont également très proches du VBS. En particulier, MAB réalise plus de 98% (resp. 99%) des performances du VBS sur l'ensemble du benchmark en termes de nombre d'instances résolues (resp. nombre d'instances satisfiables résolues) alors que la meilleure heuristique ne dépasse pas 95% (resp. 93%).

Cependant, il est important de noter que le gain est principalement au niveau des instances satisfiables alors que, pour les instances insatisfiables, MAB surpasse CHB mais pas VSIDS. Néanmoins, MAB reste compétitif avec VSIDS car il ne résout que 3 instances de moins, avec une instance de moins seulement chaque année. Cela équivaut à une perte de performance de 0,9% par rapport aux instances insatisfiables alors que

1. http ://fmv.jku.at/kissat/

2. traduite à partir de [3] de G. Audemard et L. Simon.

		VSIDS	CHB	RD_N	RD_R	MAB	VBS
Compétition 2018 (400 instances)	SAT	160	159	160	164	**167**	169
	UNSAT	**111**	109	109	110	110	113
	TOTAL	271	268	268	274	**277**	282
Race 2019 (400 instances)	SAT	158	149	155	158	**161**	162
	UNSAT	**97**	95	95	96	96	99
	TOTAL	255	244	250	254	**257**	261
Compétition 2020 (400 instances)	SAT	131	146	146	151	**154**	157
	UNSAT	**121**	119	117	120	120	123
	TOTAL	252	265	263	271	**274**	280
TOTAL (1 200 instances)	SAT	449	454	461	473	**482**	488
	UNSAT	**329**	323	321	326	326	335
	BOTH	778	777	782	799	**808**	823

TABLE 1 – Comparaison entre VSIDS, CHB, RD_N, RD_R, MAB et VBS en termes de nombre d'instances résolues dans Kissat. Pour chaque ligne, les meilleurs résultats obtenus sans considérer le VBS sont écrits en gras.

MAB réalise un gain de 6,7% en termes d'instances satisfiables. Ceci peut être dû à de nombreux facteurs. En effet, les résultats en termes d'instances insatisfiables semblent très homogènes pour chaque année et sont très proches des résultats obtenus par le VBS car les deux heuristiques (resp. la meilleure heuristique) atteignent plus de 96% (resp. 98%) de ses performances. Un autre facteur pourrait être la politique de redémarrage de Kissat qui alterne entre les phases stables et non stables comme c'est le cas dans CaDiCaL [6], renommé en mode stable et mode focalisé dans [7]. Les heuristiques mentionnées ne sont utilisées que dans les phases stables qui ciblent principalement les instances satisfiables. Pendant les phases focalisés, le solveur ne calcule pas les scores et déplace en revanche les variables analysées au début de la file de priorités des décisions. Cela peut également aider à expliquer l'homogénéité des résultats obtenus par les différentes heuristiques (y compris MAB) pour les instances insatisfiables. Un troisième facteur possible, quoique dans une moindre mesure, peut être la fonction de récompense qui oriente en partie la recherche de manière à générer plus de conflits. En effet, générer plus de conflits et donc apprendre plus de clauses sans contrôler leur qualité peut impacter négativement la performance du solveur sur les instances insatisfiables, pour lesquelles il doit explorer tout l'espace de recherche. Or, bien que la fonction de récompense ait été conçue pour générer efficacement des conflits, elle ne peut pas garantir une bonne qualité des clauses apprises.

Dans la suite, on veut évaluer MAB en termes de temps de résolution. Puisque UCB1 mène une exploration continue afin de garantir la sélection du bras le plus adéquat à chaque redémarrage, on pourrait s'attendre à des performances moindre du MAB comparé aux heuristiques considérées en terme de temps de résolution. Or, cela ne semble pas être le cas. En effet, conduire

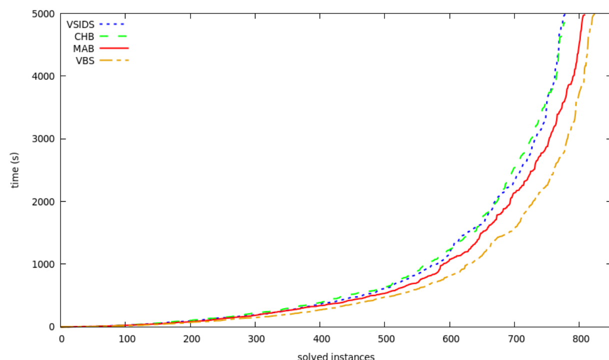

FIGURE 1 – Temps CPU (en secondes) en fonction des instances résolues pour VSIDS, CHB, MAB et VBS.

