Throwing Darts: Random Sampling Helps Tree Search when the Number of Short Certificates is Moderate

John P. Dickerson and **Tuomas Sandholm**
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

One typically proves infeasibility in satisfiability/constraint satisfaction (or optimality in integer programming) by constructing a tree certificate. However, deciding how to branch in the search tree is hard, and impacts search time drastically. We explore the power of a simple paradigm, that of throwing random *darts* into the assignment space and then using information gathered by that dart to guide what to do next. Such guidance is easy to incorporate into state-of-the-art solvers. This method seems to work well when the number of short certificates of infeasibility is moderate, suggesting the overhead of throwing darts can be countered by the information gained by these darts. We explore results supporting this suggestion both on instances from a new generator where the size and number of short certificates can be controlled, and on industrial instances from the annual SAT competition.

1 Introduction

Tree search is the central problem-solving paradigm in artificial intelligence, constraint satisfaction, satisfiability, and integer programming. There are two different tasks in tree search: A) finding a feasible solution (or a good feasible solution in the case of optimization), and B) proving infeasibility (or, if a feasible solution has been found in an optimization problem, proving that there is no better solution). These have traditionally been done together in one tree search. However, a folk wisdom has begun to emerge that different approaches are appropriate for proving feasibility and infeasibility. For example, local search can be used for the former while using a complete tree search for the latter (*e.g.*, (Kroc et al. 2009)). The two approaches are typically dovetailed in time—or run in parallel—because one usually does not know whether the problem is feasible (or, in the case of optimization, if the best solution found is optimal).

Assigning equal computational resources to both tasks comes at a multiplicative cost of at most two, and can lead to significant gains as the best techniques for each of the two parts can be used unhindered. Sampling-based approaches have sometimes been used for feasibility proving. In contrast, in this paper we explore a new kind of sampling-based approach can help for proving *infeasibility*. The techniques apply both to constraint satisfaction problems and to optimization problems. In the interest of brevity, we will mainly phrase them in the language of satisfiability.

One typically proves infeasibility by constructing a *tree certificate*, that is, a tree where each path (ordered set of variable assignments) terminates into infeasibility. The ubiquitous way of constructing a tree certificate is tree search; that is, one grows the tree starting from the root. However, deciding how to branch in tree search is hard (Liberatore 2000; Ouyang 1998), and the branching choices affect search tree size by several orders of magnitude.

We explore the idea of using random samples of the variable assignment space to guide the construction of a tree certificate. In its most general form, the idea is to repeatedly 1) throw a random *dart* into the allocation space, 2) minimize that dart, and 3) use it to guide what to do next.

At its most extreme, this idea can be instantiated as treeless tree search: throwing darts until a tree certificate can be constructed (*e.g.*, using dynamic programming) from the minimized darts directly without any tree search. A less radical instantiation of the idea is to throw some darts, use them to construct partial tree snippets (not necessarily from the root), then use the snippets to guide where more darts should be thrown, use that information to a) extend some of the snippets, b) delete other snippets, c) generate new snippets, d) meld snippets, and then repeat (by going back to dart throwing) until one of the snippets has grown into a tree certificate. An even less radical instantiation of the idea is to just have one tree search starting from the root, and to throw some darts at each node (where the variable assignments from the path form the dart) in order to guide variable ordering and curtail the search space. In this paper, a first step, we study an even simpler instantiation of this idea: one where we throw darts only at the root. We show that even when used in this way merely as a preprocessor, darts can yield significant speedups and reduce variance in runtime.[1]

Branching is so hard in practice and so important for

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]Other kinds of preprocessing techniques for satisfiability exist (Eén and Sörensson 2004; Lynce and Marques-Silva 2003), and algebraic preprocessing is standard in integer programming.

runtime that all of the best complete satisfiability solvers nowadays use random restarts (*e.g.*, (Baptista and Marques-Silva 2000; Kautz et al. 2002; Huang 2007; Kilinc-Karzan, Nemhauser, and Savelsbergh 2009)). If the run takes too long, the search is restarted anew—while saving the clauses learned via clause learning, which will be used to guide variable ordering and to curtail the search space in later runs.[2] One could think of the search runs in random restarts as samples of the search space from which the algorithm learns. However, in random restart approaches, unlucky variable ordering begets a huge search tree: in the search before the next restart, all the samples are very similar because they share all the variable assignments from the early part of the search path. Darts are not as easily misguided by bad variable ordering because the next throw is not confined to having the same variable assignments as the early part of the current search path. (Of course, both approaches are guided to an extent by the nogoods learned so far.)

Another prior family of techniques that could be viewed as a form of sampling is lookahead for determining which variable to branch on, such as in *strong branching*, its variants like *reliability branching* (Achterberg, Koch, and Martin 2005), and *entropic branching* (Gilpin and Sandholm 2011). Those approaches are very different from ours because they typically only do 1-step lookahead, and only in the context of the current search path.

