Journal on Satisfiability, Boolean Modeling and Computation 4 (2008) 117-148

Whose side are you on?*

Finding solutions in a biased search-tree

Marijn J.H. Heule† marijn@heule.nl
Hans van Maaren h.vanmaaren@ewi.tudelft.nl
Department of Software Technology,
Delft University of Technology,
The Netherlands

Abstract

We introduce a new jump strategy for look-ahead based satisfiability (SAT) solvers that aims to boost their performance on satisfiable formulae, while maintaining their behavior on unsatisfiable instances.

Direction heuristics select which Boolean value to assign to a decision variable. They are used in various state-of-the-art SAT solvers and may bias the distribution of solutions in the search-tree. This bias can clearly be observed on hard random k-SAT formulae. While alternative jump / restart strategies are based on exploring a random new part of the search-tree, the proposed one examines a part that is more likely to contain a solution based on these observations.

Experiments with - so-called *distribution jumping* - in the march_ks SAT solver, show that this new technique boosts the number of satisfiable formulae it can solve. Moreover, it proved essential for winning the satisfiable crafted category during the SAT 2007 competition.

KEYWORDS: *direction heuristics, jumping strategy, look-ahead SAT solvers*

Submitted October 2007; revised January 2008; published May 2008

1. Introduction

Various state-of-the-art satisfiability (SAT) solvers use *direction heuristics* to predict the sign of the decision variables: These heuristics choose, after the selection of the decision variable, which Boolean value is examined first. Direction heuristics are in theory very powerful: If always the correct Boolean value is chosen, satisfiable formulae would be solved without backtracking. Moreover, existence of perfect direction heuristics (computable in polynomial time) would prove that $\mathcal{P} = \mathcal{NP}$.

On some families these heuristics bias the location of solutions in the search-tree. Given a large family with many satisfiable instances, this bias can be measured on small instances. The usefulness of this depends on what we call the *bias-extrapolation property*: Given the direction heuristics of a specific solver, the observed bias on smaller instances extrapolates

∗ This paper is an extended version of: Marijn J.H. Heule and Hans van Maaren. *Whose side are you on? Finding solutions in a biased search-tree.* Proceedings of Guangzhou Symposium on Satisfiability In Logic-Based Modeling (2006), 82-89.

† Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.611.

©2008 Delft University of Technology and the authors.

to larger ones. Notice that this notion depends on the *action* of a particular solver on the family involved: i.e. a solver with random direction heuristics may also satisfy the bias extrapolation property, but not in a very useful way - probably, there is no bias at all. In case the estimated bias shows a logical pattern, it could be used to consider a jumping strategy that adapts towards the distribution of the solutions. We refer to this strategy as *distribution jumping*.

Other jump strategies have been developed for Sat solvers. The most frequently used technique is the *restart strategy* [4]: If after some number of backtracks no solution has been found, the solving procedure is restarted with a different decision sequence. This process is generally repeated for an increasing number of backtracks. This technique could fix an ineffective decision sequence. A disadvantage of restarts is its potential slowdown of performance on unsatisfiable instances, especially on look-ahead Sat solvers. However, conflict driven Sat solvers on average improve their performance using restarts.

Another jumping strategy is the *random jump* [10]. Instead of jumping all the way to the root of the search-tree, this technique jumps after some backtracks to a random level between the current one and the root. This technique could fix a wrongfully chosen sign of some of the decision variables. By storing the subtrees that have been visited the performance on unsatisfiable instances is only slightly reduced for look-ahead Sat solvers.

Both these techniques are designed to break free from an elaborate subtree in which the solver is "trapped". Our proposed technique not only jump out of such a subtree but also towards a subtree with a high probability of containing a solution. Unlike restart strategies, random jumping and distribution jumping only alter the signs of decision variables. Therefore, for look-ahead Sat solvers, the performance on unsatisfiable formulae does not influence (besides some minor overhead) costs. Yet, for conflict driven Sat solvers the performance on unsatisfiable instances is affected, as well.

The outline of this paper is as follows: Section 2 introduces the direction heuristics used in complete (DPLL based) Sat solvers. In each step, these solvers select a decision variable (decision heuristics). Whether to first visited the *positive branch* (assigning the decision variable to true) or the *negative branch* (assigning the decision variable to false) is determined by these direction heuristics. These heuristics can heavily influence the performance on satisfiable benchmarks. In case conflict clauses are added, the performance on unsatisfiable instances is affected, as well.

Section 3 studies the influence of existing direction heuristics. More specific, we ask ourselves the question: Given a (complete) Sat solver and a benchmark family, is the *distribution of solutions* in the search-tree biased (not uniform)? A possible bias is caused by the direction heuristics used in the Sat solver. We offer some tools to visualize, measure and compare the possible bias for different Sat solvers on hard random k-Sat formulae. We selected this family of formulae because it is well studied and one can easily generate many hard instances for various sizes.

The bias we observe, can intuitively be explained: Since the direction heuristics discussed in this paper aim to select the heuristically most satisfiable subtree (referred to as the *left branch*) first, a bias towards the left branches is expected and observed. Near the root of the search-tree, the considered (reduced) formulae are larger and more complex compared to those lower in the tree. Therefore, it is expected that the effectiveness of direction heuristics

improves (and thus the bias towards the left branches increases) in nodes deeper in the tree - which is also observed.

Section 4 discusses the possibilities to capitalize on a given / observed bias. We focus on the observed bias of look-ahead Sat solvers k-Sat formulae. We developed a new jump strategy, called *distribution jumping*, which visits subtrees in decreasing (observed) probability of containing a solution. We show that using the proposed (generalized) order of visiting subtrees - compared to visits them in chronological order - results in a significant speed-up in theory.

We implemented this new technique in march_ks and Section 5 offers the results. These results show performance gains on random k-Sat formulae. On many satisfiable structured benchmarks improvements were observed, as well. Due to this technique, the look-ahead Sat solver march_ks won the satisfiable crafted family of the Sat 2007 competition. Finally some conclusions are drawn in Section 6.

2. Direction heuristics

All state-of-the-art complete Sat solvers are based on the DPLL architecture [2]. This recursive algorithm (see Algorithm 1) first simplifies the formula by performing unit propagation (see Algorithm 2) and checks whether it hits a leaf node. Otherwise, it selects a decision variable x_{decision} and splits the formula into two subformulae where x_{decision} is forced - one for positive (denoted by $\mathcal{F}(x_{\text{decision}} = 1)$) and one for negative ($\mathcal{F}(x_{\text{decision}} = 0)$).

Algorithm 1 DPLL($\mathcal{F}$)

```
1: F := UnitPropagation( F )
2: if F = ∅ then
3:    return satisfiable
4: else if empty clause ∈ F then
5:    return unsatisfiable
6: end if
7: l_decision := SelectDecisionLiteral( F )
8: if DPLL( F(l_decision = 1) ) = satisfiable then
9:    return satisfiable
10: else
11:    return DPLL( F(l_decision = 0) )
12: end if
```

Algorithm 2 UnitPropagation($\mathcal{F}$)

```
1: while F does not contain an empty clause and unit clause y exists do
2:    satisfy y and simplify F
3: end while
4: return F
```

The search-tree of a DPLL based Sat solver can be visualized as a binary search-tree. Figure 1 shows such a tree with decision variables drawn in the internal nodes. Edges show the type of each branch. A black leaf refers to a unsatisfiable dead end, while a white leaf indicates that a satisfying assignment has been found. An internal node is colored black in case both its children are black, and white otherwise. For instance, if we look at depth 4 of this search-tree, three nodes are colored white meaning that at depth 4 there exist 3 subtrees containing a solution.

Two important heuristics emerge for this splitting: *variable selection heuristics* and *direction heuristics*. Both heuristics are merged in the procedure SelectDecisionLiteral. Variable selection heuristics aim to select a decision variable in each recursion step yielding a relatively small search-tree. Direction heuristics aim to find a satisfying assignment as fast as possible by choosing which subformula - $\mathcal{F}(x_{\text{decision}} = 0)$ or $\mathcal{F}(x_{\text{decision}} = 1)$ - to examine first. In theory, direction heuristics could be very powerful: If one always predicts the correct direction, all satisfiable formulae will be solved in a linear number of decisions.

Traditionally, Sat research tends to focus on variable selection heuristics. Exemplary of the lack of interest in direction heuristics is its use in the conflict-driven Sat solver minisat [3]: While this solver is the most powerful on a wide range of instances, it always branches negatively - computes $\mathcal{F}(x_{\text{decision}} = 0)$ first. An explanation for the effectiveness of this heuristic may be found to the general encoding of most (structural) Sat formulae. Also, these direction heuristics are more sophisticated then they appear: Choosing the same sign consequently is - even on random formulae - much more effective than a random selection [9].

Two well-known methods to select the decision literal are the one-sided and two-sided Jeroslow-Wang heuristics [6]. Both compute for each literal l the following measurement:

$$J(l) = \sum_{l \in C_i} 2^{-|C_i|}$$

Using the one-side Jeroslow-Wang heuristic, the literal with highest $J(l)$ is selected as decision literal. The two-sided Jeroslow-Wang heuristic first selects the variable x_i with the highest $J(x_i) + J(\neg x_i)$ as decision variable. Then it selects the positive branch if $J(x_i) \geq J(\neg x_i)$, otherwise the negative branch is preferred.

Throughout this paper, we will discuss only Sat solving techniques that do not add global constraints such a conflict clauses. So, only chronological backtracking is considered. Also, given a Sat solver and a certain formula, the complete search-tree will always be identical: Visiting (leaf)nodes in a different order will not affect the binary representation as shown in Figure 1. In case the formula is satisfiable, the order in which (leaf)nodes are visited only influences the fraction of the search-space that has to be explored to find a (first) solution.

We will focus on the direction heuristics used in look-ahead Sat solvers. This choice is motivated by the strong performance of these solvers on random k-Sat formulae (the ones we selected for our experiments). These solvers spend a lot of reasoning in selecting the decision literal. Meanwhile, they also aim to detect forced variables (variables that must be assigned to a certain value to avoid a conflict). By assigning these forced variables the formula shrinks which yields a smaller search-tree.

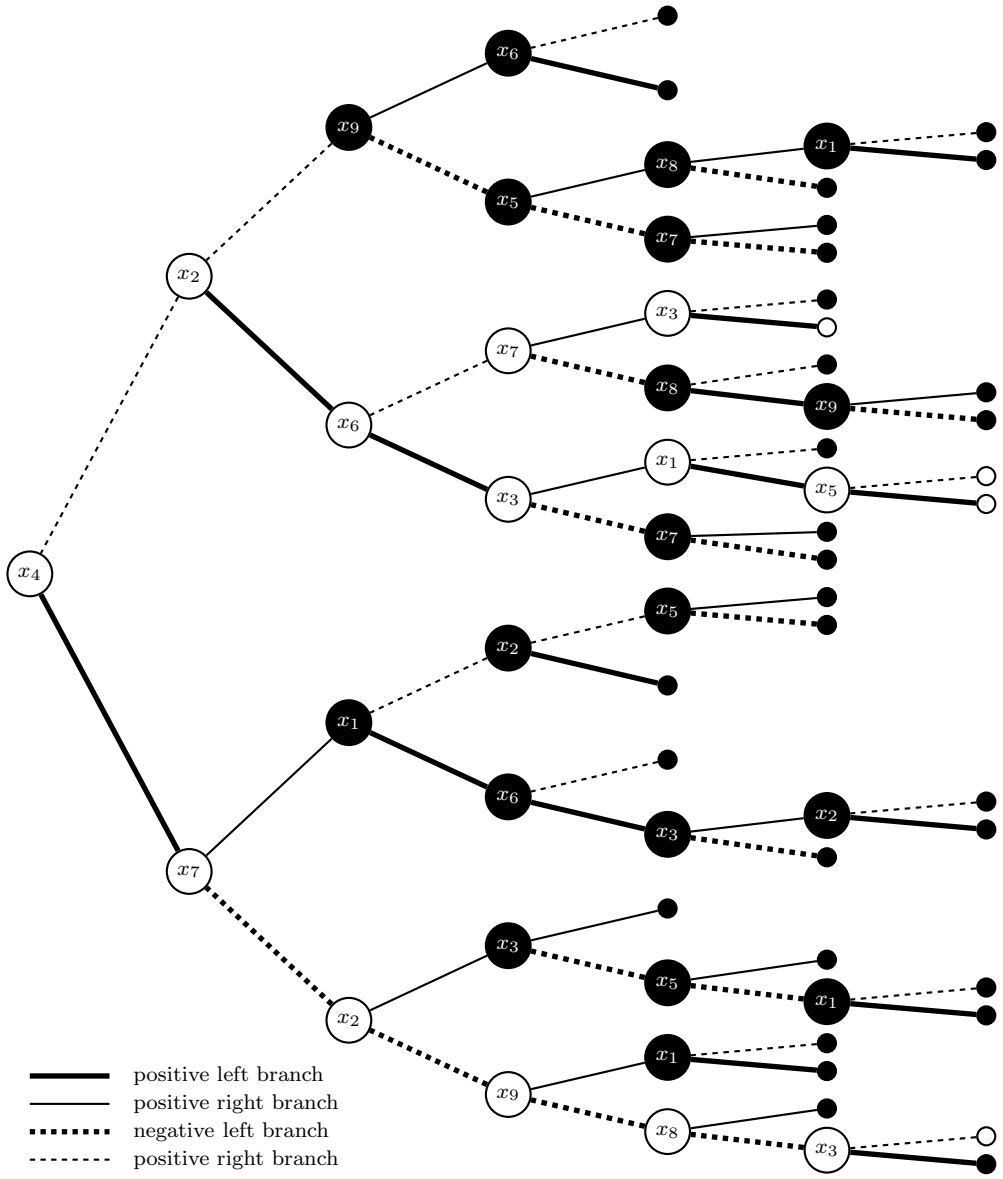

Figure 1. Complete binary search-tree (DPLL) for a formula with nine variables ($x_1, \ldots, x_9$). The decision variables are shown inside the internal nodes. A node is colored black if all child nodes are unsatisfiable, and white otherwise. The type of edge shows whether it is visited first (left branch), visited last (right branch), its decision variable is assigned to true (positive branch), or its decision variable is assigned to false (negative branch).

