Backdoors To Typical Case Complexity

Ryan Williams *
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
ryanw@cs.cmu.edu

Carla P. Gomes †
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
gomes@cs.cornell.edu

Bart Selman †
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
selman@cs.cornell.edu

Abstract

There has been significant recent progress in reasoning and constraint processing methods. In areas such as planning and finite model-checking, current solution techniques can handle combinatorial problems with up to a million variables and five million constraints. The good scaling behavior of these methods appears to defy what one would expect based on a worst-case complexity analysis. In order to bridge this gap between theory and practice, we propose a new framework for studying the complexity of these techniques on practical problem instances. In particular, our approach incorporates general structural properties observed in practical problem instances into the formal complexity analysis. We introduce a notion of "backdoors", which are small sets of variables that capture the overall combinatorics of the problem instance. We provide empirical results showing the existence of such backdoors in real-world problems. We then present a series of complexity results that explain the good scaling behavior of current reasoning and constraint methods observed on practical problem instances.

1 Introduction

Most interesting AI formalisms for reasoning, planning, and learning have been shown to be worst-case intractable. In the eighties and early nineties, such negative complexity results led to an extensive search for tractable subclasses of the general formalisms. Unfortunately, these tractable subclasses were often too restrictive for real-world applications. In the mid-nineties, we saw the emergence of a more practical approach to computationally hard problems in AI, with the introduction of fast satisfiability solvers and fast constraint based reasoning methods [17]. For example, in planning we saw the success of constraint-based planners, such as Graphplan [2] and SatPlan [13], and most recently, heuristic search based planners, *e.g.*, [11; 8; 1]. Somewhat surprisingly, on practical problem instances these methods scale well beyond what one might expect based on a formal complexity analysis. In fact, current state-of-the-art SAT solvers can handle problem instances, as they arise in finite model-checking and planning, with up to a million variables and five million clauses [15]. The success of these methods appears to hinge on a combination of two factors: (1) practical combinatorial problem instances generally have a substantial amount of (hidden) tractable sub-structure, and (2) new algorithmic techniques exploit such tractable structure, through, *e.g.*, randomization and constraint learning.

These developments suggest that a standard worst-case complexity analysis does not capture well the true complexity of typical problem instances encountered in practical applications. Theoretical computer scientists have been well-aware of the limitations of worst-case complexity results and have explored alternatives, such as average-case complexity and smoothed analysis [20]. In average-case analysis, one studies the computational cost of solving problem instances drawn from a predefined problem distribution. Such an analysis can provide valuable insights, as demonstrated by the work on uniform random instance distributions (*e.g.* random k-SAT). However, the relatively basic distributions for which one can obtain average-complexity results appear to be quite far removed from the instance distributions one encounters in practice. In fact, formally defining the distribution of real-world problem instances is generally an open problem in itself. Smoothed analysis attempts to unify worst-case and average-case, but suffers from limited applicability: it works well on algorithms for problems defined over dense fields such as the simplex algorithm, but the applicability of smoothed analysis on discrete problem domains is unclear.

An alternative approach, which we will pursue in this paper, is to identify special structural properties common to known problem instances and rigorously show how clever algorithms can exploit such properties. Informal insights about what such special structure might be are currently already used in the design of, for example, branching and variable choice heuristics in combinatorial search methods. A common feature of these techniques is an understanding that different groups of variables in a problem encoding often play quite distinct roles. For example, at the highest level,

*Supported in part by an NSF Graduate Fellowship and the NSF ALADDIN Center.
†Research supported by AFOSR, Darpa, and the NSF.

one can distinguish between dependent and independent variables. The dependent or auxiliary variables are needed to obtain compact problem encodings but the true combinatorics arises from the independent variables; *e.g.,* the independent variables in an encoding of a planning domain represent the various operators applicable in a given state of the world, whereas the dependent variables encode the consequences of selecting a particular operator. A plan search technique that branches purely on the independent variables can obtain substantial speedups over search methods that do not exploit variable dependencies [4].

Another powerful intuition in the design of search methods is that one wants to select variables that *simplify the problem instance as much as possible* when these variables are assigned values. This intuition leads to the common heuristic of branching on the most constrained variable first. In terms of Boolean satisfiability, this amounts to, in effect, focusing in on the tractable substructure of the problem, namely the unit clauses (1-SAT structure) and the binary clauses (2-SAT structure). The true effectiveness of this approach arises from the fact that setting most constraint variables also simplifies higher arity clauses, which either become satisfied or in turn shrink themselves eventually to binary or unary clauses.