The rest of the paper is organized as follows. Section 2 describes the general dart throwing framework. Section 2 provides a preliminary theoretical analysis of the framework under strict assumptions. Section 3 describes experiments performed on both generated and industrial instances from the most recent SAT competition. Sections 4 and 5 respectively give conclusions and ideas for future research using the darts framework.

2 Generating and Minimizing Each Dart

We use the underlying solver (MiniSat (Eén and Sörensson 2004) in the case of our experiments) to guide the construction of each dart. Each dart is in effect one search path from the root to a node where infeasibility of the path can be detected. The path then constitutes a conflict clause.

For the construction of the path, we use uniform random variable ordering. (This is only for darts throwing. For the actual tree search that follows the dart throwing phase, we let MiniSat use its own variable ordering heuristics.) However, we do employ repeated unit propagation using all the clauses in the formula: both the original ones and any prior minimized darts.

Proposition 2.1. *The dart throwing framework will never throw the same dart more than once,* i.e.*, we are sampling the assignment space without replacement.*

Proof. We will show that no clause in the formula (and recall that all darts thrown so far are included in the formula after the dart has been minimized) can subsume the variable assignments on the search path that is used to construct the current dart. For contradiction, suppose there is a clause c in the formula and that the search tree branch b used to throw the current dart includes all the variable assignments of c (and possibly also assignments of other variables). Then, at some point in the branch b, it included all but one of the variable assignments in c. At that point, unit propagation would have assigned to that last variable a value opposite of the same variable's value in c. Contradiction. □

To minimize each new dart right after it has been constructed, any minimization routine can be used. We use the minimization routine provided in MiniSat. After a dart has been thrown and minimized, we add it to the satisfiability formula that we are trying to prove unsatisfiable. Clearly, it does not affect the satisfiability of the formula because it is—like clauses added in clause learning—redundant. However, the added darts help guide variable ordering and prune the search space by terminating search paths early.

Once the desired number of darts have been added, we move from the dart throwing phase to the tree search phase, which is done using the unchanged solver (MiniSat in our case), but with the minimized darts now being part of the formula. We call this approach *Darts+DPLL*. We are now ready to begin the presentation of our study of the efficacy of the dart-based approach.

A *strong backdoor* of a search problem is a set of variables that, regardless of truth assignment, give a simplified problem that can be solved in polynomial time (Williams, Gomes, and Selman 2003). Specifically, in DPLL-style satisfiability, a strong backdoor of an unsatisfiable formula is a set of variables that, regardless of truth assignment, give a simplified formula that can be solved using repeated unit propagation. Discovering a strong backdoor is not easy; for instance, (Szeider 2005) discusses why strong backdoor discovery using DPLL-style subsolvers is intractable, while (Dilkina, Gomes, and Sabharwal 2007) describes tradeoffs between search speed and capability. It is well known that while pure random 3CNF formulas tend to have large backdoors—roughly 30% of variables (Interian 2003)—real-world problems (*e.g.*, logistics planning) and structured artificial problems (*e.g.*, blocks world planning) tend to have small strong backdoors (*e.g.*, (Hoffmann, Gomes, and Selman 2007)). Backdoor size has been shown to correlate with problem hardness, at least with DPLL-style search (Kilby et al. 2005).

A major motivation for our darts-based approach is that we hope that darts will identify backdoors, and thereby help in variable ordering, and ultimately lead to small search trees. But do they indeed accomplish that efficiently? That is the key question. For simplicity, we begin answering this question by an analysis of a setting where we assume that there exists what we coin a *strict* strong backdoor. A strict strong backdoor is a strong backdoor where every variable in the backdoor *must* be assigned before infeasibility of the search path can be discovered by the search algorithm. This differs from the standard notion of a strong backdoor, where *if* the backdoor variables are set, infeasibility can be shown

[2] A completely different approach is to try to restructure the search tree during the search (Glover and Tangedahl 1976; Ginsberg 1993; Ginsberg and McAllester 1994; Chvátal 1997; Hanafi and Glover 2002; Zawadzki and Sandholm 2010).

quickly—but it may be possible to show infeasibility even if some of the backdoor variables are not set.

Proposition 2.2. *Let the unsatisfiable formula that needs to be proven unsatisfiable have n variables. Let there be an (unknown) strict strong backdoor of size k. Then, after 2^k darts have been thrown without replacement and minimized, any tree search algorithm that branches only on variables that occur in the minimized darts will find the shortest tree certificate using a search tree of size at most 2^k.*

Proof. The construction of any dart will eventually terminate at a conflict. Furthermore, the learned clause corresponding to this conflict will get minimized to a clause containing only the variables in the k-backdoor. This is easy to prove. First, if the minimized clause contains those k variables, it cannot contain any other variables because in that case it would not be minimal (*i.e.*, some variable could be removed and the result would still be a conflict clause). Second, the minimized clause must contain all those k variables; otherwise, infeasibility of the assignment would have been proven without assigning all those k variables, which would contradict the definition of a strict strong backdoor.