A similar direction heuristic as the Jeroslow-Wang heuristics is used in the look-ahead Sat solver kcnfs[1]. It aims selecting the most satisfiable branch. It compares the difference of occurrences between x_{decision} and $\neg x_{\text{decision}}$. However, occurrences are not weighted as the Jeroslow-Wang heuristics. The larger of these two satisfies more clauses in which the decision variable occurs and is therefore preferred.

The look-ahead Sat solver march_ks bases its direction heuristics on the reduction caused by the decision variable [5]. The reduction from $\mathcal{F}$ to $\mathcal{F}(x_{\text{decision}} = 0)$ and from $\mathcal{F}$ to $\mathcal{F}(x_{\text{decision}} = 1)$ is measured by the number of clauses that are reduced in size without being satisfied. In general, the stronger this reduction the higher the probability the subformula is unsatisfiable. Therefore, march_ks always branches first on the subformula with the smallest reduction.

Oliver Kullmann proposes direction heuristics (used in his look-head OKsolver) based on the subformula ($\mathcal{F}(x_{\text{decision}} = 0)$ or $\mathcal{F}(x_{\text{decision}} = 1)$) with the lowest probability that a random assignment will falsify a random formula of the same size [7]. Let $\mathcal{F}_k$ denote the set of clauses in $\mathcal{F}$ of size k. It prefers either $\mathcal{F}(x_{\text{decision}} = 0)$ or $\mathcal{F}(x_{\text{decision}} = 1)$ for which the following is smallest:

$$\sum_{k \geq 2} -|\mathcal{F}_k| \cdot \ln(1 - 2^{-k}) \tag{1}$$

3. Observed bias on random k-Sat formulae

This section studies the effectiveness of existing direction heuristics of Sat solvers based on the DPLL architecture. Here we will provide a large study of different solvers on random k-Sat formulae with different sizes and densities. The main motivation to use these formulae is that one can easily create many instances of different sizes and hardness. Therefore, this family of formulae seems an obvious candidate to test whether the direction heuristics used in some Sat solvers satisfy the bias-extrapolation property. We focus on the hard random k-Sat instances - near the (observed) phase transition density. The concepts introduced in this section are developed to offer some insights in the effectiveness of direction heuristics.

3.1 Distribution of solutions

We determined the bias of the distribution of solutions amongst the various subtrees using the following experiment: Consider all the subtrees $T_{d,i}$ which are at depth d. Assuming that the search-tree is big enough, there are 2^d of these subtrees. Given a set of satisfiable formulae, what is the probability that a certain subtree contains a solution? Let the left branch in a node denote the subformula - either $\mathcal{F}(x_i = 0)$ or $\mathcal{F}(x_i = 1)$ - which a solver decides to examine first. Consequently, we refer to the right branch as the latter examined subformula.

Subtrees are numbered from left to right starting with $T_{d,0}$ (see Figure 2 for an example with $d = 3$). We generated sets of random k-Sat formulae for various sizes of the number of variables (denoted by n) and for different densities (clause-variable ratio, denoted by ρ). For each set, 10.000 formulae (satisfiable and unsatisfiable) were generated from which we discarded the unsatisfiable instances.

1. SAT 2004 competition version

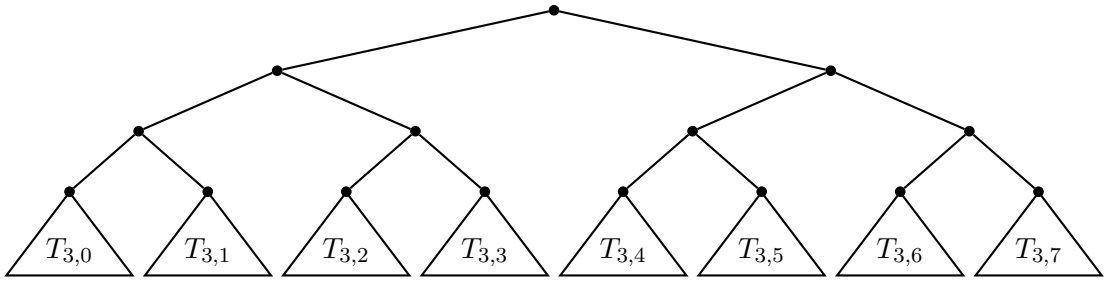

Figure 2. A search-tree with jump depth 3 and 8 subtrees T_i

Definition: The *satisfying subtree probability* $P_{\text{sat}}(d, i)$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the fraction of satisfiable instances that have at least one solution in $T_{d,i}$.

Definition: The *satisfying subtree mean* $\mu_{\text{sat}}(d)$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the average number of subtrees at depth d that have at least one solution.

We compute $\mu_{\text{sat}}(d)$ as

$$\mu_{\text{sat}}(d) = \sum_{i=0}^{2^d-1} P_{\text{sat}}(d, i) \tag{2}$$

By definition $P_{\text{sat}}(0, 0) = 1$ and $\mu_{\text{sat}}(0) = 1$. Because formulae could have solutions in both $T_{d,2i}$ and $T_{d,2i+1}$, $\mu_{\text{sat}}(d)$ is increasing (for increasing d). Or more formal:

$$P_{\text{sat}}(d, 2i) + P_{\text{sat}}(d, 2i + 1) \geq P_{\text{sat}}(d - 1, i) \text{ , and thus} \tag{3}$$

$$\mu_{\text{sat}}(d) \geq \mu_{\text{sat}}(d - 1) \tag{4}$$

Given a SAT solver and a set of satisfiable benchmarks, we can estimate for all 2^d subtrees $T_{d,i}$ the probability $P_{\text{sat}}(d, i)$. A histogram showing the $P_{\text{sat}}(12, i)$ values using march_ks on the test set with $n = 350$ and $\rho = 4.26$ is shown in Figure 3. We refer to such a plot as to the *solution distribution histogram*. The horizontal axis denotes the subtree index i of $T_{12,i}$, while the vertical axis provides the satisfying subtree probability.

Colors visualize the number of right branches (denoted as $\#RB$) required to reach a subtree: $\#RB(T_{d,0}) = 0$, $\#RB(T_{d,1}) = 1$, $\#RB(T_{d,2}) = 1$, $\#RB(T_{d,3}) = 2$,$\#RB(T_{d,4}) = 1$ etc. The figure clearly shows that the distribution is biased towards the left branches: The highest probability is $P_{\text{sat}}(12, 0)$ (zero right branches), followed by $P_{\text{sat}}(12, 2048)$, $P_{\text{sat}}(12, 1024)$, and $P_{\text{sat}}(12, 256)$ - all reachable by one right branch.

A solution distribution histogram of $P_{\text{sat}}(12, i)$ using kcnfs on the same benchmark set is shown in Figure 4. Similar to the histogram using march_ks, the $P_{\text{sat}}(12, i)$ values are higher if $T(12, i)$ can be reached in less right branches. However, the high (peak) $P_{\text{sat}}(12, i)$ values are about 50% higher for march_ks than for kcnfs, while the $\mu_{\text{sat}}(10)$ values for both solvers

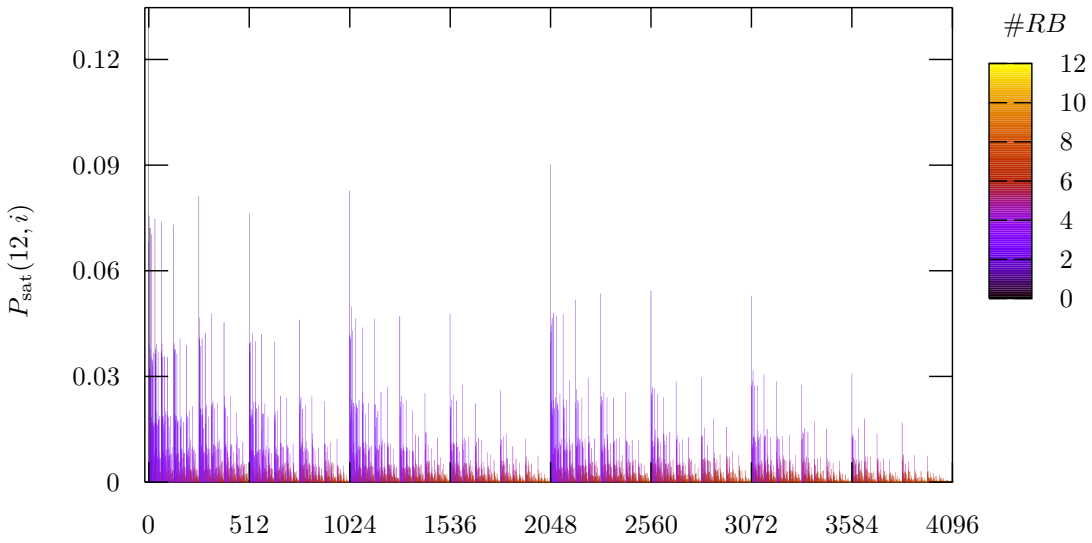

Figure 3. Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using march_ks on 10.000 random 3-SAT formulae with $n = 350$ and $\rho = 4.26$. For this experiment, $\mu_{\text{sat}}(12) = 19.534$.

does not differ much. So, the low $P_{\text{sat}}(12, i)$ values must be higher for kcnfs. This can be observed in the more dense part down in the histogram. Since the same test set is used, based on the lower peak $P_{\text{sat}}(12, i)$ values we can conclude that the direction heuristics used in kcnfs result in a smaller bias to the left branches on these instances.

Again, we see (Figure 5) a bias towards the left branches if we look at the solution distribution histogram of march_ks on random 4-SAT formulae near the phase transition density (in this case $\rho = 9.9$). Also, the high $P_{\text{sat}}(12, i)$ values for march_ks on random 3-SAT are much higher than on random 4-SAT formulae. However, the $\mu_{\text{sat}}(12)$ value is much smaller too. So, the lower peaks might be caused by the fewer number of satisfiable subtrees. Therefore, we cannot easily conclude that the direction heuristics used in march_ks result in a larger bias on random 3-SAT formulae.

Appendix A shows solution distribution histograms for various sizes and densities of random 3-SAT formulae obtained using march_ks. First and foremost, we see in all solution distribution histograms the characteristic bias towards the left branches. From this observation the claim seems justified that the actions of march_ks on random 3-SAT formulae satisfy the bias extrapolation property. Another similarity is that the peak $P_{\text{sat}}(10, i)$ values are about the same size for test sets with the same density. Regarding $\mu_{\text{sat}}(10)$ values, the we observe that $\mu_{\text{sat}}(10)$ is larger if the number of variables is larger. Also, amongst formulae with the same ρ, $\mu_{\text{sat}}(10)$ is higher while the the peak $P_{\text{sat}}(10, i)$ values are comparable. So, the $\mu_{\text{sat}}(10)$ values are higher because of higher low $P_{\text{sat}}(10)$ values. This can be observed in the histograms by the more dense in the lower sections of the plots.

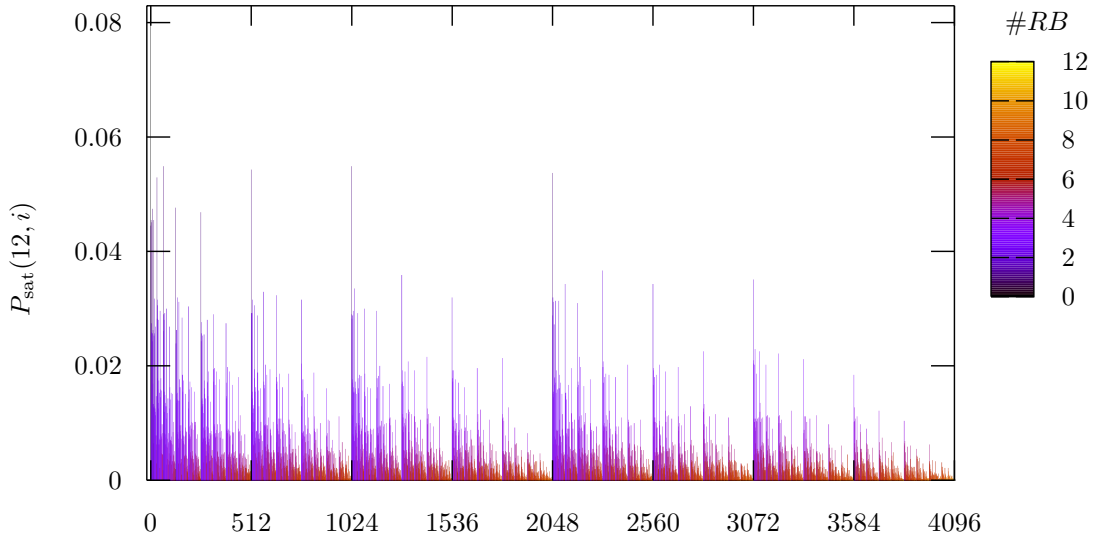

Figure 4. Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using kcnfs on 10.000 random 3-SAT formulae with $n = 350$ and $\rho = 4.26$. For this experiment, $\mu_{\text{sat}}(12) = 18.021$.