These general insights have been incorporated in state-of-the-art SAT and constraint solvers, and their effectiveness has been demonstrated empirically on a significant number of benchmark problems [18]. However, a more formal underpinning explaining the practical success of these strategies has been lacking. In this paper, we introduce a formal framework directly inspired by these techniques and present rigorous complexity results that support their effectiveness.

Preview of results. We first introduce the notion of "backdoor" variables. This is a set of variables for which there is a value assignment such that the simplified problem can be solved by a poly-time algorithm, called the "sub-solver". The sub-solver captures any form of poly-time simplification procedure as used in current SAT/CSP solvers. We also consider the notion of a "strong backdoor" where any setting of the backdoor variables leads to a poly-time solvable subproblem. The set of all problem variables forms a trivial backdoor set, but many interesting practical problem instances possess much smaller backdoors and strong backdoors. We will study backdoors in several practical problem instances, and identify backdoors that contain only a fraction of the total number of variables. For example, the SAT encoding of a logistics planning problem (`logistics.d.cnf`) contains a backdoor with only 12 variables out of a total of nearly 7,000 variables. When given a set of backdoor variables of a problem instance, one can restrict the combinatorial search by branching only on the backdoor variables and thus search a drastically reduced space.

In general, finding a small set of backdoor variables for a problem instance is, however, itself a computationally hard problem. One contribution of this paper is that we formally show how the presence of a small backdoor in a problem provides a concrete computational advantage in solving it. We analyze three scenarios. First, we consider a deterministic scenario with an exhaustive search of backdoor sets. We show that one obtains provably better search complexity when the backdoor contains up to a certain fraction of all variables. We then show that a randomized search technique, which in effect repeatedly guesses backdoor sets, provably outperforms a deterministic search. Finally, in our third scenario we consider the availability of a variable selection heuristic, which provides guidance towards the backdoor set. This strategy can yet further reduce the search space. Table 1 gives a high-level summary of the results. By exploiting restart strategies, we can identify a polynomially solvable case when the backdoor contains at most $\log(n)$ variables. We believe that this final scenario is closest to the behavior of current effective SAT and constraint solvers. Our formal analysis also suggests several novel algorithmic strategies that warrant further empirical exploration.

$B(n)$	deterministic	randomized	heuristic
n/k	small $exp(n)$	smaller $exp(n)$	tiny $exp(n)$
$O(\log n)$	$\left(\frac{n}{\sqrt{\log n}}\right)^{O(\log n)}$	$\left(\frac{n}{\log n}\right)^{O(\log n)}$	$poly(n)$
$O(1)$	$poly(n)$	$poly(n)$	$poly(n)$

Table 1: Time bounds for solving CSPs in the various scenarios considered in this work. $B(n)$ is an upper bound on the size of the smallest backdoor, where n is the number of variables in the problem. k is a fixed constant. Empirical results (Section 3) suggest that for practical instances the backdoor is often a relatively small fraction of n, *e.g.*, $n/100$, or even of size $\log n$.

2 Hidden structure: Backbones and Backdoors

Our approach and analysis applies both to SAT and CSP problems [17]. SAT is the abbreviation for the well-studied Boolean satisfiability problem. CSP is the abbreviation for the more general problem of constraint satisfaction.

A CSP problem, C, is characterized by a set $V = \{x_1, x_2, \ldots, x_n\}$ of variables, with respective domains D_1, D_2, ..., D_n (which list the possible values for each variable) and a set of constraints. A constraint is defined on a subset of variables $S_i \subset V$ denoting the variables' simultaneous legal assignments. That is, if $S_i = \{x_{i_1}, x_{i_2}, \ldots, x_{i_r}\}$, then the constraint defines a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_r}$. To simplify notation, we will assume that all variables have the same domain D. We use d to denote the size of D. An assignment a is a function from variables to D. A solution to a CSP is a complete variable assignment that satisfies all constraints. A partial assignment defines the values of a subset of the variables in V. SAT is a special case of CSP with only Boolean variables ($D = \{True, False\}$) and constraints given in the form of clauses. A clause is a disjunction of literals and a literal is a Boolean variable or its negation.