Proposition 2.1 proves that as a dart is constructed, it cannot have the same assignment vector to those k variables as any previous dart. As there are only 2^k assignments of the k variables, all those assignments will be found with 2^k darts.

Finally, any tree search algorithm that branches on variables in the minimized darts alone can only branch on at most k variables because there are only k variables used among the darts. Thus, the size of the search tree is at most 2^k. Every search path terminates in a conflict because the minimized darts act as the relevant conflict clauses. □

The efficiency of this darts-based approach need not hinge on the assumption of existence of a strict strong backdoor (or even on the existence of a strong backdoor). However, without that assumption the analysis would become much more difficult due, at least, to the following reasons:

- There can be any number of tree certificates, and they can be of different sizes and of different tree shapes.
- The details of the dart minimization routine may affect the analysis. The routine might return the same minimized dart for a large set of unminimized variable assignments. For some other minimized dart, there might be only a small region of unminimized variable assignments that minimize to it. In fact, there can even be some minimized darts (minimal conflict clauses) that the routine never outputs, starting from any (complete) variable assignment.

For these reasons, we opt to study the darts paradigm with an extensive experimental section, where we cover a variety of random and real-world problems of varying degrees of difficulty. Through these experiments, we are able to discern clear patterns regarding the efficacy of this new technique.

3 Experiments

In this section, we explore the effects of dart throwing on the runtime of proving infeasibility. First, we introduce a general generator of unsatisfiable formulas with easily controllable strong backdoor properties. We can control both the size of the strong backdoor, as well as the expected number of unique strong backdoors. We study how the runtime depends on the number of darts thrown and the number of backdoors in a formula. Second, we provide results for industrial unsatisfiable instances used in the most recent SAT competition[3]. We show that the approach of throwing darts followed by MiniSat (we call this approach *Darts+DPLL*) sometimes results in a significant speedup while rarely resulting in a significant slowdown. We also show that throwing even a few dozen darts—a tiny portion of the expected number of darts needed to discover a full backdoor—results, surprisingly, in significant runtime changes.

Our experimental system was built on MiniSat 2.20 (Eén and Sörensson 2004), a well-known and highly competitive DPLL-based SAT solver. We kept as much of the original MiniSat solver unchanged as possible in order to isolate the impact of darts. The unchanged parts include MiniSat's variable ordering heuristics, random restart methods, clause learning technique, subsumption checking, and clause minimization technique. Our adjustments to the MiniSat system comprised only about 200 lines of code. We ran all the experiments in parallel on the world's largest shared-memory supercomputer—with 4096 cores and 32TB of shared memory—at the Pittsburgh Supercomputing Center.

We used MiniSat's default parameter settings: Luby restarts (`-luby`) with a base restart level of 100 (`-rfirst`) and increase factor of 2 (`-rinc`), deep conflict clause minimization (`-ccmin-mode`), a clause activity decay factor of 0.999 and variable activity decay factor of 0.95 (`-cla-decay`, `-var-decay`), no random variable selection (`-rnd-freq`), and default garbage collection and CPU/memory limits. These settings are those used by the default MiniSat solver. All reported times include any overhead incurred from dart throwing and minimization.

Random UNSAT Instances with Controllable Size and Number of Strong Backdoors

For these experiments, we developed a generator of random unsatisfiable formulas that allows easy control over the size and expected number of strong backdoors. Having such control is important because a generate-and-test approach to creating instances with desirable numbers and sizes of strong backdoors would be intractable. This is due to the facts that finding a backdoor is difficult (Szeider 2005) and, for many desired settings of the two parameters, the common instance generators extremely rarely create instances with such parameter values (*e.g.*, pure random 3CNF formulas tend to have large backdoors—roughly 30% of variables (Interian 2003)).

Our test suite consists of a set of graph coloring problems that, while originally satisfiable, are tweaked to prevent feasibility. Our generator is a generalized version of that introduced in (Zawadzki and Sandholm 2010). We first ensure that a randomly generated connected graph $G = (V, E)$

[3]`http://www.satcompetition.org/`

is k-colorable. This is done via the canonical transformation of a graph coloring instance into a SAT instance (see, e.g., (Van Gelder 2008)), followed by a satisfiability check using MiniSat. Unsatisfiable instances are discarded so they do not interfere with our control of the size and number of backdoors discussed next. Then, a number n of $(k+1)$-cliques are introduced into the graph, with n proportional to the desired number of short certificates. While not significantly changing the structure of the graph, no $(k+1)$-clique can be k-colored and thus this augmented graph cannot be k-colored. So, the SAT formula is unsatisfiable.

