# 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. 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.

## Throwing Darts

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.

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). 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 (Szeider 2005; Dilkina, Gomes, and Sabharwal 2007); a major motivation of this work is trying to identify backdoors, leading to better variable ordering and smaller search trees.

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. Interestingly, this simple approach appears to decrease average runtime as well as runtime variance when the number of short tree certificates proving the infeasibility of a formula is moderate.

## Experimental Design

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

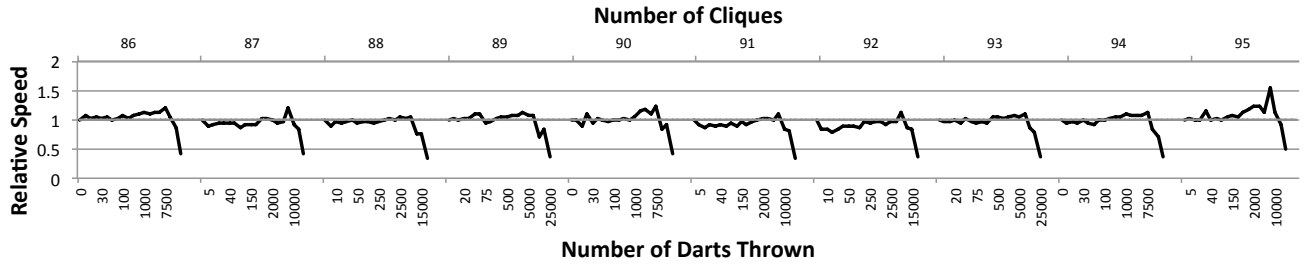


Figure 1: Median relative speed of increasing numbers of darts (bottom axis) in each of $\{86, \ldots, 95\}$ cliques (top axis).

that a randomly generated connected graph $G = (V, E)$ 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. 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.[1]

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 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 original and prior minimized darts.

## Results

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, 20 instances were generated, and on each of them, 20 independent runs of the algorithm were conducted.

We compare the solution times for a darts-based strategy against that of pure MiniSat. We witnessed significant speedup from darts in the "middle ground" of 80–95 cliques. We hypothesize that when there are very few short certificates available, throwing darts will often result in such certificates being missed. Conversely, when short certificates are ubiquitous (*e.g.*, coloring a complete graph), DPLL is likely to find a certificate quickly.

Figure 1 shows the relative speedups across 10 different experimental settings in this promising range. Values above 1 represent a performance improvement over MiniSat without darts. 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. After 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.

Furthermore, speed improvements from darts tend to correlate with large decreases in variance. Our largest runtime improvements occurred at 2500–7500 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.

**Conclusions.** These experiments suggest that a simple dart throwing strategy provides legitimate benefit, both in terms of runtime and variance in runtime, 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.

[1] 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.

## References


Dilkina, B.; Gomes, C.; and Sabharwal, A. 2007. Tradeoffs in the complexity of backdoor detection. *Principles and Practice of Constraint Programming* 256–270.

Eén, N., and Sörensson, N. 2004. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, 333–336. Springer.

Interian, Y. 2003. Backdoor sets for random 3-SAT. *Theory and Applications of Satisfiability Testing* 231–238.

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.

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

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 Mathematics* 156(2):230–243.

Williams, R.; Gomes, C.; and Selman, B. 2003. Backdoors to typical case complexity. In *International Joint Conference on Artifical Intelligence*, volume 18, 1173–1178.

Zawadzki, E., and Sandholm, T. 2010. Search tree restructuring. Technical Report CMU-CS-10-102, Carnegie Mellon University. Presented at the INFORMS Annual Conference, 2010.