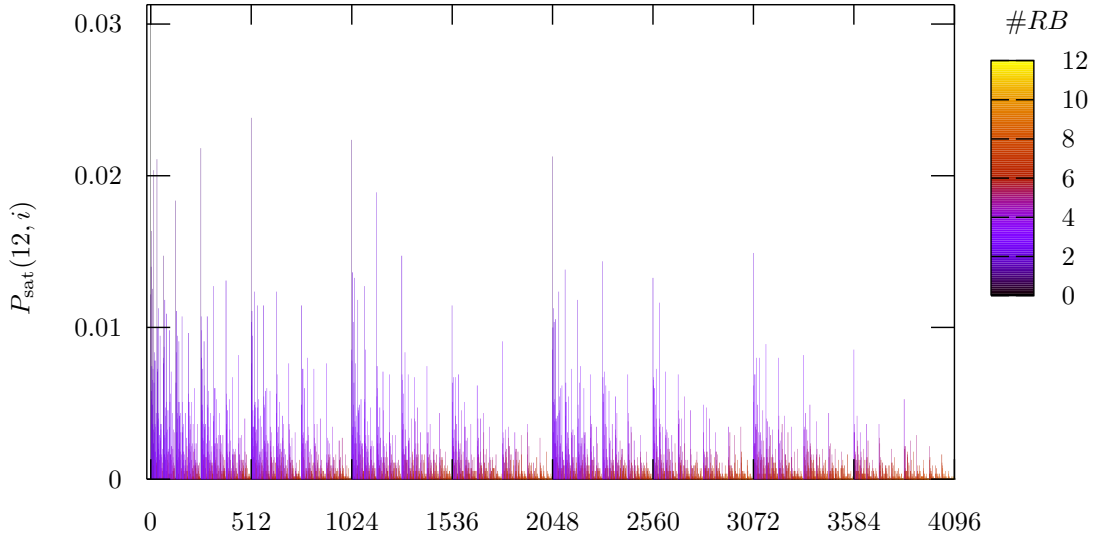

Figure 5. Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using march_ks on 10.000 random 4-SAT formulae with $n = 120$ and $\rho = 9.9$. For this experiment, $\mu_{\text{sat}}(12) = 4.817$.

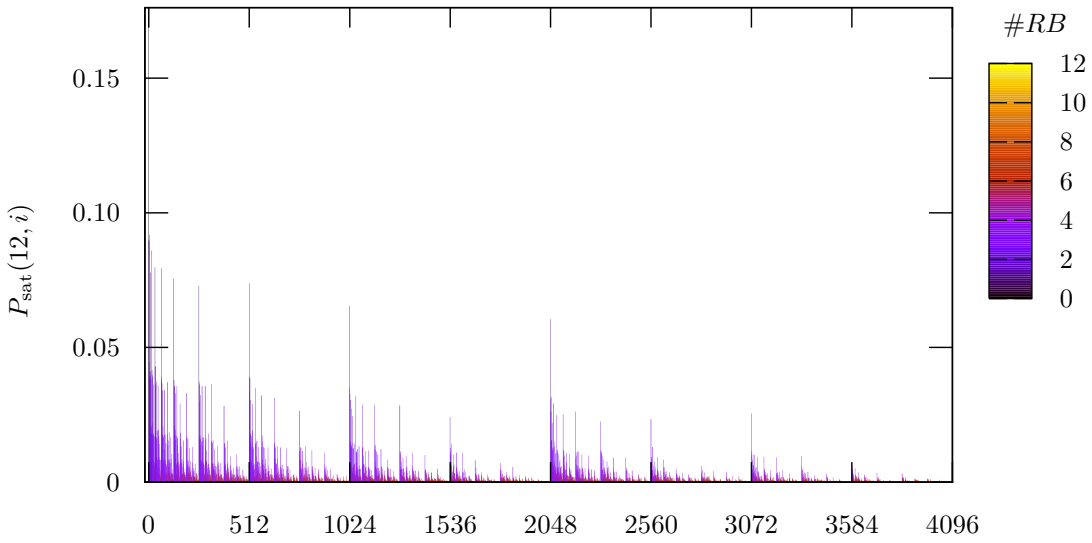

Figure 6. Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using the Jeroslow-Wang one-sided heuristic on 10.000 random 3-SAT formulae with $n = 120$ and $\rho = 4.26$.

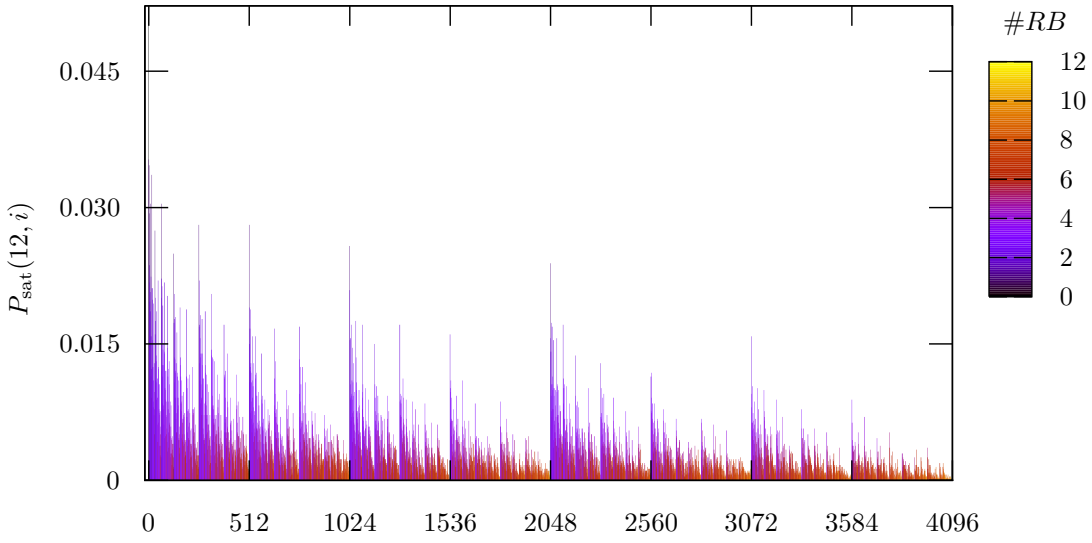

Figure 7. Solution distribution histogram showing $P_{\text{sat}}(12, i)$ using the Jeroslow-Wang two-sided heuristic on 10.000 random 3-SAT formulae with $n = 120$ and $\rho = 4.26$.

We also studied the effectiveness of both Jeroslow-Wang heuristics on random 3-SAT formulae. It must be noted that these heuristics - without the additional reasoning techniques used in state-of-the-art look-ahead solvers - are not very strong on solving these kind of formulae. Therefore, we had to experiment with much smaller sized instances with $n = 120$ and $\rho = 4.26$. Recall that the difference between the these two heuristics is the selection of the decision variable: The one-sided heuristic selects variable x_i such that $\texttt{max}(J(x_i), J(\neg x_i))$ is maximal, while the two-sided heuristic prefers the variable x_i with the highest $J(x_i) + J(\neg x_i)$. Both branch positive if $J(x_i) \geq J(\neg x_i)$, and negative otherwise.

The results for the one-sided and two-sided Jeroslow-Wang heuristics are shown in Figure 6 and 7, respectively. Comparing these figures, we see that the peak $P_{\text{sat}}(12,i)$ values are about three times larger using the one-sided Jeroslow-Wang heuristic. Notice that the two-sided heuristic selects decision variables x_i for which $J(x_i)$ and $J(\neg x_i)$ are more balanced. Selecting balanced decision variables - which is also an important focus of decision variable selection heuristics used in look-ahead SAT solvers - yield in general smaller search-trees. However, focussing on less balanced variables may increases the bias caused by the direction heuristics.

Although the Jeroslow-Wang one-sided heuristic appears to result in a relatively large bias towards the left branches, this heuristic is not very suitable for solving random k-SAT formulae: The variable selection heuristic results in large search-trees. Therefore, the actual solving time is much larger compared to those of look-ahead SAT solvers.

3.2 Satisfying subtree bias

We observed that the distribution of solutions (while experimenting with random k-SAT formulae) is biased towards the left branches due to the direction heuristics used in some SAT solvers. Although there is a great deal of resemblance between the various solution distribution histograms, the magnitude of the $P_{\text{sat}}(d, i)$ values differs. Yet, this magnitude does not easily translate into a comparable bias. To express the differences, we introduce a measurement called the *satisfying subtree bias*.

Definition: The satisfying subtree bias $B_{\text{sat}}(d, i)$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the normalized fraction of all formulae which have a solution in $T_{d-1,\frac{i}{2}}$ that also have a solution in $T_{d,i}$.

The satisfying subtree bias $B_{\text{sat}}(d, i)$ is computed as:

$$B_{\text{sat}}(d, i) = \frac{P_{\text{sat}}(d, i)}{P_{\text{sat}}(d, 2\lfloor\frac{i}{2}\rfloor) + P_{\text{sat}}(d, 2\lfloor\frac{i}{2}\rfloor + 1)} \tag{5}$$

For even i, we say that the branch towards $T_{d,i}$ is biased to the left if $B_{\text{sat}}(d, i) > 0.5$. In case $B_{\text{sat}}(d, i) < 0.5$ we call these branches biased to the right. For odd i the complement holds. The larger the difference between $B_{\text{sat}}(d, i)$ and 0.5, the larger the bias.

To compare the effectiveness of direction heuristics of SAT solvers on random k-SAT formulae, we measured on various depths the average bias towards the left branches. This bias for depth d is computed as the sum of all $B_{\text{sat}}(d, i)$ values with i even, divided by the number of nonzero $B_{\text{sat}}(d - 1, i)$ values.

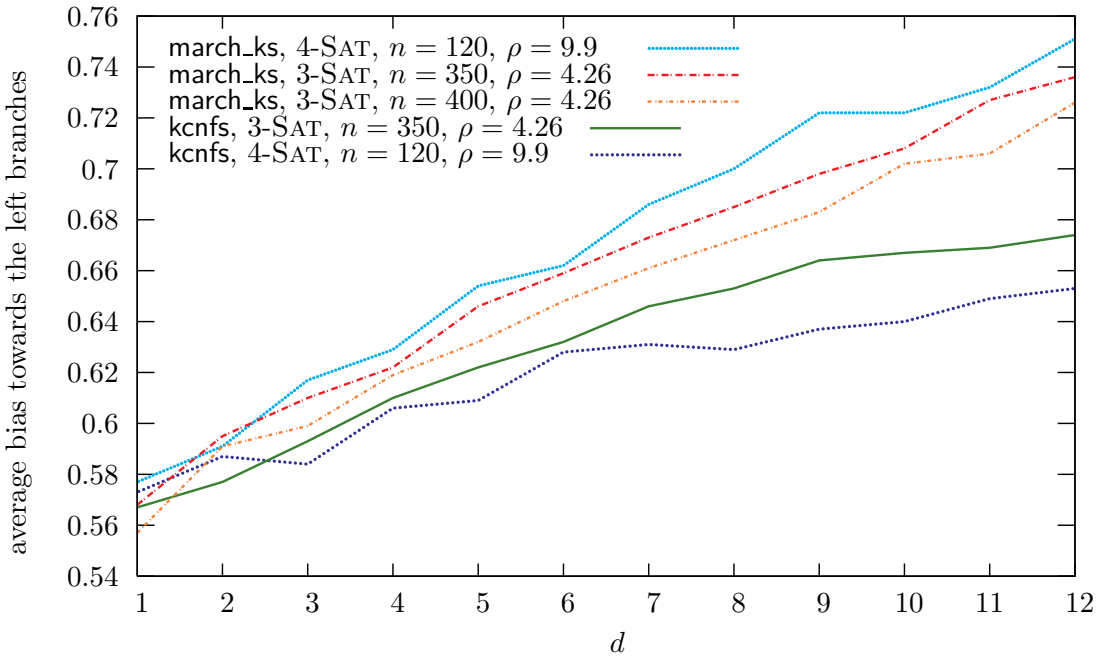

Figure 8. The average (for various d) bias towards the left branches using march_ks and kcnfs on random k-SAT formulae.

Figure 8 shows the average bias and offers some interesting insights: First, we see that, for both march_ks and kcnfs, the average bias towards the left branches is comparable for small d and in particular $d = 1$. This can be explained by the fact that the direction heuristics used in both solvers is similar for formulae with no or few binary clauses. For small d on random k-SAT formulae this is the case. Second, the larger d, the larger the average bias towards the left branches, - for all solver / test set combinations. This supports the intuition that direction heuristics are more effective lower in the search-tree, because the formula is reduced and therefore less complex. Third, the direction heuristics used in march_ks are clearly more effective on the experimented random k-SAT test sets.

Another visualization to compare the effectiveness of direction heuristics are $P_{\text{sat}}/B_{\text{sat}}$ trees. The root of such a tree contains all satisfiable formulae in the given test set. The number in the vertices show the fraction of formulae that still have a solution (the $P_{\text{sat}}(d,i)$ values), while the edges are labeled with the $B_{\text{sat}}(d,i)$ values.

Appendix B shows $P_{\text{sat}}/B_{\text{sat}}$ trees for march_ks, kcnfs, and the Jeroslow-Wang heuristics on random k-SAT formulae. Figure 15 shows such a tree for random 3-SAT with $n = 350$ and $\rho = 4.26$ using march_ks. Again, we see for all d, $P_{\text{sat}}(d,i)$ values are higher if $T(d,i)$ can be reached with less right branches. Also, again we see that the $B_{\text{sat}}(d,i)$ towards the left increases, while d increases. So, the used direction heuristics seem to "guess" the branch containing a solution lower in the search-tree more accurately. Both observations above are also supported by the other $P_{\text{sat}}/B_{\text{sat}}$ trees.

3.3 Finding the first solution

The observed distribution of solutions as well as the bias provide some insight in the effectiveness of the used direction heuristics. Yet, for satisfiable instances we are mainly interested in the usefulness of these heuristics to find the *first* solution quickly.

In order to naturally present the effect of distribution jumping, we first (this subsection) consider the march_ks solver *without* this feature and refer to it as march_ks$^-$. More precisely, march_ks$^-$ visits subtrees in chronological (or depth first) order. We denote chronological order at jump depth d by $\pi_{\text{chro},d} = (0, 1, 2, 3, \ldots, 2^d - 1)$. Since other (jump) orders will be discussed later, all definitions use an arbitrary order at depth d called $\pi_{j,d}$.