A SAT solver can conclude infeasibility of such formulas by reasoning only about the variables pertaining to a single $k+1$ vertex clique. In this way, we can control the size (k) of a short certificate in the propositional formula as well as the estimated number (n) of short certificates.[4]

Results. In the following experiments, we varied both the cardinality of the formula's set of short tree certificates of infeasibility (*i.e.*, number of cliques in the graph) and the number of darts thrown. Experiments were performed on these random, unsatisfiable graphs with $|V| = 100$ and $|E| = 1000$, with the number of colors $k = 10$. Translating the augmented unsatisfiable coloring problem to propositional logic yielded CNF formulas with 900 variables and between 12000 and 35000 clauses, depending on the number of cliques added. For every parameter setting, 40 instances (*i.e.*, random graphs) were generated, and on each of them, 40 independent runs of the algorithm were conducted.

Figure 1 compares the solution times for a darts-based strategy against that of pure MiniSat. Of particular note is the significant speedup from darts in the "middle ground" of 75–95 cliques. We hypothesize that when there are very few short certificates available, throwing darts will often result such certificates being missed; thus, we realize little benefit from darts. Conversely, when short certificates are ubiquitous (*e.g.*, coloring a complete graph), DPLL is likely to find such a certificate even without using darts. Another important aspect of dart throwing in these experiments is that it never hurts the runtime much.

Figure 2 provides some insight into the hypothesis given above by plotting the variance of the runtime of Darts+DPLL relative to MiniSat. For a small number of cliques, we see that throwing darts can actually increase variance slightly; intuitively, if darts happen to find a short certificate—however unlikely that may be—the overall runtime changes significantly. Similarly, when we have very few cliques, throwing a larger number of darts increases the variance more than a smaller number of darts, since there is a higher chance of hitting one of the rare short certificates. Conversely, when short certificates are ubiquitous (at and above 100 cliques), darts increase variance slightly due to interactions between the random information provided by darts and the structured pure MiniSat search. Most significantly, however, Figure 2 shows that darts reduce variance by an order of magnitude at 75–95 cliques. This suggests that at a certain (relatively small) number of initial darts, we are nearly guaranteed to find at least one short certificate, thus dramatically reducing the variance.

[4]The number of short certificates will not necessarily be exactly n: two randomly placed cliques can overlap in such a way as to create more than two cliques of the same size, given existing edges in the graph G. For large enough V and relatively low numbers of short certificates, we expect this to be rare and inconsequential.

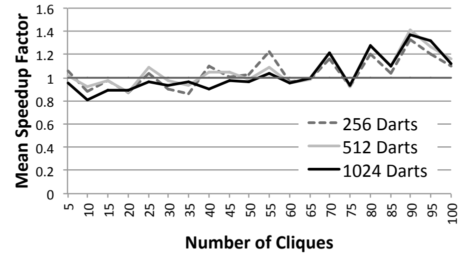

Figure 1: Mean relative speed of throwing 10, 250, and 500 darts before running MiniSat versus running MiniSat without darts. Points above 1 correspond to the darts strategy yielding a runtime improvement. There is a clear speedup in the 75–95 cliques range.

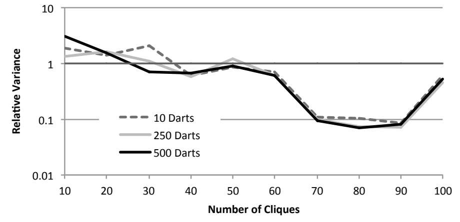

Figure 2: Log-scale runtime variance relative to a baseline of 1 provided by MiniSat without darts. Darts reduce the variance by an order of magnitude in the 75–95 cliques range. More darts reduce the variance more.

Zooming in on the "Sweet Spot" for Darts. The experiments above show that the performance of the dart-based strategy is tied to the number of short certificates in an unsatisfiable formula. When enough certificates exist to be found with high probability by darts, but not so many as to guarantee discovery by MiniSat without darts, we see a significant decrease in runtime and runtime variance. We now explore the most promising part of that 75–95 clique range in detail.

Figure 3 shows the relative speedups across 10 different experimental settings—each of $\{86, 87, \ldots, 94, 95\}$ infeasible cliques—forming the upper half of the promising range. Values above 1 represent a performance improvement over MiniSat without darts. To reduce uncertainty in these graphs, for each of the 10 clique values, 20 random graphs (*i.e.*, instances) were generated, and 20 different dart amounts (increasing roughly logarithmically from 0 to 25000) were thrown for each of those. All of this was repeated 40 times (*i.e.*, $11 \cdot 20 \cdot 20 \cdot 40 = 176000$ runs were conducted). Regardless of number of cliques, throwing just 250 darts provides, in terms of both mean and median, a clear decrease in runtime. Runtime monotonically decreases as we add more darts until between 5000 and 7500 darts. Af-

ter that, the overhead of adding new clauses to the original propositional formula (and the time spent on throwing and minimizing the darts themselves) outweighs the benefit of any information provided by the darts.