We use the notation $C[v/x]$ to denote the simplified CSP obtained from a CSP, C, by setting the value of variable x to value v. (A constraint involving x is simplified by keeping only the allowed tuples that have x assigned to v.) Let a_S :

$S \subseteq V \rightarrow D$ be a partial assignment. We use $C[a_S]$ to denote the simplified CSP obtained by setting the variables defined in a_S. In a SAT problem, this corresponds to simplifying the formula by fixing the truth values of some of the variables.

Our goal is to capture structural properties of real world problem instances. We start by reviewing the concept of a backbone in a SAT/CSP problem, as introduced in [14]. A variable is called a backbone variable if in all solutions to the CSP the variable is assigned the same value. Such variables are also called frozen variables [6]. Backbone variables are useful in studying the properties of the solution space of a constraint satisfaction problem.

Definition 2.1 [backbone] *S is a* backbone *if there is a unique partial assignment $a_S : S \rightarrow D$ such that $C[a_S]$ is satisfiable.*

We contrast this variable type with the kind we introduce, backdoors. Backdoors are variable subsets defined with respect to a particular algorithm; once the backdoor variables are assigned a value, the problem becomes easy under that algorithm. (Note that contrarily to the backbone there can be different sets of backdoor variables.)

To begin our exposition of backdoors, we define the sort of algorithms we have in mind. We will call them *sub-solvers*, as they solve tractable subcases of the general constraint satisfaction problem.

Definition 2.2 *A sub-solver A given as input a CSP, C, satisfies the following:*

• *(Trichotomy) A either rejects the input C, or "determines" C correctly (as unsatisfiable or satisfiable, returning a solution if satisfiable),*

• *(Efficiency) A runs in polynomial time,*

• *(Trivial solvability) A can determine if C is trivially true (has no constraints) or trivially false (has a contradictory constraint),*

• *(Self-reducibility) if A determines C, then for any variable x and value v, then A determines $C[v/x]$.*

For instance, A could be an algorithm that solves 2-SAT instances but rejects all other instances. It is important to note that the results we will show in this paper are independent of a particular sub-solver; our results will hold for *any* A satisfying the above four properties.

In what follows, let A be a sub-solver, and C be a CSP.

We first consider a notion of "backdoor" that is suitable for satisfiable CSPs.

Definition 2.3 [backdoor] *A nonempty subset S of the variables is a* backdoor in C for A *if for some $a_S : S \rightarrow D$, A returns a satisfying assignment of $C[a_S]$.*

Intuitively, the backdoor corresponds to a set of variables, such that when set correctly, the sub-solver can solve the remaining problem. In a sense, the backdoor is a "witness" to the satisfiability of the instance, given a sub-solver algorithm.[1] We also introduce a stronger notion of the backdoor to deal with both satisfiable and unsatisfiable (inconsistent) problem instances.

Definition 2.4 [strong backdoor] *A nonempty subset S of the variables is a* strong backdoor in C for A *if for all $a_S : S \rightarrow D$, A returns a satisfying assignment or concludes unsatisfiability of $C[a_S]$.*

In contrast to backbones which are *necessarily* set to a certain value, a (strong) backdoor S is *sufficient* for *solving* a problem. For example, when given the backdoor for a SAT problem, the search cost is of order $|D|^{|S|}$. (Simply check all possible assignments of S.) This means if S is relatively small, one obtains a large improvement over searching the full space of variable/value assignments.

We observe that *independent* variables are a particular kind of backdoor. As stated in [12], they are a set S of variables for which all other variables may be thought of as *defined* in terms of S. For example, a maximal subset of independent variables in a SAT encoding of a hardware verification problem is a backdoor for unit propagation, as the other variables' values may be directly determined after setting the independent ones [19].

There are two key questions concerning backdoors:

- What is the size of the backdoor in practical problem instances?
- When taking into account the cost of searching for a backdoor set, can one still obtain an overall computational advantage in solving the CSP?

We address these two key questions below. We will first show that practical problem instances can have surprisingly small backdoors. In the subsequent section, we show how even by taking into account the cost of searching for a backdoor, one can provably obtain an overall computational advantage by using the backdoor. As we will see, the magnitude of this improvement is, of course, a function of the size of the backdoor.

3 Size of backdoors

We did an empirical study of the size of backdoors in several practical SAT instances, using the SAT solver Satz-rand, a randomized version of Satz [16]. Satz incorporates powerful variable selection heuristics and an efficient simplification strategy (*i.e.*, a good sub-solver). We modified Satz-rand to trace the variables selected for branching, and to keep track of the minimum number of variables that need to be set before Satz-rand's simplification found a satisfying assignment efficiently. (We are currently modifying this procedure to also handle unsatisfiable instances and find strong backdoors.)