Definition: The *first solution probability* $P_{\text{first}}(\ \pi_{j,d},\ i\ )$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the probability that while visiting subtrees at depth d using jump order $\pi_{j,d}$, the solver needs to explore more than i subsequent subtrees to find the first solution (satisfying assignment).

For any $\pi_{j,d}$ holds $P_{\text{first}}(\pi_{j,d},0) = 1$, $P_{\text{first}}(\pi_{j,d},2^d) = 0$, and $P_{\text{first}}(\pi_{j,d},i) \geq P_{\text{first}}(\pi_{j,d},i{+}1)$. A *first solution probability plot* for a given SAT solver and benchmark family shows the probabilities $P_{\text{first}}(\pi_{j,d}, i)$ with $i \in \{0, \ldots, 2^d\}$. Figure 9 shows the probability plot for various random 3-SAT formulae solved using march_ks$^-$. The order of the test sets in the legend represents the order of the probability plots. The shape for the various size and densities have many similarities.

Definition: The *expected tree size fraction* $E_{\text{size}}(\pi_{j,d})$ denotes - for a given SAT solver and a set of satisfiable benchmarks - the expected fraction of the complete search-tree which has to be explored to solve an instance while visiting subtrees at depth d using jump order $\pi_{j,d}$.

$E_{\text{size}}(\ \pi_{j,d}\ )$ is computed as:

$$E_{\text{size}}(\pi_{j,d}) = 2^{-d} \sum_{i \in \{0,\ldots,2^d-1\}} P_{\text{first}}(\pi_{j,d}, i) \tag{6}$$

Notice that $E_{\text{size}}(\pi_{j,d})$ is defined for satisfiable formulae; for unsatisfiable formulae the whole search space has to be explored - hence $E_{\text{size}}(\pi_{j,d})$ is trivially 1. Also, $E_{\text{size}}(\pi_{j,d})$ correlates with the size of the surface below the first solution probability plot with solver and jump depth d. Based on Figure 9, we can state that for march_ks$^-$ the expected tree size fraction $E_{\text{size}}(\pi_{\text{chro},12})$ increases if the density increases. This was expected because formulae with a higher density have on average fewer solutions; it takes longer to find the first solution. For test sets with the same density, $E_{\text{size}}(\pi_{\text{chro},12})$ is slightly smaller for those formulae with more variables. Based on these data, no conclusions can be drawn regarding the computational time: Exploring the search-tree requires much more effort for hard random formulae with increased size.

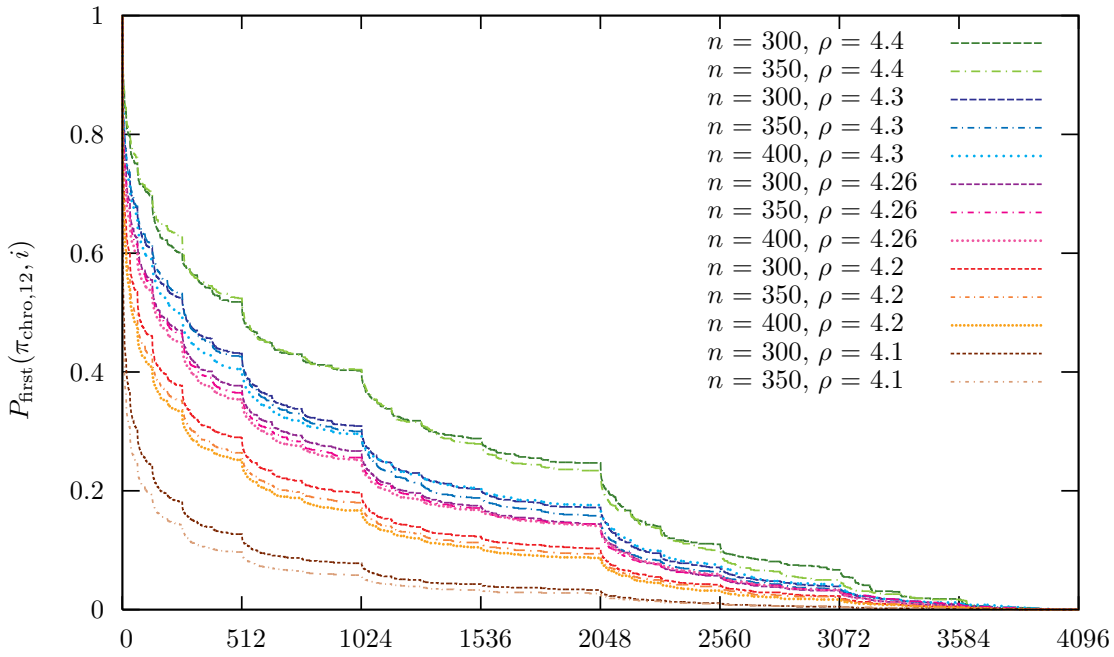

Figure 9. First solution probability plot showing $P_{\text{first}}(\pi_{\text{chro},12}, i)$ on random 3-SAT formulae using march_ks.

4. Distribution jumping

The idea behind *distribution jumping* arises naturally from the observations made in the prior section: While visiting the subtrees in chronological (or depth first) order, the first solution probability plots show some characteristic angels - hinting that the expected tree size fraction (for satisfiable formulae) is far from optimal. So, could we construct an alternative jump order which visits the subtrees in a (near) optimal order? First we will discuss the possibilities to optimize the jump order, followed by some thoughts on the optimal jump depth.

4.1 Optimizing the jump order

This section deals with the question how to construct a jump order with the (near) minimal expected tree size fraction. First, we motivate why we focus on the minimization of the expected tree size fraction. Second, we determine the (close to) minimal $E_{\text{size}}(\pi_{j,12})$ for various random k-SAT formulae using a greedy algorithm. Third, we construct a generalized jump order that can be implemented into a SAT solver.

4.1.1 Theoretical speed-up

Why shall we try to optimize the expected tree size fraction? The reason is closely related to the *theoretical speed-up*. The performance gain realized by distribution jumping can be approximated if the following two restrictions are met:

R_1) the size of the subtrees is about equal;

R_2) the jump depth is much smaller than the size of subtrees.

Due to R_1 the expected computational costs of each subtree is equal and R_2 marginalizes the overhead costs - getting from one subtree to another - of the distribution jumping technique. Using march_ks, the search-trees for hard random k-Sat appear to be quite balanced (satisfying R_1). Given a relatively small jump depth (satisfying R_2) the speed-up could be computed. However, R_1 and R_2 are probably hard to meet for more structured formulae. Given a large set of satisfiable benchmarks, we can compute the theoretical speed-up caused by distribution jumping $\pi_{j,d}$:

$$S(\pi_{j,d}) := \frac{E_{\text{size}}(\pi_{\text{chro},d})}{E_{\text{size}}(\pi_{j,d})} \tag{7}$$

4.1.2 Greedy jump order.

For jump depth d there exist $2^d!$ jump orders in which subtrees can be visited. This makes it hard to compute the *optimal jump order*. We denote the optimal jump order $\pi_{\text{opt},d}(\text{test set})$, the $\pi_{j,d}$ for which $E_{\text{size}}(\pi_{j,d})$ is minimal for the test set using a given Sat solver. Because $\pi_{\text{opt},d}(\text{test set})$ is hard to compute, we focused on an approximation.

Any $\pi_{\text{opt},d}(\text{test set})$ has the property that it is *progressive*: Given a Sat solver and benchmark set, we call $\pi_{j,d}$ a progressive jump order if the probability that the first solution is found at the i-th visited subtree according to $\pi_{j,d}$ $(P_{\text{first}}(\pi_{j,d}, i+1) - P_{\text{first}}(\pi_{j,d}, i))$ is decreasing.

A $\pi_{j,d}$ that is not progressive can be easily improved with respect to a smaller expected tree size fraction: If the probability is not decreasing for a certain i, then swap the order in which the i-th and $i+1$-th subtrees are visited according to that $\pi_{j,d}$ in order to obtain a $\pi_{j,d}{}^*$ which has a smaller expected tree size fraction. Any $\pi_{j,d}$ can be made progressive by applying a finite number of those swaps.

Out of the $2^d!$ possible jump orders, probably only a small number is progressive. We assume that other progressive jump orders of a given test set have an expected tree size fraction close to the minimal one. Therefore, we developed a greedy algorithm which computes for a given Sat solver and test set a progressive jump order called $\pi_{\text{greedy},d}(\text{test set})$. Consider the following algorithm:

1. Start with an empty jump order and a set of satisfiable benchmarks.
2. Select the subtree $T_{d,i}$ in which most formulae have at least one solution. In case of a tie-break we select the one with the smallest i. The selected subtree is the next to be visited in the greedy jump order.
3. All formulae that have at least one solution in the selected subtree (of step 2) are removed from the set.

4. Repeat the steps 2 and 3 until all formulae of the set are removed.

We computed the greedy jump order for various random k-SAT test sets (denoted by $R_{k,n,\rho}$) using march_ks. The results are shown in Table 1. Columns two to five show the order in which $T_{12,i}$'s should be visited according to the greedy orders based on different hard random k-SAT formulae. To clearly present also the $\#RB(T_{12,i})$ values within the order, only the indices i are shown. These are printed in binary representation with bold 1's (right branches). E.g., in Table 1 $T_{12,1024}$ is shown as 0**1**0000000000.

Table 1. $\pi_{\text{greedy},12}(k, n, \rho)$ computed for random 3-SAT ($n \in \{300, 305, 400\}$, $\rho = 4.26$) and 4-SAT ($n = 120$, $\rho = 9.9$). The index of the greedy jump orders is shown in binary representation.

$\pi_{\text{chro},12}$	$\pi_{\text{greedy},12}(R_{3,300,4.26})$	$\pi_{\text{greedy},12}(R_{3,350,4.26})$	$\pi_{\text{greedy},12}(R_{3,400,4.26})$	$\pi_{\text{greedy},12}(R_{4,120,9.9})$
0	000000000000	000000000000	000000000000	000000000000
1	**1**00000000000	**1**00000000000	**1**00000000000	00**1**000000000
2	0**1**0000000000	0**1**0000000000	00**1**000000000	0**1**0000000000
3	00**1**000000000	000**1**00000000	000**1**00000000	000**1**00000000
4	0000**1**0000000	00**1**000000000	0**1**0000000000	**1**00000000000
5	000000**1**00000	000000**1**00000	0000**1**0000000	0**1**0010000000
6	000**1**00000000	0000**1**0000000	000000**1**0000	000000**1**00000
7	0000**1**000000	**1**01000000000	0000**1**000000	0000**1**0000000
8	**11**0000000000	000000001000	**1**00001000000	000000**1**0000
9	**1**00001000000	**1**00**1**00000000	**1**01000000000	**11**0000000000
10	**1**0001**0**000000	000000**1**0000	**1**00001000000	**1**00**1**00000000
11	000000**1**0000	**11**0000000000	000000**1**00000	**1**01000000000
12	0**1**0**1**00000000	0**11**000000000	**11**0000000000	**1**0001**0**000000
13	000000001000	000000000**1**0	000000001000	000000000**1**00
14	**1**00001000000	**1**0001**0**000000	0**1**001**0**000000	0**1**0**1**00000000
15	00**11**00000000	0000**1**000000	00**11**00000000	0**1**0001000000
16	**1**01000000000	**1**00000**1**0000	00**1**01**0**000000	000**11**0000000
17	**1**00**1**00000000	**1**0000**1**000000	**1**00**1**00000000	00**11**00000000
18	000**11**0000000	0**1**0001000000	0**11**000000000	0000**1**000000
19	000000000001	00**1**000**1**0000	000**1**0**1**00000	0**1**0000**1**00000
...	...	...	...	...

An interesting trait that can be observed from the $\pi_{\text{greedy},12}(R_{k,n,\rho})$ jump orders is that they spoil the somewhat perfect pictures presented in Section 3. Recall that in the solution distribution histograms all subtrees which can be reached in a single right branch show higher peaks than all subtrees reachable with two (or more) right branches. Here, we observe a similar tendency. Yet, $T_{12,1}$, $T_{12,2}$, $T_{12,4}$, $T_{12,8}$ seem far less important than for instance $T_{12,3072}$ (**11**0000000000). So, based on the greedy jump orders we can conclude that for optimal performance, $T_{d,i}$ should not solely be visited in increasing number of right branches. In other words, the $P_{\text{sat}}(d, i)$ values are not a perfect tool for constructing the ideal jump order.

4.1.3 Generalized jump order.

Since we only computed $\pi_{\text{greedy},12}$ (only for jump depth 12) and the order is slightly different for the different experimented data-sets, a more generalized permutation is required for the actual implementation. Although the greedy jump orders are our best approximation of the optimal jump order, we failed to convert them to a generalized jump order. Instead, we created a generalized jump order based on two prior observations: First, subtrees reachable in less right branches have a higher probability of containing a solution (see section 3.1). Second, among subtrees which are reachable in the same number of right branches, those with right branches near the root have a higher probability of containing a solution (see section 3.2).

Based on these observations, we propose the following jump order: First, visit $T_{d,0}$, followed by all $T_{d,i}$'s that can be reached in only one right branch. Continue with all T_i's that can be reached in two right branches, etc. All $T_{d,i}$'s that can be reached in the same number of right branches are visited in decreasing order of i. We refer to this permutation as $\pi_{\text{left},d}$.

Table 2 shows the visiting order of the subtrees with $\pi_{\text{chro},4}$ and $\pi_{\text{left},4}$. There are only a few similarities between $\pi_{\text{chro},d}$ and $\pi_{\text{left},d}$: The first and the last subtree ($T_{d,0}$ and $T_{d,2^d-1}$) have the same position in both jump orders. Also, both $\pi_{\text{chro},d}$ and $\pi_{\text{left},d}$ are symmetric: If subtree $T_{d,i}$ has position j in the order than subtree $T_{d,2^d-1-j}$ has position 2^d-1-j in that order.