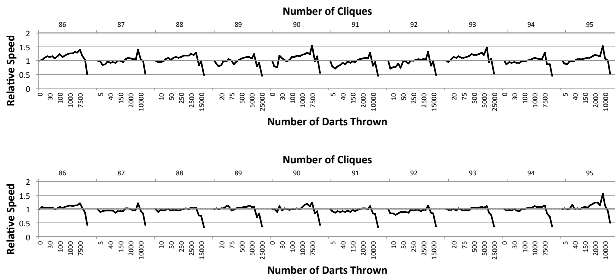

Figure 3: Mean (top) and median (bottom) relative speed of increasing numbers of darts (bottom axis) in each of $\{86, 87, \ldots, 95\}$ cliques (top axis). The optimal number of darts is in the 5000–7500 range.

Figure 4 studies how darts affect the variance in runtime. Values below 1 represent a reduction in variance compared to plain MiniSat. As in the broader case studied above, we see that speed improvements from darts tend to correlate with large decreases in variance. Our largest runtime improvements occurred at 5000–7500 darts; in that region the variance was reduced by over an order of magnitude. This suggests that throwing just a few thousand darts—with low computational overhead—seems to cut off the heavy tail of the runtime distribution, at least on these instances, better than MiniSat's tree search, which itself uses random restarts.

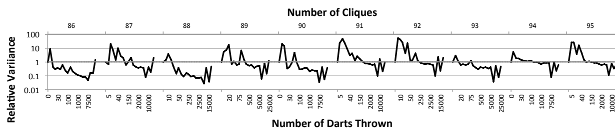

Figure 4: Log-scale runtime variance relative to a baseline of 1 provided by MiniSat without darts. Note the common behavior across all numbers of cliques—when we see a speedup from dart throwing, we also see a significant decrease in runtime variance.

Conclusions on the Random SAT Instances from Graph Coloring. The experiments we presented so far suggest that a simple dart throwing strategy provides legitimate benefit, both in terms of runtime and variance in runtime across multiple runs, on formulas with a "medium" number of short certificates. With too few short certificates, dart throwing can unluckily miss all of them, thus providing little new information to the subsequent DPLL search. With too many certificates, dart throwing provides redundant information to the tree search, resulting in little speedup. However, on instances in between these extremes, throwing even a few hundred darts—at almost no computational cost—can often result in both a significant runtime boost and a significant decrease in runtime variance. Successful dart throwing adds enough information to alleviate the variance introduced by the heavy tail of these runtime distributions.

Industrial UNSAT Instances

We now provide results for a number of unsatisfiable instances taken from industry. These real-world formulas were taken from the international SAT competition's Industrial Track, and have served as benchmarks for dozens of state-of-the-art SAT solvers. The current SAT competition groups instances as easy, medium, or hard; we provide results for instances from each group. Each instance was run at least 25 times for each of 24 different numbers of darts (increasing roughly logarithmically from 0 to 25000 darts).

Case Studies. First, we present three case study success stories for the dart throwing paradigm. The first, `hsat_vc11813`, is currently ranked as easy while the second, `total-5-13`, is medium and third, `AProVE07-27`, is hard. All three exhibit substantially different behavior, and darts help significantly on them all.

Runtime and variance results for the `hsat_vc11813` instance are given in Figure 5. MiniSat solves this problem in approximately 10 seconds. Relative mean speedup peaks at just below 25x (slightly under 0.5 seconds), after 70 darts. This peak correlates exactly with a decrease of between two and three orders of magnitude in runtime variance. This enormous speedup can be attributed to a particular behavior witnessed on some other easy instances as well: after throwing enough darts, the space of legal solutions becomes so constrained that the problem, given the extra clauses from the darts, is trivially infeasible at the root of the search tree. In the case of `hsat_vc11813`, the average length (*i.e.*, number of variables) of a dart was merely between 2 and 4. Smaller darts carve out a large portion of the search space; by throwing many of them, we whittle down the search space to something that can be solved quickly given a few variable assignments and unit propagation.

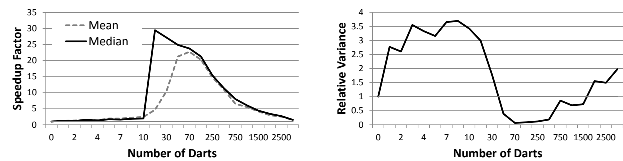

Figure 5: Mean and median runtime (left) and variance (right) relative to MiniSat without darts on the easy `hsat_vc11813` instance. The maximum mean speedup and the minimum relative variance occur at 70 darts.

Figure 6 shows runtime and variance results for the `total-5-13` instance. On our test machine, MiniSat without darts solves this instance in approximately 95 seconds; this aligns with the instance's medium hardness ranking. After just 1–3 darts, the average runtime drops to just over 65 seconds—albeit with nearly three times the variance. With just 20 darts, we maintain this speedup while simultaneously *reducing* variance compared to MiniSat without darts

by about a factor of two.