[1] Observe that *any* satisfiable CSP has a backdoor of size at most $n-1$; however, we will see that significantly smaller backdoors arise in practice and give a computational advantage in search.

instance	# vars	# clauses	backdoor	fract.
logistics.d	6783	437431	12	0.0018
3bitadd_32	8704	32316	53	0.0061
pipe_01	7736	26087	23	0.0030
qg_30_1	1235	8523	14	0.0113
qg_35_1	1597	10658	15	0.0094

Table 2: Size of backdoors for several practical SAT instances.

Table 2 summarizes our results. Our instances are from a variety of domains [18]. These instances are now well within the range of the fastest current solvers, such as Chaff [15]. However, they are non-trivial and cannot be solved with the previous generation of SAT solvers, *e.g.* Tableau [3]. Clearly, the new solvers are better able to discover and exploit hidden structure, such as small backdoors. In fact, as we can see from the table, these instances have fairly tiny backdoors. That is, only a very small fraction of all variables can be used to "unlock" a satisfying assignment. We conjecture that such small backdoors occur in many other real-world problem instances.

4 Exploiting backdoors formally

We will analyze three, increasingly powerful strategies: *deterministic*, *randomized*, and *heuristic branching variable selection*. The first two are meant to work for any CSP where the instance has a small fraction of backdoor variables, with respect to the sub-solver. The randomized strategy generally outperforms the deterministic one with high probability ($1 - 1/n$, where n is the number of variables). This reflects the performance gain found in practice when backtracking SAT solvers are augmented with randomization [15; 9]. The third strategy yields tighter runtime bounds than the first two, but requires us to assume the existence of a good heuristic for choosing backdoor variables (which we find to be the case in practice).

4.1 Deterministic strategy

The deterministic procedure may be construed as a generalization of iterative deepening that runs over all possible search trees of each depth. We assume the algorithm has access to a particular sub-solver A running in $T(n)$ (polynomial) time, which defines the backdoor variables, and C is an arbitrary CSP instance.

Algorithm 4.1 Given a CSP C with n variables,

For $i = 1, \ldots, n$,

For all subsets S of the n variables with $|S| = i$,

Perform a standard backtrack search (just on the variables in S) for an assignment that results in C being solved by sub-solver A.

An analogous algorithm works for finding and exploiting *strong backdoors* in a CSP to prove *unsatisfiability*: simply keep track of whether all assignments to the variables in S result in C being a contradiction (as determined by A). All of the following we will say holds for strong backdoors and unsatisfiable CSPs under this modified algorithm.

Note the procedure uses only polynomial time for CSPs with a constant sized backdoor. We are interested in the case where a backdoor of size $B(n)$ exists, for some $B(n) < n/2$ almost everywhere. The following gives a simple runtime bound in terms of n and $B(n)$.

Theorem 4.1 *If C has a variable domain of size d and a backdoor of size at most $B(n)$, then Algorithm 4.1 runs in $O(p(n)(\frac{d \cdot n}{B(n)^{1/2}})^{B(n)})$ time, for some polynomial $p(n)$.*

Proof. The runtime is bounded by $T(n)\sum_{i=1}^{B(n)} \binom{n}{i} d^i$ (recall $T(n)$ is a runtime bound on A). Since $B(n) < n/2$, $\sum_{i=1}^{B(n)-1} \binom{n}{i} \leq q(n)\binom{n}{B(n)}$ for some polynomial $q(n)$. This and $\sum_{i=1}^{B(n)-1} d^i \leq d^{B(n)}$ imply that the runtime is dominated by the last term of the sum. Putting it together, the bound is $T(n)\sum_{i=1}^{B(n)} \binom{n}{i} d^i \leq T(n)q(n)\binom{n}{B(n)} d^{B(n)} \leq T(n)q(n)\frac{(d \cdot n)^{B(n)}}{B(n)!} \leq p(n)\frac{(d \cdot n)^{B(n)}}{B(n)^{B(n)/2}}$ asymptotically, for $p(n) = T(n) \cdot q(n)$. $\square$

The theorem implies that when small backdoors (or strong backdoors) are present, a substantial speedup almost always results. For example:

Corollary 4.1 *If C has a backdoor of size $B(n) = O(\log n)$, then C is solvable in $(\frac{n}{(\log n)^{1/2}})^{O(\log n)}$ time.*

In our exposition of heuristic branching variable selection, we will see an improvement on this (a poly-time bound). For a visual representation of the deterministic strategy's runtime, when $d = 2$ and backdoors of size n/k are considered, see Figure 1. This graph also indicates the following corollary in the case of SAT (proof omitted):

Corollary 4.2 *For Boolean formulas with a backdoor of size at most $n/4.404$, Algorithm 4.1 solves the formula in $O(c^n)$ time, where $c < 2$.*

As we have seen in the previous section, in practice, backdoors can be quite tiny ($\approx 1/566 = 0.18\%$ of the variables, for `logistics.d.cnf`). Therefore, these results have real bearing on the improved solvability of real-world CSPs.

4.2 Randomized strategy

Better performance results from adding randomization. This speed-up formally verifies a well-known fact about real-world solvers: augmenting a solver with randomization can dramatically improve performance [9; 10].

Again, we assume a sub-solver A is on tap, with runtime $T(n)$. Let $B(n)$ be a poly-time computable function on $\mathbb{N}$ that bounds the backdoor size, and b be a parameter to be later determined. The idea is to repeatedly choose random subsets of variables that are larger than $B(n)$, searching these subsets for a backdoor.

Algorithm 4.2 Given a CSP C with n variables,

Repeat $n(\frac{(n/B(n)-1)}{(b-1)})^{B(n)}$ times (and at least once):

> Randomly choose a subset S of the n variables, of size $b \cdot B(n)$. Perform a standard backtrack search on variables in S. If C is ever solvable by A, return the satisfying assignment.

As before, an analogous algorithm works for general (satisfiable or unsatisfiable) CSPs with strong backdoors: if every leaf in the search tree ends with A reporting unsatisfiability, then the C is unsatisfiable.

The algorithm as stated requires *a priori* knowledge of $B(n)$. This may be corrected by choosing a constant $\alpha > 1$, then running the algorithm assuming a backdoor of size 1. If that fails, run it again assuming a backdoor of size α, then α^2, α^3, etc., until a solution to C is found.

Theorem 4.2 *If C has a backdoor of size $B(n)$, Algorithm 4.2 finds a satisfying assignment with probability approaching 1.*

Proof. Given there is a $B(n)$-sized backdoor in C, the probability that a randomly chosen S of size $j \geq B(n)$, $j < n$ contains the entire backdoor is at least

$$\binom{n-B(n)}{j-B(n)}/\binom{n}{j} = \frac{j(j-1)\cdots(j-B(n)+1)}{n(n-1)\cdots(n-B(n)+1)} \geq \left(\frac{(j-B(n))}{(n-B(n))}\right)^{B(n)}.$$

Setting $j = b \cdot B(n)$, the probability that backtracking results in A finding a solution is at least $\left(\frac{(bB(n)-B(n))}{(n-B(n))}\right)^{B(n)} = (\frac{(b-1)}{(n/B(n)-1)})^{B(n)}$, due to the self-reducibility property of A. Repeating this experiment $n(\frac{(n/B(n)-1)}{(b-1)})^{B(n)}$ times, the algorithm succeeds with probability at least $1 - 1/n$. $\square$

One can show that the algorithm runs in $O\left(\max\{1, n(\frac{(n/B(n)-1)}{(b-1)})^{B(n)}\}d^{b \cdot B(n)}T(n)\right)$ time. It remains to choose b to minimize this expression. As b depends directly on $B(n)$, we evaluate two natural cases for $B(n)$.

• When $B(n) = k \log n$ for some constant k, the runtime is $n(\frac{(n/(k\log(n))-1)}{(b-1)})^{k \log n} n^{b \cdot k'}$ for some constant k'. For large k', the runtime is optimized when b is constant; it is $(\frac{n}{\log n})^{O(\log n)}$, an improvement over the deterministic bound.

• When $B(n) = n/k$ for some constant k, we can show the runtime is minimized when $b = \frac{\ln(d)k}{1+\ln(d)}$, resulting in a $O(d^{n/k \cdot (1+\ln(d))/\ln(d)} (\ln(d)(k-1))^{n/k})$ time bound. For example, when $d = 2$ (the case of SAT), $b = k/2.443$ and the following holds.