Notice that $\pi_{\text{chro},d}$ and $\pi_{\text{left},d}$ are in some sense complementary: $\pi_{\text{chro},d}$ starts by visiting subtrees that have right branches near the leaf nodes, while $\pi_{\text{left},d}$ starts by visiting subtrees having right branches near the root. Since the next subtree selected by $\pi_{\text{chro},d}$ is the nearest subtree, "jumping" is very cheap. A motivation for using $\pi_{\text{left},d}$ is that direction heuristics are more likely to fail near the root of the search tree because at these nodes it is harder to predict the "good" direction. Therefore, subtrees with few right branches of which most are near the root of the search-tree have a higher probability of containing a solution.

Table 2. Jump orders in which subtrees can be visited with jump depth 4.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi_{\text{chro},4}$	$T_{4,0}$	$T_{4,1}$	$T_{4,2}$	$T_{4,3}$	$T_{4,4}$	$T_{4,5}$	$T_{4,6}$	$T_{4,7}$	$T_{4,8}$	$T_{4,9}$	$T_{4,10}$	$T_{4,11}$	$T_{4,12}$	$T_{4,13}$	$T_{4,14}$	$T_{4,15}$
$\pi_{\text{left},4}$	$T_{4,0}$	$T_{4,8}$	$T_{4,4}$	$T_{4,2}$	$T_{4,1}$	$T_{4,12}$	$T_{4,10}$	$T_{4,9}$	$T_{4,6}$	$T_{4,5}$	$T_{4,3}$	$T_{4,14}$	$T_{4,13}$	$T_{4,11}$	$T_{4,7}$	$T_{4,15}$

Using the data from the experiments on random k-Sat formulae to compute the distribution of solutions, we obtained[2] the $P_{\text{first}}(\pi_{\text{left},d}, i)$ and $E_{\text{size}}(\pi_{\text{left},d}, i)$ values.

Figure 10 shows the first solution probability plots based on this data for random 3-Sat test sets of different sizes and densities using march_ks$^-$ and $\pi_{\text{left},12}$. The lines decrease much more rapidly compared to those of Figure 9, also indicating that the expected tree size fraction is much smaller using this jump order. The sequence (from top to bottom)

2. The data was gathered using march_ks$^-$ while visiting the subtree in chronological order. Because march_ks$^-$ uses adaptive heuristics, actual jumping according to $\pi_{\text{left},d}$ might result in a slightly different search-tree and thus may influence the $P_{\text{first}}(\pi_{\text{left},d}, i)$ and $E_{\text{size}}(\pi_{\text{left},})$ values. Further discussion in Section 5.1

of these lines is about the same for both figures: The larger ρ, the higher the line in the sequence. However, in Figure 10 a larger n does not always result in a lower line.

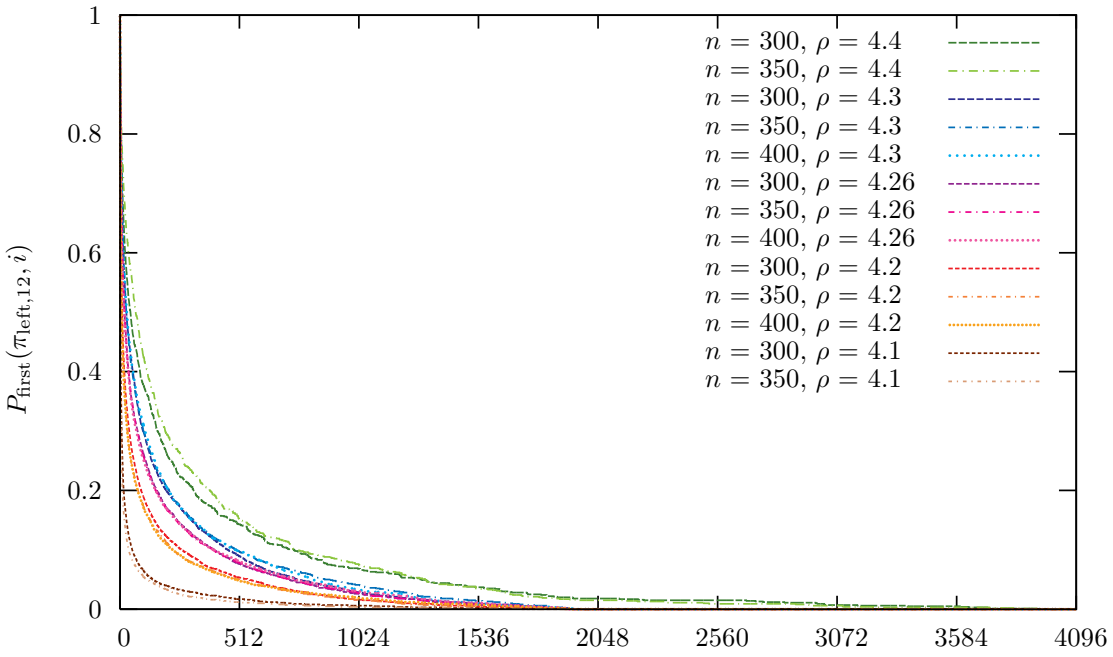

Figure 10. First solution probability plot showing $P_{\text{first}}(\pi_{\text{left},12}, i)$ on random 3-SAT formulae using march_ks$^-$. Please compare to Figure 9.

Table 3 summarizes the results of these tests on all the experimented data. Shown is $\mu_{\text{sat}}(d)$, $E_{\text{size}}(\pi_{\text{chro},d})$, $E_{\text{size}}(\pi_{\text{left},d})$ and the theoretical speed-up which is computed using these values. The speed-up using march_ks$^-$ with $\pi_{\text{left},d}$ is about a factor 4. On formulae below the phase transition density, the speed-up is even greater, while above the phase transition density it is less. Also, formulae with a higher $\mu_{\text{sat}}(d)$ value have a greater speed-up - although the correlation is much less clear. The speed-up realized by kcnfs with $\pi_{\text{left},d}$ is smaller than by march_ks$^-$. A possible explanation is that the direction heuristics used in kcnfs result in a smaller bias towards the left. Using march_ks$^-$ with $\pi_{\text{left},d}$, the speed-up on the 4-SAT test set is the smallest. This may be caused by the relatively small $\mu_{\text{sat}}(12)$ value.

4.2 Optimizing the jump depth

Under the assumption that $\pi_{\text{left},d}$ is an effective jump order, we now face the question: When to jump? Or more precise: What is the optimal jump depth? For different values of jump depth d, the leaf nodes of the search-tree are visited in a different order - in contrast to jumping using $\pi_{\text{chro},d}$. Therefore, which d is optimal? First, we will try to answer this question from a theoretical viewpoint followed by some practical difficulties.

Table 3. Expected tree size fraction and speed-up for random k-SAT formulae.

solver	family	n	ρ	$\mu_{\text{sat}}(12)$	$E_{\text{size}}(\pi_{\text{chro},12})$	$E_{\text{size}}(\pi_{\text{left},12})$	$S(\pi_{\text{left},12})$
march_ks$^-$	3-SAT	300	4.1	95.803	0.04965	0.00993	5.00
march_ks$^-$	3-SAT	300	4.2	29.575	0.11863	0.02727	4.35
march_ks$^-$	3-SAT	300	4.26	15.775	0.15673	0.03804	4.12
march_ks$^-$	3-SAT	300	4.3	10.843	0.18026	0.04365	4.13
march_ks$^-$	3-SAT	300	4.4	5.379	0.23408	0.05456	4.29
march_ks$^-$	3-SAT	350	4.1	147.933	0.03934	0.00811	4.85
march_ks$^-$	3-SAT	350	4.2	40.199	0.10919	0.02510	4.35
march_ks$^-$	3-SAT	350	4.26	19.534	0.15406	0.03822	4.03
kcnfs	3-SAT	350	4.26	18.021	0.16990	0.06113	2.78
march_ks$^-$	3-SAT	350	4.3	12.925	0.17409	0.04569	3.81
march_ks$^-$	3-SAT	350	4.4	5.871	0.22971	0.06399	3.59
march_ks$^-$	3-SAT	400	4.2	52.906	0.10237	0.02572	3.98
march_ks$^-$	3-SAT	400	4.26	24.461	0.15047	0.04056	3.71
march_ks$^-$	3-SAT	400	4.3	15.281	0.17682	0.04715	3.75
march_ks$^-$	4-SAT	120	9.9	4.817	0.23514	0.07864	2.99

Notice that

$$\begin{aligned} P_{\text{first}}(\pi_{\text{chro},d}, 2i) &= P_{\text{first}}(\pi_{\text{chro},d-1}, i) \quad \text{and} \\ P_{\text{first}}(\pi_{\text{chro},d-1}, i+1) \leq P_{\text{first}}(\pi_{\text{chro},d}, 2i+1) &\leq P_{\text{first}}(\pi_{\text{chro},d-1}, i) \end{aligned}$$

So,

$$\begin{aligned} \frac{2 \cdot E_{\text{size}}(\pi_{\text{chro},d-1}) - 2^{-(d-1)} \cdot P_{\text{first}}(\pi_{\text{chro},d-1}, 0)}{2} \leq E_{\text{size}}(\pi_{\text{chro},d}) &\leq \frac{2 \cdot E_{\text{size}}(\pi_{\text{chro},d-1})}{2} \\ E_{\text{size}}(\pi_{\text{chro},d-1}) - 2^{-d} \leq E_{\text{size}}(\pi_{\text{chro},d}) &\leq E_{\text{size}}(\pi_{\text{chro},d-1}) \end{aligned}$$

In other words, $E_{\text{size}}(\pi_{\text{chro},d})$ is decreasing for increasing d and converges fast. These properties are independent of the used SAT solver[3.] and the used benchmark family. Figure 11 shows how $P_{\text{first}}(\pi_{\text{chro},d}, \text{i})$ (and thus $E_{\text{size}}(\pi_{\text{chro},d})$ evolves for increasing d based on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$ using march_ks. For these formulae and solver, $E_{\text{size}}(\pi_{\text{chro},d})$ converges to 0.150.

Both properties - decrease and convergence - may not hold for $E_{\text{size}}(\pi_{\text{left},d})$ for some solver / benchmark family combination. The influence of the jump depth on $E_{\text{size}}(\pi_{\text{left},d})$ depends heavily on the direction heuristics used in a solver. Using march_ks on random 3-SAT formulae, we observed that $E_{\text{size}}(\pi_{\text{left},d})$ is decreasing for increasing d. However (fortunately), $E_{\text{size}}(\pi_{\text{left},d})$ is still visibly decreasing for increasing d during our experiments (using $d \in \{0, \ldots, 12\}$). This observation is visualized in Figure 12. Consequently the theoretical speed-up of using march_ks with distribution jumping based on $\pi_{\text{left},d}$ instead of march_ks$^-$ improves while increasing d.

3. Assuming that subtrees are visited in chronological order and no restarts are performed

This is also the main conclusion of Table 4 - showing the influence of the jump depth on $E_{\text{size}}(\pi_{\text{chro},d})$, $E_{\text{size}}(\pi_{\text{left},d})$ and the theoretical speed-up. Notice also that $E_{\text{size}}(\pi_{\text{chro},0}) = E_{\text{size}}(\pi_{\text{left},0})$ and $E_{\text{size}}(\pi_{\text{chro},1}) = E_{\text{size}}(\pi_{\text{left},1})$. While using march_ks on random 3-SAT formulae with $n = 400$ and $\rho = 4.26$, $E_{\text{size}}(\pi_{\text{left},d}) < E_{\text{size}}(\pi_{\text{chro},d})$ during the experiments (i.e. for $d \in \{2, \ldots, 12\}$).

Table 4. The influence of the jump depth denoted as d on $E_{\text{size}}(\pi_{\text{chro},d})$, $E_{\text{size}}(\pi_{\text{left},d})$ and the expected speed-up $S(\pi_{\text{left},d})$ on random 3-SAT formulae with 400 variables and density 4.26.

d	$E_{\text{size}}(\pi_{\text{chro},d})$	$E_{\text{size}}(\pi_{\text{left},d})$	$S(\pi_{\text{left},d})$
0	1.000	1.000	1.000
1	0.571	0.571	1.000
2	0.357	0.347	1.029
3	0.252	0.231	1.089
4	0.200	0.165	1.215
5	0.175	0.128	1.373
6	0.163	0.102	1.589
7	0.156	0.085	1.837
8	0.153	0.072	2.125
9	0.152	0.062	2.440
10	0.151	0.054	2.790
11	0.151	0.048	3.173
12	0.151	0.041	3.713

So, based on the theoretical speed-up, the optimal value for the jump depth is probably $d = \infty$. In other words, theoretically jumping between the leaf nodes of the search-tree results in the largest performance gain.

However, in practice, two kinds of overhead exist: First, the cost of jumping. While solving, march_ks spends most time to determine for each node the decision variable and to detect forced variables. Compared to these calculations, the cost of jumping (backtracking to the nearest parent node and descending in the tree) for one subtree to another is relatively cheap. Therefore, the overhead of jumping is marginal.

On the other hand, because the cost of obtaining the decision variable and forced literals is huge, one would like to remember this information. Therefore, these data should be stored for nodes that will be visited more than once (i.e. those nodes at depth $\leq d$). That will cost a lot of memory and time if d is large.

To reduce the cost of overhead, we therefore decided to use a jump depth in such way that only a fraction of the nodes need to be stored: In the first phase of solving (before the first jump) the average depth of the search-tree is estimated: The jump depth is set to be 7 levels above this average. Using this heuristic only about 1 in 100 nodes is stored.