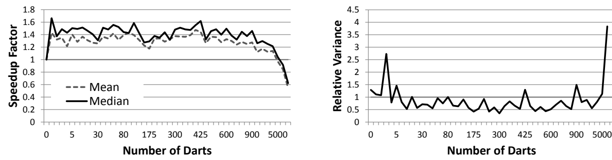

Figure 6: Runtime (left) and variance (right) relative to MiniSat without darts on the medium hardness `total-5-13` instance. There is a uniform speedup of approximately 50%, corresponding to a roughly 50% drop in runtime variance.

Figure 7 shows both runtime and average dart length of the hard `AProVE07-27` instance. Dart length is measured after the dart has been minimized, that is, after MiniSat's minimization procedure has been applied to the clause that is the dart so as to obtain a minimal clause from it (*i.e.*, a clause from which no variable assignment cannot be removed while keeping the clause a nogood). MiniSat solves this instance in slightly over 3000 seconds. Unlike on many other instances, adding tens of thousands of darts decreased runtime significantly. This is coupled with the darts' average clause size flattening out to a very low length of approximately 9, even after 25000 were thrown. Typically, due to the paring down of the full assignment space of variables, average lengths increase to a much longer length as more are thrown. Low length darts slice off large portions of the assignment space; in this instance, the constantly low length darts manage to prune the assignment space, yielding a significant speedup.

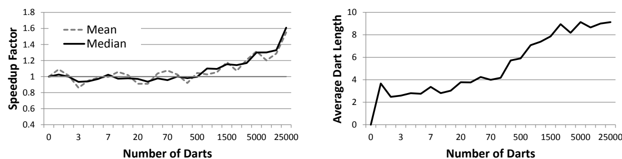

Figure 7: Mean and median runtime relative to MiniSat without darts (left) and average dart length (right) on the hard `AProVE07-27` instance. Along with a roughly monotonic speedup, dart length remains quite low.

Overall Results. We now present and discuss results of running the Darts+DPLL strategy on a wide variety of infeasible problem classes taken from the most recent international SAT competition. Figure 8 shows these results for 10, 250, and 2500 darts; we found that these three amounts were typically indicative of the overall performance (up to a maximum of 25000 darts tested).

Overall, these aggregate results show that, while in some cases dart throwing from the root node alone can help (significantly), the strategy alone is not a panacea. Our overall results show that darts tend to increase solution speed while decreasing runtime variance when the number of short certificates in the formula is neither extremely small nor extremely large; intuitively, some industrial instances, especially those found in the SAT competition, fall to either end of this extreme. It is with this in mind that we recommend, as future research, inclusion of a darts-based strategy in a portfolio-based solver like SATzilla (Xu et al. 2008). By using an ensemble of different search techniques and parameterizations of those techniques, a portfolio solver could leverage the Darts+DPLL method only on instances (like those detailed above) where a significant gain can be achieved.

Conclusions on the Industrial SAT Instances. The behavior of the Darts+DPLL strategy on real-world, infeasible industrial problems differs somewhat from that of our controlled test suite. Excitingly, while the graph coloring problems saw significant improvements in runtime up through 1000 or even 10000 darts, industrial instances generally required only a few dozen darts to realize the performance change (on those instances that saw a runtime decrease). A notable outlier occurred with the hard `AProVE07-27` instance, where clause lengths remained constant and small even as thousands of darts were thrown, resulting in a 60% runtime improvement. This hints that darts thrown on this hard instance continued to reveal large amounts of information about the search space; further exploration is required. Darts also caused the biggest changes in runtime on the industrial instances, up to a median speedup of 30x (on `hsat_vc11813` with 70 darts).

When darts decrease average runtime on an industrial instance, they also tend to decrease variance in runtime. When darts increase runtime, they tend to fail gracefully (unless time is wasted throwing thousands of darts), typically with only a 10-15% runtime increase. However, one instance—`hsat_vc11803`—responded quite poorly to even just 250 darts. MiniSat without darts solves this instance in under one second; throwing darts (and thus adding clauses into the formula) merely adds overhead on this very easy instance.

4 Conclusions

In this paper, we explored a simple strategy that uses random sampling of the variable assignment space to guide the construction of a tree certificate. This approach repeatedly throws a dart into the assignment space and, after minimizing the resulting clause, uses this information to decide what to do next (*e.g.*, how to guide search, where to throw the next darts). This can be used to find short certificates to satisfiability and constraint satisfaction problems, as well as prove infeasibility in integer programming.

We showed that a simple dart throwing strategy, where all darts are thrown before tree search begins, can easily be incorporated into a state-of-the-art SAT solver, MiniSat. In addition to darts helping guide its tree search, we were able to make use of the solver's fast mechanisms for guiding the construction of darts and for minimizing them. We provided experimental results on both unsatisfiable industrial instances from the recent SAT competition, as well as an instance generator that allows for easy control of the size and number of short certificates in the generated formula.