Corollary 4.3 *For Boolean formulas with at most $n/2.443$ backdoor variables, Algorithm 4.2 solves the formula in $O(c^n)$ time, where $c < 2$.*

In the Corollary, c is a function of k. See Figure 1.

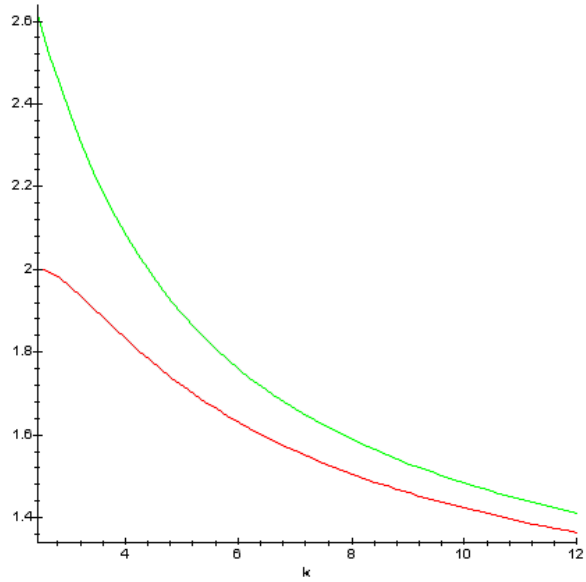

Figure 1: **Improved exponential time.** When $d = 2$ (SAT) and the size of the backbone is a constant fraction of the number of variables ($B(n) = n/k$), the runtime of Alg. 4.1 (deterministic) and 4.2 (randomized) is of the form c^n. c (vertical axis) is a function of k. The top curve gives c as a function of k for the deterministic procedure. The bottom curve gives c for the randomized procedure. Note that for $k \geq 2.443$, the randomized algorithm performs exponentially better than 2^n, whereas such an exponential improvement for the deterministic algorithm does not occur until $k \geq 4.04$.

4.3 Heuristic strategy

So far, we have considered general systematic and randomized search strategies for finding and exploiting backdoors. However, practical combinatorial solvers generally use heuristics to guide the variable selection process. As noted in the introduction, a common principle is to first branch on variables that simplify an instance the most. In effect, this means such heuristics steer the variable choice towards variables in a backdoor set. We will now formally analyze such heuristic guidance.

Restart Strategies for Heuristic Search. By incorporating the notion of a variable choice heuristic into our framework, our results are further sharpened. We consider the case where a randomized depth-first search (DFS) solver with a sub-solver A is running on an instance C having a backdoor of size B. The solver chooses variables to branch on according to a heuristic H, which has a success probability of at least $1/h$ of choosing a backdoor variable at any point in the search. We will use the notation **(DFS,H,A)** to denote a solver with the above properties.

Informally, a *restart strategy* is simply a policy that restarts a solver after running it for a specified amount of time, until a solution is found. Our main result here gives a condition under which a polynomial time restart strategy exists for DFS solving CSPs with small backdoors.

Theorem 4.3 *If the size of a backdoor of a CSP C is $B \leq$*

$\frac{c \log n}{\log h + \log d}$ *for some constant c, then* **(DFS,**H**,**A**)** *has a restart strategy that solves* C *in polynomial time.*

Proof. Since the probability of choosing a backdoor variable is at least $1/h$, the probability that we consecutively choose them is $1/h^B$. The probability of choosing the correct solution with only a polynomial amount of backtracking in the DFS is at least $1/d^{B-k \log n}$ for some constant k. Suppose $1/d^{B-k \log n} \cdot 1/h^B \geq 1/n^e$, for some constant e. Then by restarting the solver after every $T_A(n)$ steps (where $T_A(n)$ is the runtime of A), there is $1/n^e$ probability in each run that the backdoor will be found within a $O(n^k)$ amount of backtracking, and set correctly. From this one can show that the above inequality holds precisely when $B \leq \frac{c \log n}{\log h + \log d}$ for some constant c. □

An analogous result holds for strong backdoors. It turns out that the given bound on B is asymptotically tight; we will not prove that here. When the variable domain size is constant (*e.g.* SAT, 3-coloring, etc.), we have the following. Let f be any poly-time computable function on the natural numbers.