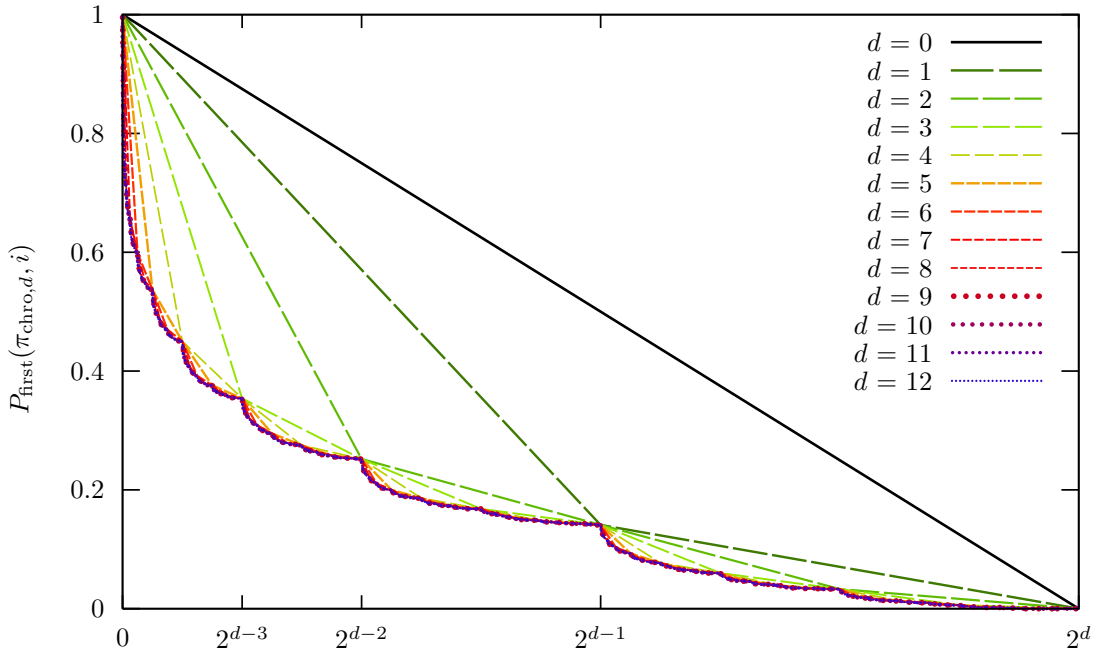

Figure 11. Influence of the jump depth d on the first solution probability $P_{\text{first}}(\pi_{\text{chro},d}, \text{i})$ using march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$.

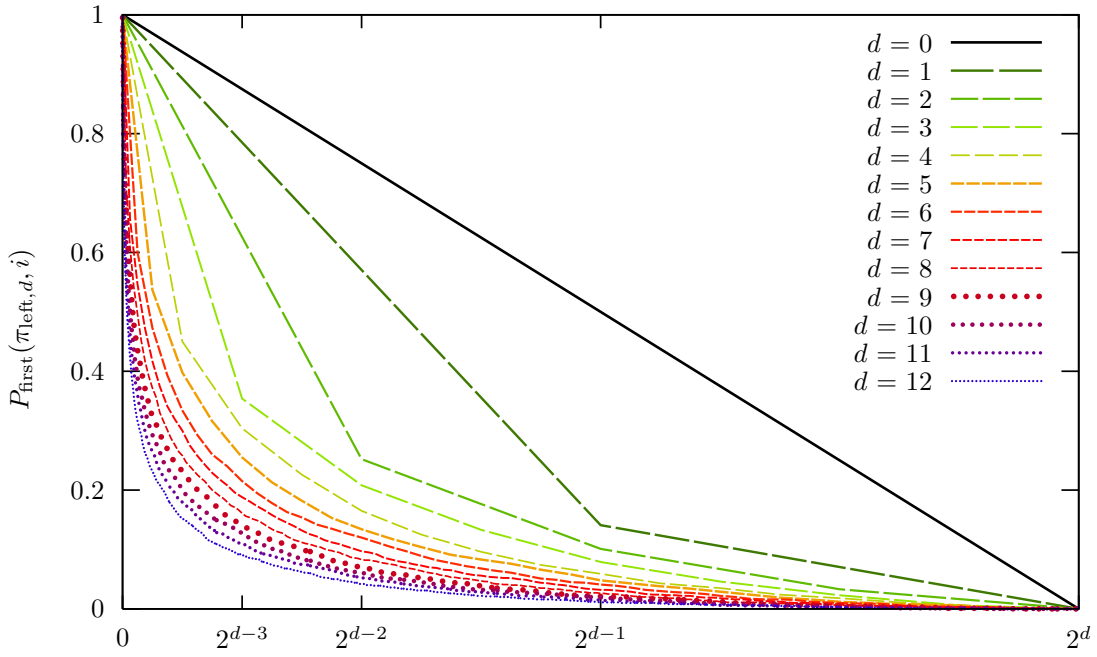

Figure 12. Influence of the jump depth d on the first solution probability $P_{\text{first}}(\pi_{\text{left},d}, \text{i})$ using march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$.

5. Results

Distribution jumping is one of the main new features in the march SAT solver resulting in version march_ks which participated at the SAT 2007 competition: The technique is implemented as discussed above. First, using the dynamic jump depth heuristic d is computed. Second, the solver jumps using $\pi_{\text{left},d}$.

5.1 Random 3-SAT

To examine the actual speed-up resulting from distribution jumping, we run both march_ks$^-$ and march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$. The results are shown in Figure 13. On most of these instances, the adaptive jump depth for march_ks was set to 15. On satisfiable formulae, the performance of march_ks is much better. Although march_ks$^-$ is faster on several instances, on average it takes about 3.5 times more effort to compute the first solution. Regarding the extremes: The largest performance gain on a single instance is a factor 744 (from 81.94 to 0.11 seconds), while the largest performance loss is a factor 37 (from 35.52 to 0.95 seconds).

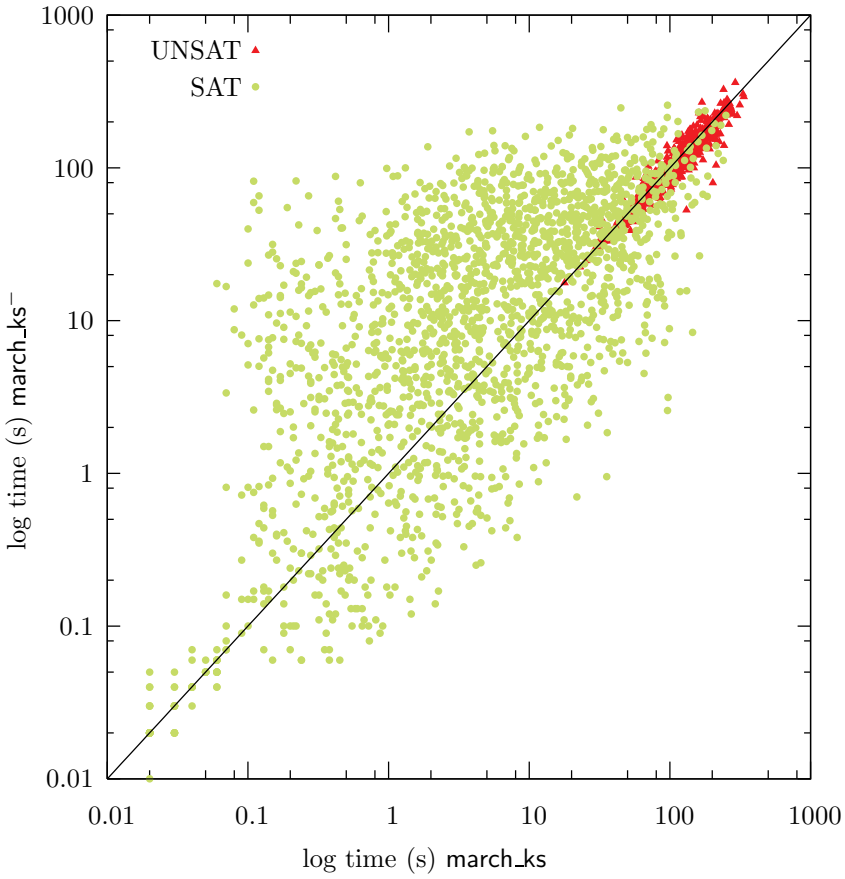

Figure 13. Performance comparison (in seconds and logscale) between march_ks$^-$ and march_ks on 10.000 random 3-SAT formulae with $n = 400$ and $\rho = 4.26$

The performance of march_ks$^-$ and march_ks on unsatisfiable instances is comparable. Due to the use of some adaptive heuristics in these SAT solvers, they may not explore exactly the same search-tree. It appears that this could result in a computational cost difference of about 10 % in favor of either march_ks$^-$ or march_ks. The overhead costs in march_ks are only marginally.

Table 5. Number of solved instances within a timeout of 600 seconds

		march_ks$^-$		march_ks		R$^+$AdaptNovelty$^+$	
n	ρ	#SAT	#UNKNOWN	#SAT	#UNKNOWN	#SAT	#UNKNOWN
600	4.0	86	14	100	0	100	0
700	4.0	75	25	100	0	100	0
600	4.1	73	27	100	0	100	0
700	4.1	15	85	90	10	100	0

Large random 3-SAT formulae with density 4.26 appeared still hard for our current implementation. So, for our experiments we generated random 3-SAT formulae for four different sizes - 600 and 700 variables both at density 4.0 and 4.1 - to test the improvement of march_ks. We compared the performance of our solvers with R$^+$AdaptNovelty$^+$ [1] - an incomplete (local search) solver which appeared the strongest on these kind of formulae during the SAT 2005 competition [8]. In general, incomplete SAT solvers are dominant on satisfiable random formulae, but they fail to compete (with complete SAT solvers) on most structured satisfiable instances.

Table 5 shows the results of this experiment. The progress from march_ks$^-$ to march_ks is clear. However, R$^+$AdaptNovelty$^+$ is still more successful on these instances.

5.2 SAT Competition 2007

March_ks participated at the SAT Competition 2007. It won the satisfiable crafted category and several other awards. Especially in the former category, distribution jumping contributed to the success of the solver. Moreover, without this feature it would not have won.

Four hard crafted benchmarks `connm-ue-csp-sat-n800-d-0.02` (connamacher), `QG7a-gensys-ukn005`, `QG7a-gensys-brn004`, and `QG7a-gensys-brn100` (quasigroup) are not solved by march_ks without distribution jumping in 12 hours. However, the competition version which includes this feature solves these instances in 306, 422, 585, 3858 seconds, respectively. The dynamic jump depth heuristic selects $d = 12$ for the connamacher instance and $d = 26$ for the quasigroup instances. The solutions were found with only one or two right branches (above the jump depth). Various other solvers were able to solve the quasigroup instances within the 1200 seconds timeout. However, the connamacher instance was only solved by march_ks and satzilla.

Two other hard crafted benchmarks, `ezfact64-3` and `ezfact64-6`, both factorization problems, where solved by march_ks in the second round in 3370 and 2963 seconds, respectively. Only minisat was also able to solve `ezfact64-3`, while `ezfact64-6` was exclusively solved by march_ks. Without distribution jumping, solving these instances requires about twice the computational time. Since the timeout in the second round was 5000 seconds,

march_ks$^-$ would not have solved them. Therefore, if march_ks$^-$ would have participated in the Sat 2007 competition - instead of march_ks - it would not have won the crafted satisfiable category, because it would not have solved these six benchmarks.

6. Conclusions

We observed that both march_ks and kcnfs bias on random k-Sat formulae - due to the used direction heuristics - the distribution of solutions towards the left branches. We introduced a measurement called the satisfying subtree bias B_{sat} to quantify the bias. Using B_{sat} the bias of the direction heuristics used in different solvers can be compared.

To capitalize on these observations we developed the new jumping strategy *distribution jumping*. While alternative jump strategies examine a random new part of the search-tree, our proposed method jumps towards a subtree that has a high probability of containing a solution.

Distribution jumping has been implemented in march_ks. With this new feature, the Sat solver can solve significantly more satisfiable random k-Sat formulae without hurting its performance on unsatisfiable instances. Despite the progress, march_ks is still outperformed by incomplete solvers on these benchmarks.

Yet, also the performance of march_ks is improved on satisfiable structured formulae. Apparently, distribution jumping is applicable outside the experimented domain. Thanks to this new technique, march_ks won the satisfiable crafted category of the Sat 2007 Competition.

The usefulness of distribution jumping could be even further increased by better direction heuristics: The more biased the distribution of solutions, the larger the expected speed-up. Also, the results of the greedy jump orders suggest that there is an opportunity to improve the generalized jump order.

Acknowledgments

The authors would like to thank Stephan van Keulen for computing and obtaining most data used in this paper.

References

[1] Anbulagan, Duc Nghia Pham, John K. Slaney, and Abdul Sattar. Old resolution meets modern SLS. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 354–359. AAAI Press / The MIT Press, 2005.

[2] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, **5**(7):394–397, 1962.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, **2919** of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[4] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In Gert Smolka, editor, *CP*, **1330** of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1997.

[5] Marijn J.H. Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation*, **2**:47–59, 2006.

[6] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Ann. Math. Artif. Intell.*, **1**:167–187, 1990.

[7] Oliver Kullmann. Investigating the behaviour of a SAT solver on random formulas. Technical Report CSR 23-2002, University of Wales Swansea, Computer Science Report Series (http://www-compsci.swan.ac.uk/reports/2002.html), October 2002. 119 pages.

[8] Daniel LeBerre and Laurent Simon. Preface special volume on the SAT 2005 competitions and evaluations, 2006. Journal on Satisfiability, Boolean Modeling and Computation **2**.

[9] Dimos Mpekas, Michiel van Vlaardingen, and Siert Wieringa. The first steps to a hybrid SAT solver, 2007. MSc report, SAT@Delft.

[10] Hantao Zhang. A complete random jump strategy with guiding paths. In Armin Biere and Carla P. Gomes, editors, *SAT*, **4121** of *Lecture Notes in Computer Science*, pages 96–101. Springer, 2006.