l'exploitation avec le meilleur bras et alterner les heuristiques semble compenser cet inconvénient. Dans la figure 1, on représente le temps de résolution en fonction des instances résolues du benchmark. On observe que MAB surpasse à la fois VSIDS et CHB. En fait, MAB réalise plus de 4% (resp. 5%) de gain en termes de temps de résolution sur l'ensemble du benchmark par rapport à la meilleure heuristique si l'on donne une pénalité de 5 000 s (resp. 10 000 s, ce qui équivaut à deux fois le temps limite) aux instances non résolues. Ce gain est important d'autant plus qu'on réalise les tests avec le solveur compétitif Kissat, vainqueur de la compétition SAT 2020. Un autre résultat intéressant est la performance de MAB sur des instances « dures », c'est-à-dire dont le temps de résolution dépasse 4 000 s, qui se rapproche beaucoup du VBS comme le montre la figure 1. Les temps de résolution de VSIDS, CHB et MAB sont respectivement 110%, 110,9% et 105,6% plus élevés que le VBS sur l'ensemble du benchmark. Ces résultats montrent que MAB réalise un gain substantiel non seulement en termes de nombre d'instances

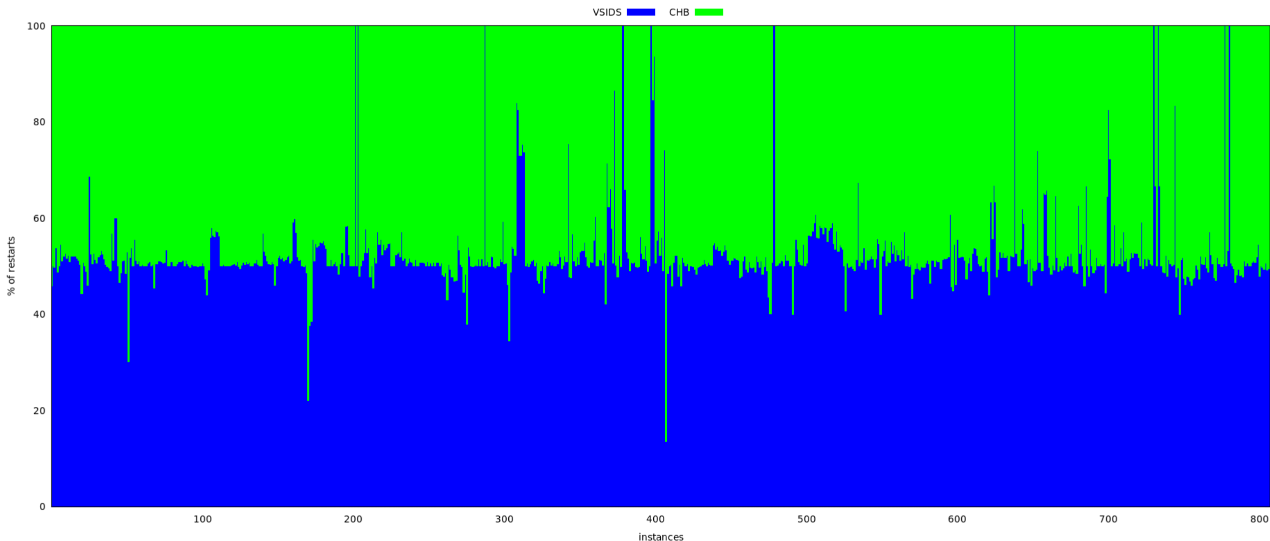

FIGURE 2 – Pourcentages d'utilisation de chaque bras dans MAB par rapport à l'ensemble des instances.

résolues mais aussi en termes de temps de résolution et ses performances sont très proches du VBS.

5.3 Comportement de MAB

Ci-après, on veut analyser plus en détail le comportement du MAB. Tout d'abord, on se focalise sur l'utilisation des bras du bandit. Dans la figure 2, on représente les pourcentages d'utilisation de chaque bras, à savoir VSIDS et CHB. Plus précisément, on représente le pourcentage de redémarrages où chaque bras est choisi par UCB1. On observe que MAB alterne entre les deux heuristiques mais les pourcentages d'utilisation se situent principalement dans l'intervalle $[40\%, 60\%]$ et sont souvent proches de 50%. Il est important de rappeler que les pourcentages d'utilisation des bras sont directement influencés par la politique de redémarrage et par le paramètre c. Cependant, ces résultats ne sont pas surprenants compte tenu du nombre généralement très faible de redémarrages stables dans Kissat au cours desquels les heuristiques VSIDS et CHB sont utilisées. Pour donner une idée, le nombre moyen de redémarrages stables effectués par Kissat sur l'ensemble du benchmark est de 771 alors que la valeur médiane est beaucoup plus faible et s'élève à 313. Par conséquent, les pourcentages obtenus semblent adéquats surtout que le solveur doit parvenir à un bon compromis entre l'exploration et l'exploitation.