Corollary 4.4 *Given CSPs with a* $O(\frac{\log n}{\log f(n)})$ *backdoor for which* H *has success probability* $1/f(n)$, **(DFS,**H**,**A**)** *has a polynomial time restart strategy.*

When the success probability is constant, then CSPs with $O(\log n)$ backdoors can be solved using a polynomial time restart strategy on **(DFS,**H**,**A**)**. This result is the best possible in terms of backdoor size, as it would take super-polynomial time to search for a solution among $\omega(\log n)$ backdoor variables. The heuristic search runtime when $B(n) = n/k$ is still exponential, but this exponential drops dramatically as n/k decreases, even when compared to the previous two algorithms. That is, the runtime is on the order of $c^{n/k}$, where $c = d \cdot 1/h$ (recall d is the domain size and $1/h$ is success probability).

Formal Discovery of Heavy-Tails in Heuristic Search. We briefly outline our theoretical results connecting the heuristic search model described earlier with heavy-tailed runtime phenomena found empirically [9]. It was conjectured that "critically constrained" variables were a cause of the heavy-tailed behavior. We can prove that small sets of backdoor variables lead to runtime profiles that are bounded from below by heavy-tails.

The analysis that achieves this result introduces a self-similar binary tree structure, which we call a *variable choice tree*. Such trees recursively model a heuristic's selection of backdoor variables; as more backdoor variables are chosen, the resulting search cost is much lower. It turns out that backtracking solvers with variable choice heuristics can be modeled precisely by these variable choice trees, when the size of a backdoor in the instance is small. Analysis of these trees leads to the following:

Theorem 4.4 *(Heavy-tail lower bound) If the backdoor size of an CSP* C *is* $o(n/\log n)$, *then the runtime distribution of* **(DFS,**A**,**H**)** *on* C *is lower-bounded by a Pareto-Levy distribution, when the success probability of* H *is constant.*

5 Conclusions

We have formalized the idea of backdoor variables in CSP/SAT instances. Backdoor variables can be used to significantly reduce the search needed in solving CSP/SAT problems. We showed that practical instances can have surprisingly small backdoors. We also provided a detailed formal analysis demonstrating that one can obtain a concrete computational advantage by exploiting such backdoors.

References

[1] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *AIJ* 116, 2000.

[2] A. Blum and M. Furst. Fast planning through planning graph analysis. *Proc. IJCAI-95*, 1636–1642, 1995.

[3] J. Crawford. Tableau SAT solver. www.informatik.tu-darmstadt.de/AI/SATLIB/

[4] A. Gerevini and I. Serina. Planning as Propositional CSP: from Walksat to Local Search. *CONSTRAINTS*, to appear.

[5] H. Chen, C., Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search, *Proc. CP'01*, 408–422, 2001.

[6] J. Culberson and I. P. Gent. Well out of reach: why hard problems are hard, Tech Report APES-13-1999, APES Research Group, 1999.

[7] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming, *ACM Computing Surveys* 33(3): 374–425, 2001.

[8] H. Geffner. Perspectives on Artificial Intelligence Planning, *Proc. AAAI*, 1013–1023, 2002.

[9] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems, *J. Autom. Reasoning* 24: 67–100, 2000.

[10] C. Gomes, B. Selman, and H. Kautz. Boosting Combinatorial Search Through Randomization, *Proc. AAAI'98*, 1998.

[11] J. Hoffman and B. Nebel. The FF Planning System, *JAIR* 14:253–302, 2001.

[12] H. Kautz, D. McAllester, and B. Selman. Exploiting Variable Dependency in Local Search, *Proc. IJCAI*, 1997.

[13] H. Kautz and B. Selman. Planning as satisfiability, *Proc. 10th European Conf. on AI*, 359–363, 1992.

[14] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky, Determining Computational Complexity from Characteristic 'Phase Transitions', *Nature* 400: 133–137, 1999.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver, *Proc. DAC*, 2001.

[16] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems, *Proc. IJCAI*, 366–371, 1997.

[17] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, 2nd ed., Prentice Hall, 2002.

[18] SAT benchmarks. http://www.lri.fr/~simon/satex/satex.php3

[19] J. P. M. Silva, *Search Algorithms for Satisfiability Problems in Combinatorial Switching Circuits*, Ph.D. Thesis, Dept. of EECS, U. Michigan, 1995.

[20] D. Spielman and S.-H. Teng. Smoothed Analysis: Why the Simplex Algorithm Usually Takes Polynomial Time, *Proc. STOC*, 296–305, 2001.