Appendix A. Solution distribution histograms

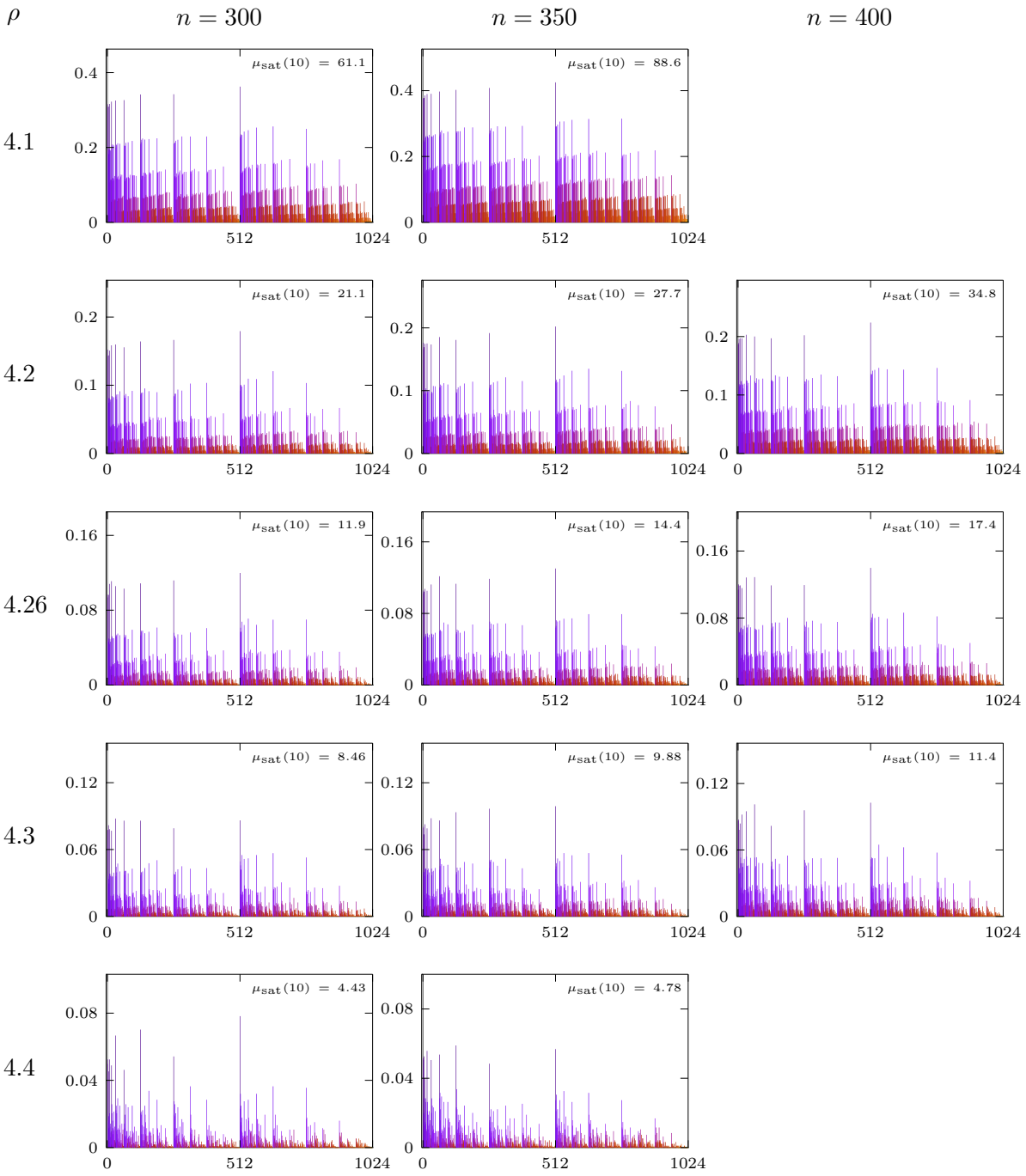

Figure 14. Solution distribution plots showing $P_{\text{sat}}(10, i)$ (y-axis) of random 3-SAT formulae with $n \in \{300, 350, 400\}$ and $\rho \in \{4.1, 4.2, 4.26, 4.3, 4.4\}$ using march_ks.

Appendix B. P_{sat}/ B_{sat} trees

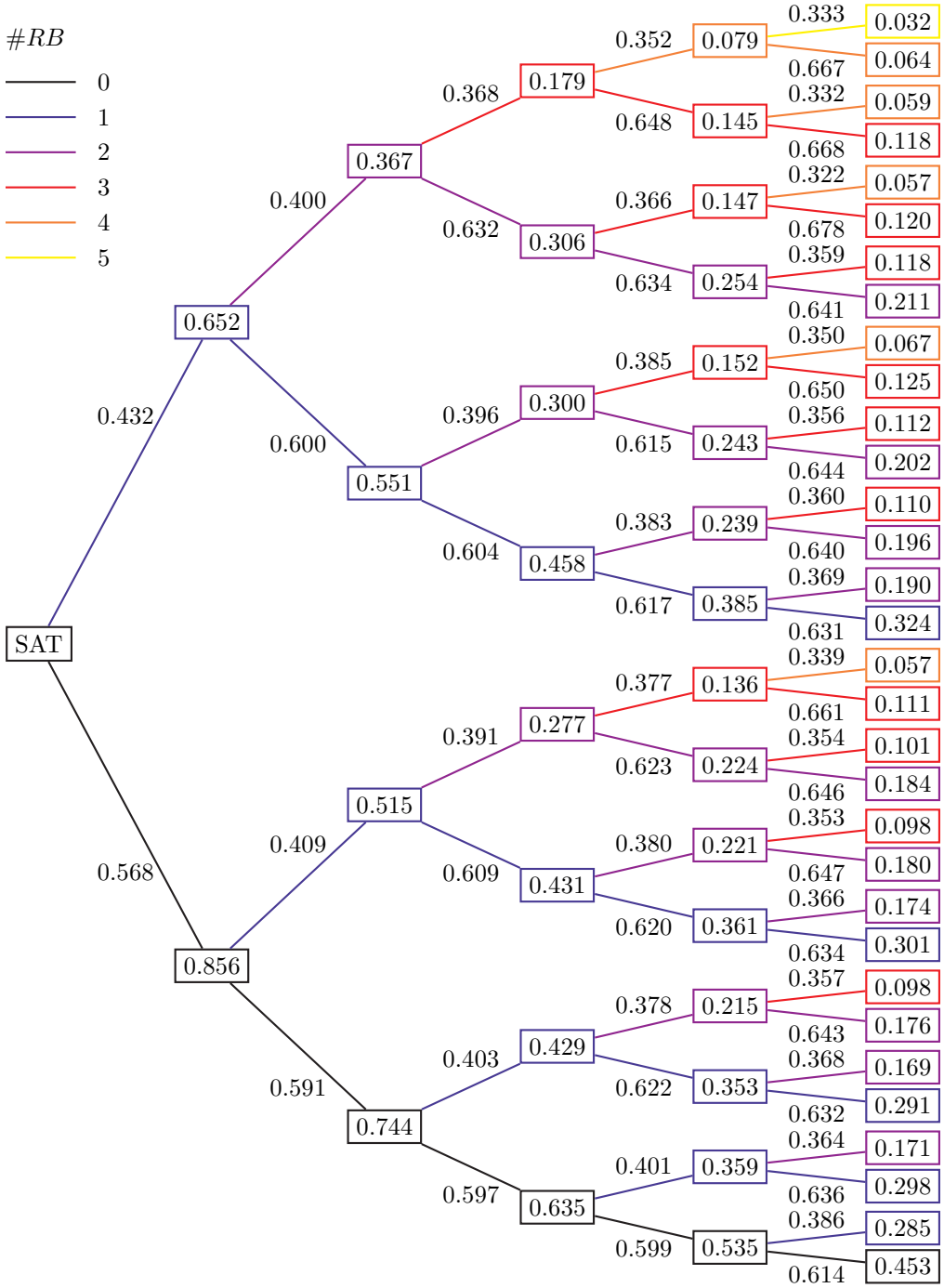

Figure 15. $P_{\text{sat}}/B_{\text{sat}}$ tree of march_ks^{-} running on random 3-SAT with $n = 350$ and $\rho = 4.26$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

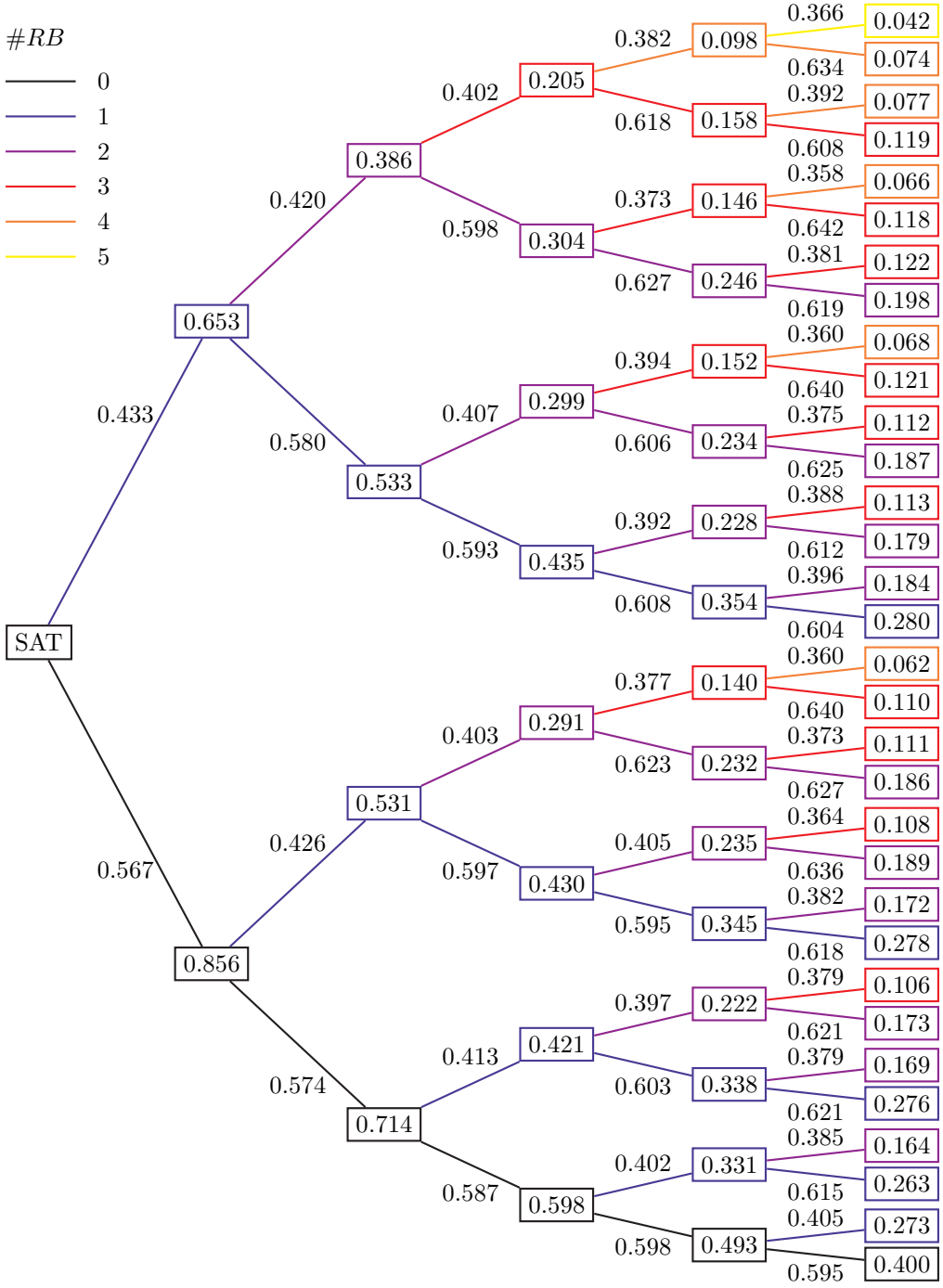

Figure 16. $P_{\text{sat}}/B_{\text{sat}}$ tree of kcnfs running on random 3-SAT with $n = 350$ and $\rho = 4.26$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

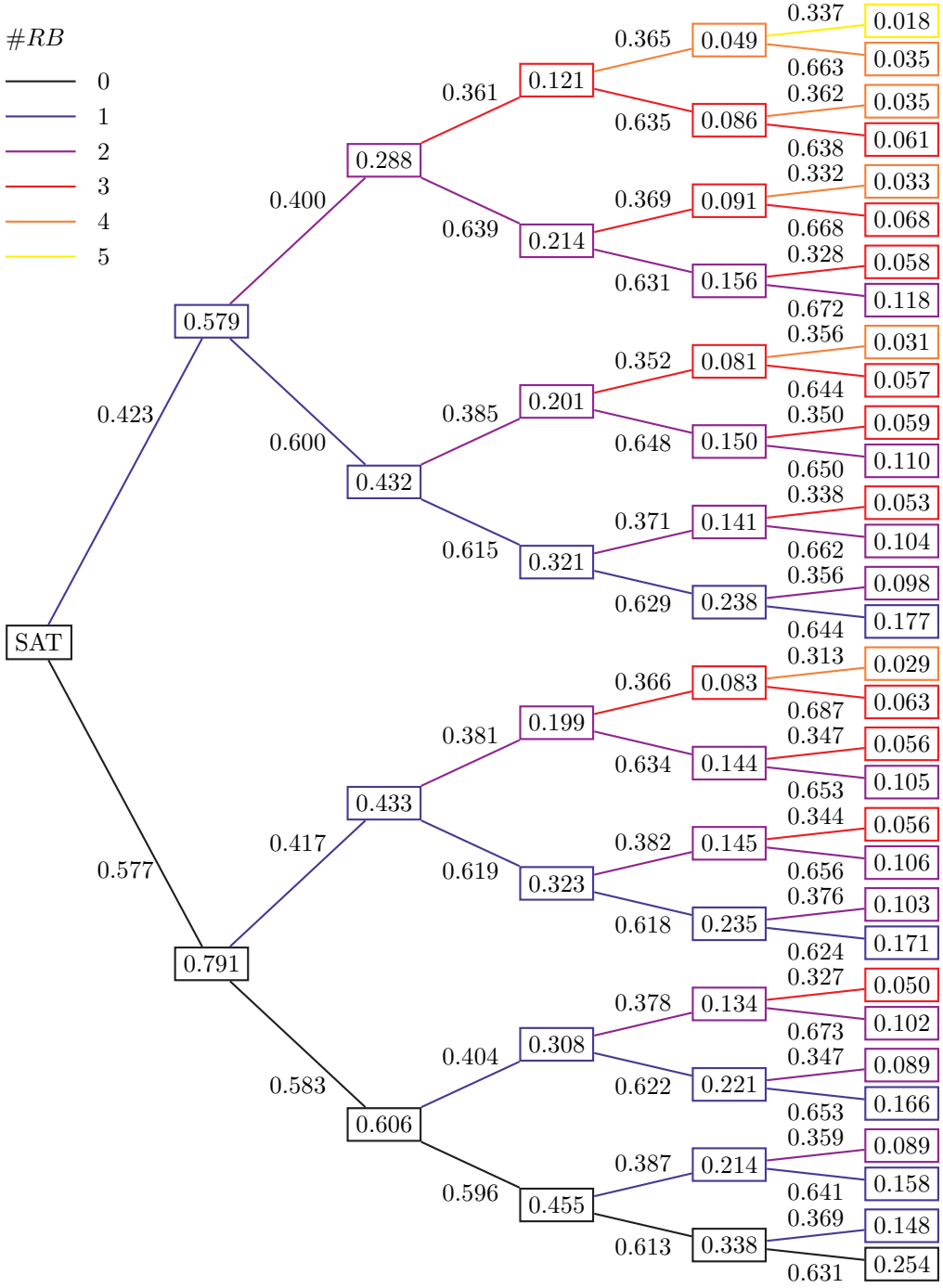

Figure 17. $P_{\text{sat}}/B_{\text{sat}}$ tree of march_ks$^-$ running on random 4-SAT with $n = 120$ and $\rho = 9.9$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