De plus, il est important de noter que tirer parti du mécanisme de redémarrage pour combiner VSIDS et CHB via l'utilisation du bandit n'était pas un choix arbitraire. En effet, on a mené une expérience afin de choisir le niveau approprié pour combiner les deux heuristiques grâce au bandit manchot. On a implémenté deux versions de Kissat, RD_N et RD_R, dans lesquelles une heuristique parmi VSIDS et CHB est choisie au hasard respectivement à chaque décision ou à chaque redémarrage. Les résultats moyens (sur 10 exécutions avec des graines différentes) de RD_N et RD_R sont représentés dans le tableau 1 et indiquent que RD_R surpasse RD_N sur l'ensemble du benchmark avec un gain de plus de 2% en termes d'instances résolues et de 40% en termes de temps de résolution (si on attribue une pénalité de 10 000 s pour les instances non résolues). En effet, la combinaison des deux heuristiques au niveau des décisions peut provoquer des interférences et peut ne pas permettre à chaque heuristique de mener un apprentissage robuste puisqu'elles sont constamment interchangées. Étonnamment, les deux versions sont compétitives avec CHB et VSIDS. En particulier, RD$_R$ les surpasse et résout, en moyenne, 21 instances de plus (+2,7%) que la meilleure heuristique. Ceci est dû à la randomisation et à la diversification qui aident à éviter les phénomènes à queue lourde dans SAT et qui peuvent rendre les solveurs plus efficaces [15, 13]. Néanmoins, MAB reste plus efficace que RD$_R$ puisqu'il résout 9 instances supplémentaires (+1,1%) et réalise un gain de 1% en termes de temps de résolution sachant que les résultats individuels obtenus pour RD$_R$ varient et que la différence devient beaucoup plus importante si on considère les tests avec les moindres résultats.

5.4 MAB et familles d'instances

On termine cette section par une analyse plus approfondie des résultats obtenus par MAB sur les familles

Famille		VSIDS		CHB		MAB		VBS	
nom	#inst	#inst	temps	#inst	temps	#inst	temps	#inst	temps
Antibandwidth	14	2	1 010	7	9 804	9	15 651	7	9 628
Keystream Generator Cryptanalysis	18	18	14 494	14	15 104	18	13 118	18	19 240
Baseball-lineup	13	18	3 317	12	2 949	12	2 947	12	2 906
Core-based	14	13	8 438	13	10 860	14	14 165	13	7 239
Chromatic Number (CNP)	20	20	1 708	20	1 972	20	1 180	20	1 194
Edge-Matching Puzzle †	14	3	4 476	3	6 201	4	8 674	4	8 823
Logical Cryptanalysis	20	20	5 606	20	10 476	20	4 241	20	4 946
Hgen	13	12	3 168	12	2 423	12	783	12	2 365
Course Scheduling	20	14	14 439	14	15 363	15	9 323	14	9 654
Relativized Pigeonhole Principle (RPHP)	20	11	14 890	10	11 344	11	14 065	11	14 890
Station Repacking	12	6	15 286	12	10 656	12	8 766	12	10 656
Stedman Triples †	27	10	8 766	11	11 508	12	7 399	11	6 947
Timetable †	26	1	1 565	10	5 082	11	6 247	10	5 082
Vlsat	14	3	103	7	4 457	7	500	7	3 934

TABLE 2 – Comparaison entre VSIDS, CHB, MAB et VBS en termes de nombre d'instances résolues et de temps de résolution cumulé en secondes (pour les instances résolues) dans Kissat pour certaines familles d'instances du benchmark. Les résultats des familles marquées par † sont joints à partir de deux benchmarks annuels différents.

d'instances. Dans le tableau 2, on décrit quelques résultats intéressants obtenus sur certaines familles d'instances au sein du benchmark considéré [17, 16, 5] pour lesquelles MAB surpasse à la fois CHB et VSIDS en termes d'instances résolues ou de temps de résolution, ou les deux. Plus important encore, pour certaines familles, MAB parvient à obtenir de meilleurs résultats, principalement en termes de temps de résolution, que ceux obtenus par le VBS. Par exemple, MAB améliore considérablement le temps de résolution des familles `ASG` (-54%), `Hgen` (-66,9%), `Station Repacking` (-17%) et `Vlsat` (-87,3%) par rapport au VBS. De plus, MAB est également capable de résoudre des instances qui n'ont pas été résolues par le VBS comme dans les familles `Antibandwidth`, `Core-based`, `Polynomial Multiplication`, `Ofer` et `Stedman Triples`.