Our results show that darts tend to increase solution speed while decreasing runtime variance when the number of short

Unsatisfiable Industrial Instances									
Darts Thrown	10			250			2500		
Instance	Mean	Median	Var.	Mean	Median	Var.	Mean	Median	Var.
total-10-13-u	1.047	1.155	4.344	0.943	1.091	8.814	0.971	1.097	7.947
total-5-11-u	0.683	0.728	3.534	0.674	0.739	4.772	0.410	0.423	11.670
total-5-13-u	1.310	1.462	0.524	1.288	1.439	0.925	1.138	1.215	0.554
uts-l05-ipc5-h26-unsat	0.931	0.883	0.264	0.884	0.814	0.332	0.950	0.987	0.354
uts-l05-ipc5-h27-unknown	1.167	1.112	0.820	1.113	1.072	0.793	1.371	1.364	0.660
uts-l06-ipc5-h28-unknown	0.848	0.791	2.000	0.996	0.796	1.176	1.194	1.209	1.632
uts-l06-ipc5-h31-unknown	1.174	1.100	0.343	1.189	1.094	0.245	1.345	1.282	0.565
uts-l06-ipc5-h33-unknown	0.824	0.784	1.351	0.823	0.822	1.249	0.817	0.789	2.672
hsat_vc11803	0.878	0.876	2.305	0.304	0.293	18.775	0.048	0.052	3299
hsat_vc11813	2.423	1.952	11.703	14.773	14.379	0.014	2.523	2.686	2.236
q_query_3_l44_lambda	0.828	0.861	2.112	0.807	0.830	1.714	0.817	0.812	0.774
q_query_3_l45_lambda	0.814	0.8323	2.000	0.786	0.810	1.728	0.734	0.782	2.462
q_query_3_l46_lambda	0.826	0.721	2.342	0.807	0.732	3.888	0.788	0.804	2.625
q_query_3_l47_lambda	1.107	1.151	1.849	1.106	1.088	1.064	1.102	1.091	1.799
q_query_3_l48_lambda	0.793	0.825	2.288	0.795	0.868	1.636	0.803	0.780	1.158
gus-md5-04	1.211	1.290	1.274	0.816	0.931	2.329	0.090	0.073	216.9
gus-md5-05	0.765	0.978	5.418	0.629	0.686	6.101	0.142	0.132	88.49
gus-md5-06	0.973	0.945	0.587	0.859	0.875	0.655	0.299	0.251	11.66
gus-md5-07	1.064	1.045	0.395	1.008	0.937	0.364	0.483	0.430	4.037
gus-md5-09	1.100	1.026	0.341	1.033	0.927	0.492	0.773	0.634	0.635
eq.atree.braun.10.unsat	1.107	1.220	987.8	1.143	1.223	398.6	1.003	1.143	708.5
eq.atree.braun.8.unsat	0.900	0.891	5.755	0.867	0.852	5.447	0.665	0.624	22.51
eq.atree.braun.9.unsat	0.985	0.972	22.87	0.881	0.887	22.16	0.765	0.778	41.47
AProVE07-8	0.715	0.734	1822	0.560	0.515	966.7	0.670	0.758	3706
AProVE07-27	1.019	0.979	8.890	0.920	0.981	20.12	1.208	1.169	19.767
countbitsrotate016	1.221	1.264	11.39	1.067	1.050	3.065	1.395	1.445	13.25
countbitssrl016	0.833	0.920	10.15	0.587	0.592	15.82	0.445	0.488	21.59
countbitswegner064	0.839	0.893	33.92	0.760	0.743	51.67	0.591	0.628	496.8
icbrt1_32	0.982	1.007	1.248	0.990	1.000	3.712	1.149	1.240	8.800
minand128	1.081	1.005	0.997	1.198	1.163	2.375	0.939	0.982	2.866

Figure 8: Mean, median, and variance of runtime relative to Minisat without darts, for 10, 250, and 2500 darts. Unsatisfiable industrial instances are grouped by problem type, and were drawn from the most recent industrial SAT competition.

certificates in the formula is neither extremely small nor extremely large. From this, we conclude that darts are an effective way to curtail the heavy tail of certain runtime distributions. In random restarting, unlucky variable ordering not only creates a huge search tree, but also confines all samples of variable assignments to being paths in that poor tree. In contrast, our sampling is not misguided by bad variable ordering. The experiments show the benefits of this.

5 Future Research

A clear direction for future research is to expand the basic darts throwing strategy used in the experimental section of this paper to a more informed strategy, like those discussed in the introduction. For example, an interesting direction for future research would be to use darts as a semi-random restart strategy during tree search. Since, if darts work, they tend to work with very low variability and high speedup, one could throw a small number of darts—with low overhead—and then restart the subsequent tree search after a small time period if the dart throws did not yield immediate improvement. Combined with one of the standard randomized restart strategies—which have been shown to yield multiple orders of magnitude speedup on heavy-tail distributions (*e.g.*, Gomes, Selman and Kautz (1998))—it would be interesting to see how the added information from darts affects runtime speed and variance.