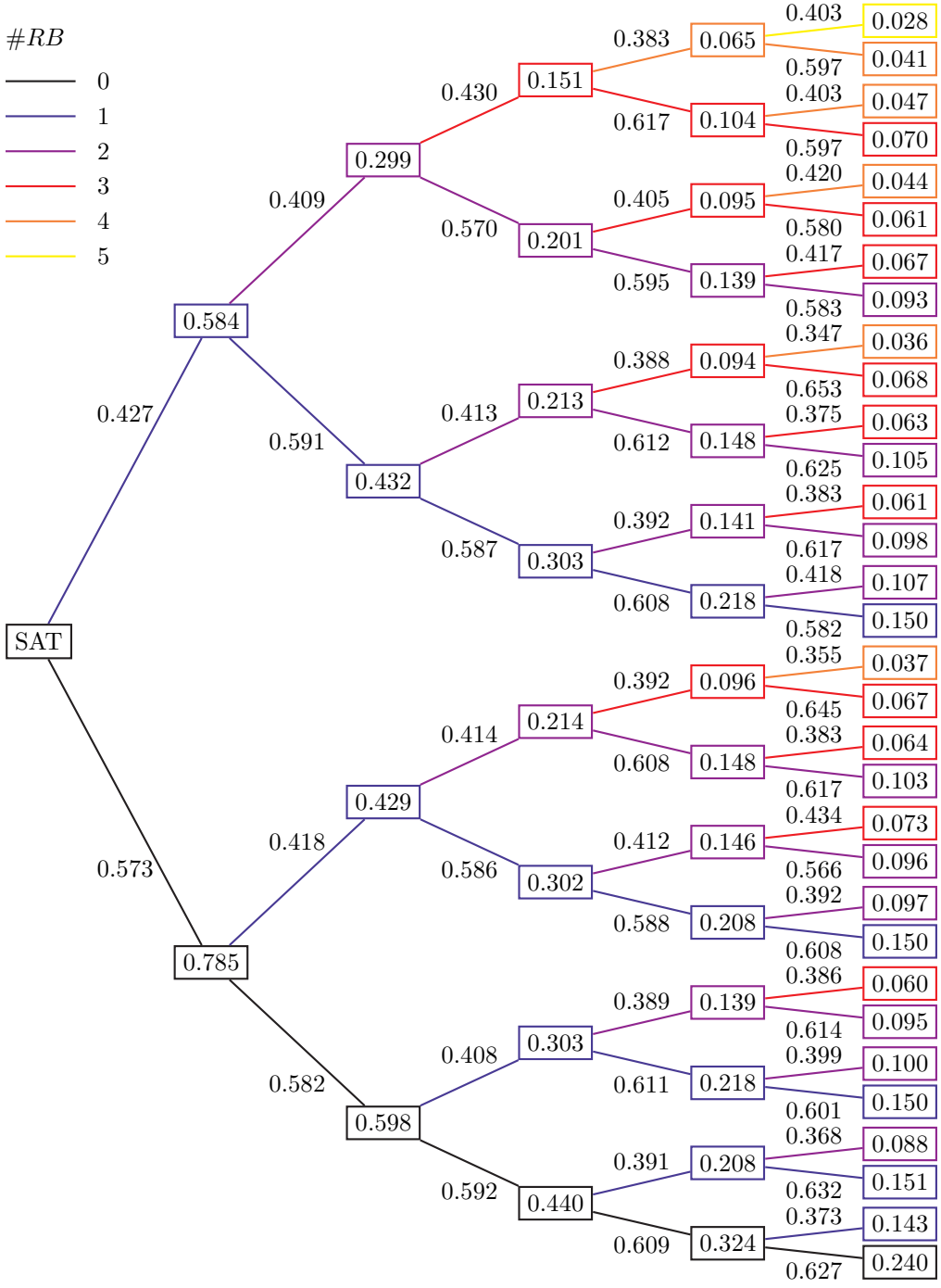

Figure 18. $P_{\text{sat}}/B_{\text{sat}}$ tree of kcnfs running on random 4-SAT with $n = 120$ and $\rho = 9.9$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

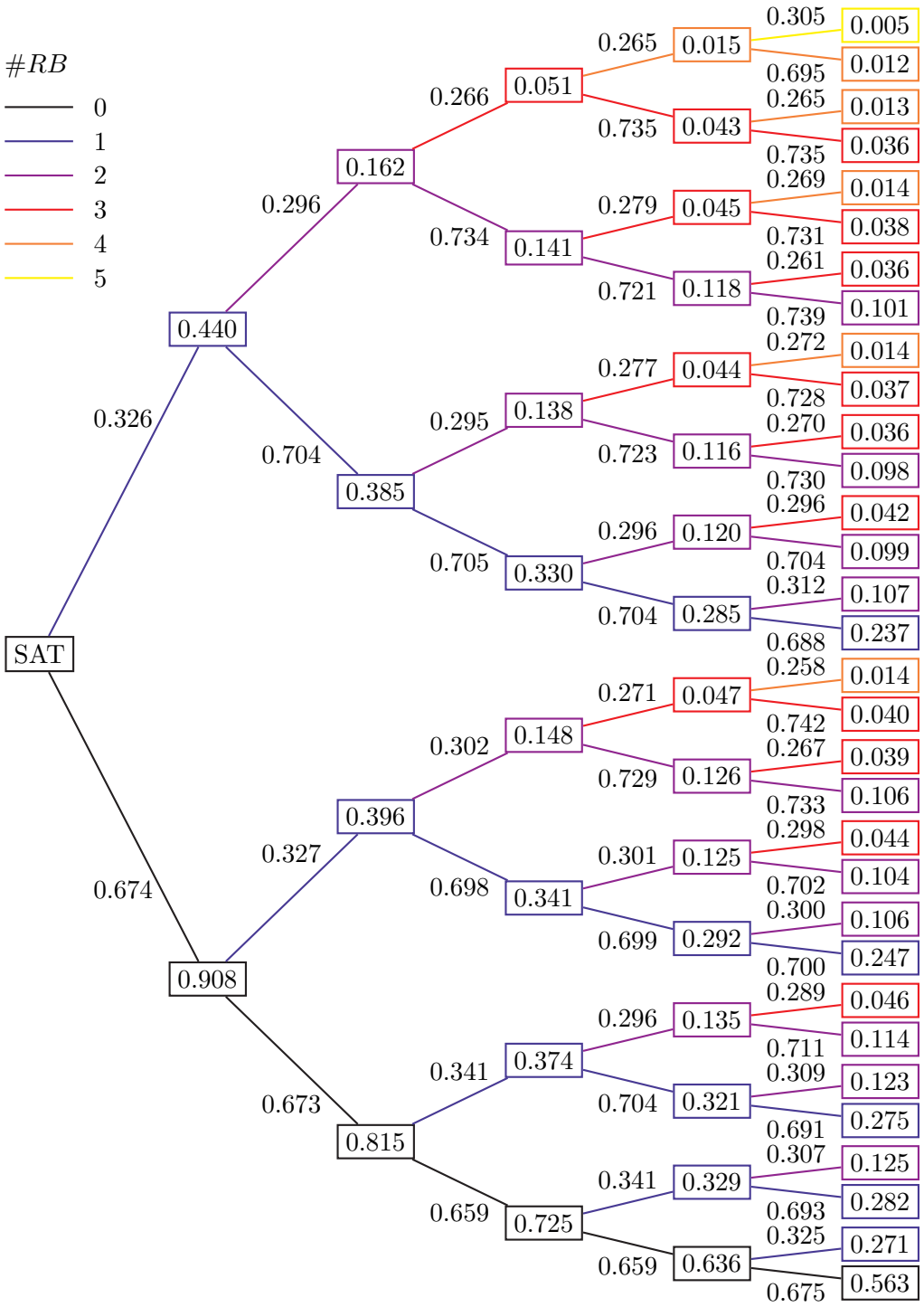

Figure 19. $P_{\text{sat}}/B_{\text{sat}}$ tree of Jeroslow-Wang one-sided heuristic on random 3-SAT with $n = 120$, $\rho = 4.26$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.

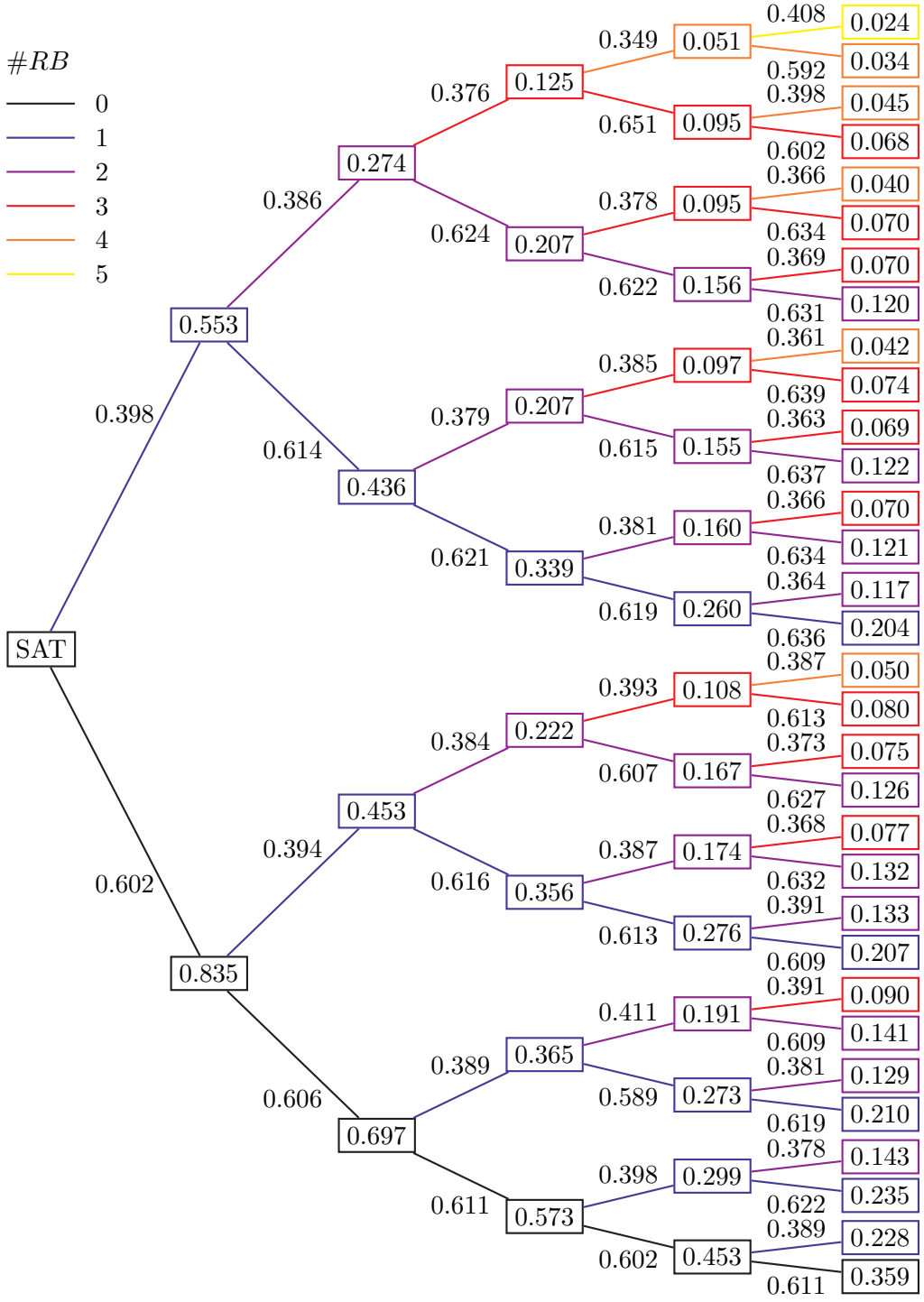

Figure 20. P_{sat} / B_{sat} tree of Jeroslow-Wang two-sided heuristic on random 3-SAT with $n = 120$, $\rho = 4.26$. $P_{\text{sat}}(d, i)$ values are shown in the vertices and $B_{\text{sat}}(d, i)$ values are shown on the edges.