Enfin, on discute brièvement du choix de combiner VSIDS et CHB même si MAB permet d'inclure plus d'heuristiques. En effet, on pourrait argumenter que l'ajout d'autres heuristiques pourrait permettre de traiter avec succès plus de familles et d'instances grâce à la diversification. Cependant, il faut noter que les solveurs SAT modernes, et en particulier Kissat, sont très compétitifs et s'appuient sur des heuristiques puissantes pour obtenir des résultats impressionnants. Une mauvaise heuristique ou un mauvais réglage des paramètres peut considérablement détériorer les performances d'un solveur. De plus, pratiquement toutes les heuristiques utilisées dans les solveurs SAT modernes sont des variantes de VSIDS, qui a été l'heuristique dominante depuis son introduction en 2001 [27]. Seulement récemment, l'heuristique CHB a été introduite et a été démontrée compétitive avec VSIDS [22]. Les résultats reportées dans le tableau 1 montrent également que CHB peut atteindre de nouvelles instances (le VBS réalise un gain de plus de 5,7% en termes d'instances résolues) tout en restant globalement compétitive et comparable à VSIDS sur les benchmarks des dernières compétitions. Un autre facteur qui peut rendre inefficace l'augmentation du nombre de bras est le nombre très faible de redémarrages et, en particulier, de redémarrages stables dans Kissat. En effet, augmenter le nombre d'heuristiques rendrait le solveur incapable d'identifier le bras adéquat à chaque redémarrage en raison du manque d'essais. On pourrait également penser que l'augmentation du nombre de redémarrages en modifiant les paramètres du solveur peut facilement résoudre ce problème. Ce n'est pas le cas, car la modification des paramètres des solveurs compétitifs, qui sont généralement fixés finement après un réglage approfondi, entraîne généralement une détérioration drastique de leurs performances. C'est pourquoi, après de nombreuses expérimentations sur Kissat, la diversification qui peut être obtenue en combinant plusieurs heuristiques via MAB, y compris des heuristiques non compétitives, va détériorer les performances du solveur.

6 Conclusion

Dans cet article, on a proposé d'utiliser une approche par bandit manchot pour combiner deux heuristiques de l'état de l'art, à savoir VSIDS et CHB. Le bandit tire parti du mécanisme de redémarrage pour sélectionner une heuristique pertinente tout en maintenant un bon équilibre entre l'exploration et l'exploitation. En outre, les heuristiques sont évaluées sur leur capacité à résoudre les conflits rapidement et efficacement et sont sélectionnées par le biais de la politique Upper Confidence Bound (UCB). L'évaluation expérimentale menée montre que MAB surpasse les heuristiques considérées à la fois en termes d'instances résolues, principalement

les instances satisfiables, et en termes de temps de résolution. Le gain est d'autant plus important qu'on a réussi à améliorer Kissat qui est non seulement un solveur de référence (réimplémentation de CaDiCaL en C) mais aussi un solveur des plus compétitifs qui a remporté la compétition SAT 2020. De plus, MAB obtient des résultats très proches du VBS sur VSIDS et CHB et parvient même à les surpasser pour certaines familles d'instances.

Comme perspective de ce travail, il serait intéressant d'affiner la fonction de récompense en s'appuyant sur une combinaison de différents critères [9] afin d'améliorer la performance du bandit, notamment sur les instances insatisfiables. Il serait également intéressant de se focaliser sur une heuristique et d'essayer de la raffiner en utilisant le bandit manchot, une approche qui s'est avérée pertinente dans le contexte du problème de satisfaction des contraintes [8]. Enfin, il serait intéressant de concevoir une approche similaire pour améliorer d'autres composants des solveurs SAT modernes tels que la suppression de clauses apprises [2].

Références

[1] Rajeev AGRAWAL : Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.

[2] Gilles AUDEMARD et Laurent SIMON : Predicting learnt clauses quality in modern SAT solvers. *In Proceedings of the International Joint Conference on Artificial Intelligence*, 2009.

[3] Gilles AUDEMARD et Laurent SIMON : Refining restarts strategies for sat and unsat. *In International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[4] Peter AUER, Nicolò CESA-BIANCHI et Paul FISCHER : Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[5] Tomáš BALYO, Nils FROLEYKS, Marijn HEULE, Markus ISER, Matti JÄRVISALO et Martin SUDA : Proceedings of SAT Competition 2020 : Solver and Benchmark Descriptions. 2020.