A dart-based strategy would be a particularly appropriate candidate for inclusion in a portfolio-based solver, (*e.g.*, in the satisfiability world, SATzilla (Xu et al. 2008)). When darts work, they tend to work quite well; however, there are some industrial instances that do not realize a runtime benefit. Inclusion in a portfolio of algorithms would help utilize the various wins of a dart-based strategy, while providing further information regarding when darts work, and how this general paradigm could be augmented.

One should study the more radical dart-based approaches offered in the introduction beyond throwing darts at the root of a search tree, possibly in conjunction with similar ideas by Richards and Richards (2000). Also, one could explore using darts in other proof systems beyond DPLL (*e.g.*, (Beame et al. 2010; Hertel et al. 2008)).

References

Achterberg, T.; Koch, T.; and Martin, A. 2005. Branching rules revisited. *Operations Research Letters* 33(1):42–54.

Baptista, L., and Marques-Silva, J. 2000. Using randomization and learning to solve hard real-world instances of satisfiability. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 489–494.

Beame, P.; Impagliazzo, R.; Pitassi, T.; and Segerlind, N. 2010. Formula caching in DPLL. *ACM Transactions on Computation Theory* 1.

Chvátal, V. 1997. Resolution search. *Discrete Applied Mathematics* 73(1):81–99.

Dilkina, B.; Gomes, C.; and Sabharwal, A. 2007. Tradeoffs in the complexity of backdoor detection. In *CP*, 256–270. Springer.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *SAT*, 333–336.

Gilpin, A., and Sandholm, T. 2011. Information-theoretic approaches to branching in search. *Discrete Optimization* 8:147–159. Early version in IJCAI-07.

Ginsberg, M., and McAllester, D. 1994. GSAT and dynamic backtracking. *Lecture Notes in Computer Science* 243–243.

Ginsberg, M. 1993. Dynamic Backtracking. *Journal of Artificial Intelligence Research* 1:25–46.

Glover, F., and Tangedahl, L. 1976. Dynamic strategies for branch and bound. *Omega* 4(5):571–576.

Gomes, C.; Selman, B.; and Kautz, H. 1998. Boosting combinatorial search through randomization. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*.

Hanafi, S., and Glover, F. 2002. Resolution search and dynamic branch-and-bound. *Journal of Combinatorial Optimization* 6(4):401–423.

Hertel, P.; Bacchus, F.; Pitassi, T.; and Gelder, A. V. 2008. Clause learning can effectively p-simulate general propositional resolution. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 283–290.

Hoffmann, J.; Gomes, C.; and Selman, B. 2007. Structure and problem hardness: Goal asymmetry and DPLL proofs in SAT-based planning. *Logical Methods in Computer Science* 3(1):6.

Huang, J. 2007. The effect of restarts on the efficiency of clause learning. In *IJCAI*, 2318–2323.

Interian, Y. 2003. Backdoor sets for random 3-SAT. *SAT* 231–238.

Kautz, H.; Horvitz, E.; Ruan, Y.; Gomes, C.; and Selman, B. 2002. Dynamic restart policies. In *AAAI*, 674–681.

Kilby, P.; Slaney, J.; Thiébaux, S.; and Walsh, T. 2005. Backbones and backdoors in satisfiability. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1368.

Kilinc-Karzan, F.; Nemhauser, G.; and Savelsbergh, M. 2009. Information-Based Branching Schemes for Binary Linear Mixed Integer Problems. *Mathematical Programming Computation* 1(4):249–293.

Kroc, L.; Sabharwal, A.; Gomes, C.; and Selman, B. 2009. Integrating systematic and local search paradigms: A new strategy for MaxSAT. *IJCAI*.

Liberatore, P. 2000. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence* 116(1-2):315–326.

Lynce, I., and Marques-Silva, J. 2003. Probing-based preprocessing techniques for propositional satisfiability. In *15th IEEE International Conference on Tools with Artificial Intelligence*, 105–110. IEEE.

Ouyang, M. 1998. How good are branching rules in DPLL? *Discrete Applied Mathematics* 89(1-3):281–286.

Richards, E., and Richards, B. 2000. Nonsystematic search and nogood learning. *Journal of Automated Reasoning* 24(4):483–533.

Szeider, S. 2005. Backdoor sets for dll subsolvers. *Journal of Automated Reasoning* 35:73–88.

Van Gelder, A. 2008. Another look at graph coloring via propositional satisfiability. *Discrete Applied Math.* 156(2):230–243.

Williams, R.; Gomes, C.; and Selman, B. 2003. Backdoors to typical case complexity. In *IJCAI*, volume 18, 1173–1178.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2008. Satzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32(1):565–606.

Zawadzki, E., and Sandholm, T. 2010. Search tree restructuring. Technical Report CMU-CS-10-102, Carnegie Mellon University.