[6] Armin BIERE : CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. *In* Tomáš BALYO, Marijn HEULE et Matti JÄRVISALO, éditeurs : *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 de *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[7] Armin BIERE, Katalin FAZEKAS, Mathias FLEURY et Maximillian HEISINGER : CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. *In* Tomas BALYO, Nils FROLEYKS, Marijn HEULE, Markus ISER, Matti JÄRVISALO et Martin SUDA, éditeurs : *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 de *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[8] Mohamed Sami CHERIF, Djamal HABET et Cyril TERRIOUX : On the refinement of conflict history search through multi-armed bandit. *In* Miltos ALAMANIOTIS et Shimei PAN, éditeurs : *Proceedings of 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 264–271. IEEE, 2020.

[9] Wei CHU, Lihong LI, Lev REYZIN et Robert SCHAPIRE : Contextual Bandits with Linear Payoff Functions. *In Proceedings of International Conference on Artificial Intelligence and Statistics*, pages 208–214, 2011.

[10] Stephen A. COOK : The Complexity of Theorem-proving Procedures. *In Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC'71, pages 151–158, New York, NY, USA, 1971. ACM.

[11] Todd DESHANE, Wenjin HU, Patty JABLONSKI, Hai LIN, Christopher LYNCH et Ralph Eric MCGREGOR : Encoding first order proofs in SAT. *In Proceedings of the International Conference on Automated Deduction*, pages 476–491, 2007.

[12] Niklas EÉN et Niklas SÖRENSSON : An extensible SAT-solver. *In Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[13] Carla P GOMES, Bart SELMAN, Nuno CRATO et Henry KAUTZ : Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100, 2000.

[14] Shai HAIM et Marijn HEULE : Towards Ultra Rapid Restarts. *CoRR*, abs/1402.4413, 2014.

[15] William D. HARVEY et Matthew L. GINSBERG : Limited discrepancy search. *In IJCAI'95*, page 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[16] Marijn HEULE, Matti JÄRVISALO et Martin SUDA : Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions. 2019.

[17] Marijn HEULE, Matti Juhani JÄRVISALO, Martin SUDA *et al.* : Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions. 2018.

[18] T. Hong, Y. Li, S. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner et S. Mitra : QED : Quick Error Detection tests for effective post-silicon validation. *In Proceedings of the International Test Conference*, pages 154–163, 2010.

[19] Vitaly Kurin, Saad Godil, Shimon Whiteson et Bryan Catanzaro : Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. *CoRR*, 2019.

[20] Tze Leung Lai et Herbert Robbins : Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[21] Nadjib Lazaar, Youssef Hamadi, Said Jabbour et Michèle Sebag : Cooperation control in Parallel SAT Solving : a Multi-armed Bandit Approach. Research Report RR-8070, INRIA, septembre 2012.

[22] Jia Hui Liang, Vijay Ganesh, Pascal Poupart et Krzysztof Czarnecki : Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. *In Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3434–3440, 2016.

[23] Jia Hui Liang, Vijay Ganesh, Pascal Poupart et Krzysztof Czarnecki : Learning rate based branching heuristic for SAT solvers. *In Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140, 2016.

[24] Michael Luby, Alistair Sinclair et David Zuckerman : Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[25] Inês Lynce et João P. Marques-Silva : SAT in Bioinformatics : Making the Case with Haplotype Inference. *In Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pages 136–141, 2006.

[26] João P Marques-Silva et Karem A Sakallah : Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[27] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang et Sharad Malik : Chaff : Engineering an efficient SAT solver. *In Proceedings of the Design Automation Conference*, pages 530–535, 2001.

[28] Anastasia Paparrizou et Hugues Wattez : Perturbing branching heuristics in constraint solving. *In* Helmut Simonis, éditeur : *Principles and Practice of Constraint Programming*, pages 496–513, Cham, 2020. Springer International Publishing.

[29] Richard S. Sutton et Andrew G. Barto : *Reinforcement Learning : An Introduction.* MIT Press, Cambridge, MA, USA, 1st édition, 1998.

[30] William R Thompson : On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4): 285–294, 1933.

[31] Toby Walsh : Search in a Small World. *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1172–1177, 2002.

[32] Hugues Wattez, Frederic Koriche, Christophe Lecoutre, Anastasia Paparrizou et Sébastien Tabary : Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. *In Proceedings of the European Conference on Artificial Intelligence*, 2020.

[33] Wei Xia et Roland H. C. Yap : Learning Robust Search Strategies Using a Bandit-Based Approach. *In Proceedings of the AAAI Conference on Artificial Intelligence*, pages 6657–6665, 2018.