Search Tree Restructuring

Erik Zawadzki and Tuomas Sandholm

May 20, 2010
CMU-CS-10-102

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work supported by the National Science Foundation under grant IIS-0905390.
Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Keywords: tree search, integer programming, branching, variable ordering

Abstract

Poor branching decisions in a search tree can increase solve time by orders of magnitude. We introduce a new approach where, instead of committing to the tree so far, we restructure the tree throughout the search. This has the advantages of removing unnecessary decisions from the tree and relocating important decisions closer to the root. We define two tree-modifying operators: delete and promote, and prove that they maintain correctness and completeness. We show that conflict-directed backtracking (CDBT) can be expressed as a compound operation of the two; however it is only one member of a rich space of compound tree operations. We study a more general and aggressive compound operation which we coin *path contraction*. We develop several policies for applying that operation. We prove that under a natural assumption, one of the policies (and CDBT) is never harmful. In practice we show that policies that apply path contraction more aggressively are even better. The technique applies to both optimization and constraint satisfaction. We present binary integer programming experiments on unsatisfiable graph coloring and 3SAT. Our techniques yield significant speed improvements and reduce node count by up to 82%.

1 Introduction

The solve time of tree search can vary by orders of magnitude depending on the branching decisions. There has been significant experimental work on branching heuristics, but there is no universally good way of selecting branching decisions, much less a theory for doing so. In practice, tree search algorithms can generate huge trees even on problem instances that have a short tree certificate of unsatisfiability (or optimality in the optimization context). In fact, Liberatore [2000] proved that finding a branching decision (for the DPLL SAT algorithm) yielding a tree that is at most twice the minimum size is, itself, NP-hard.

Branching is so important because tree search is a core reasoning technique in AI and in integer programming in operations research. Integer programming has in the last few years found its way into AI applications directly, ranging from combinatorial auctions to machine learning (*e.g.*, clustering) to Bayesian reasoning. Tree search in AI and integer programming really constitute one body of techniques. In integer programming, linear programming is used as the propagator at every search node. Integer programming is one of the most-used forms of tree search, and specifically the most basic and perhaps most prevalent form of integer programs are *binary integer programs (BIPs)*:

$$\max [cx + dy : Ax + By \leq b, \quad y \in \{0,1\}^m] .$$

We will present our results in this context, but the techniques apply to other tree search settings as well.

The most frequently used method for solving these problems is branch-and-bound search. As it builds the search tree, every new node is evaluated by solving the linear relaxation of the remaining problem at that node, *i.e.*, the linear programming (LP) problem left if all the binary constraints of the remaining problem $y \in \{0,1\}^m$ are relaxed to $y \in [0,1]^m$. The LP can be solved, for example, with the simplex algorithm. If, after solving the LP, any of the binary variables are fractional in the optimal LP solution, then one of these variables is chosen to be branched on. A search path in the tree is terminated (we call it *fathomed*) when the node's LP is infeasible or integral, or the LP objective value is no better than the value of the incumbent (best solution found so far). The answer to the problem is either an optimal solution or the assertion that no feasible solution exists. In both cases, the search tree constitutes a proof (certificate) of the answer.

How can we deal with the problem of bad branching decisions? One approach is simply to try to find better branching decisions (see, *e.g.*, Achterberg et al. [2005], Gilpin and Sandholm [2007], and Achterberg and Berthold [2009]).

Another technique is to use conflict analysis information to determine which decisions were essential to a subproblem's fathoming. Modern descendants of DPLL, like GRASP [Marques-Silva and Sakallah, 1999], analyze infeasible partial assignments and generate *conflict clauses*: partial assignments that generate infeasibility. These conflict clauses (aka *nogoods*) are used to 1) curtail the search space by adding implied clauses into the clause database and 2) prune the search with *conflict-directed backtracking (CDBT)*.

Conflict analysis has also been applied to CSPs. For example, *dynamic backtracking* [Ginsberg, 1993, Ginsberg and McAllester, 1994] is a complete search for CSPs that tries to preserve the elimination of as many values as possible from the domains of decision variables after a backtrack. This approach is similar to our own, but we can preserve entire subtrees rather than just eliminated

values. Additionally, dynamic backtracking can *forget* particular nogoods, whereas we do not consider forgetful operators in this stage of our work. Additionally, Katsirelos and Bacchus [2005] show that using a generalization of nogoods can lead to significant performance improvements for CSP solvers. Furthermore, both Sandholm and Shields [2006] and Achterberg [2007] have shown that integrating conflict analysis into integer program solvers can improve performance.

Restarting the search repeatedly sometimes helps reduce the harm from unfortunate branching decisions. Restarting policies vary in sophistication. Gomes et al. [1998] hand tuned their policy. Later approaches simultaneously learn about run-time distributions and follow restart policies that exploit them [Kautz et al., 2002]. Learning is not limited to collecting information about distributions: modern SAT solvers [Moskewicz et al., 2001, Goldberg and Novikov, 2007, Huang, 2007], CSP solvers [Lecoutre et al., 2007], and some BIP solvers [Kilinc-Karzan et al., 2009] combine restarting with constraint learning procedures that store nogoods between restarts. Solvers also use conflict information to guide their branching heuristics (see, for example, Moskewicz et al. [2001]). However, even with constraint learning, some information is generally lost and the search tree discarded after each restart. Furthermore, restarts have mainly been shown to improve performance on CSP and SAT instances, while they have been shown not to help if the underlying solver uses LP-based upper bounding. This was shown on several classes of combinatorial auction winner determination problems (without constraint learning) [Sandholm et al., 2005].

We explore a new approach to the branching problem: we restructure the tree throughout the search to remove bad branch decisions and to move important ones closer to the root. The motivation is to be able to re-decide what questions we branch on, and where in the tree, without losing information or tree structure. We show the promise of this idea even when using LP-based upper bounding and constraint learning.

2 Search Tree Restructuring Techniques

We first introduce two operators for restructuring a search tree.[1] We require that they preserve correctness and completeness. An operator preserves correctness if it only prunes parts of the search space in the transformed tree that were pruned in the original tree, *i.e.*, it does not prune any new part of the search space. An operator preserves completeness if we never erase any of our search progress, *i.e.*, never lose any fathoming information by using the operator. We additionally require that a completeness-preserving operator never introduces any pointless rebranching on any decision variable (no tree-path every connects two instances of the same decision variable). We call any such pointless rebranching a *double branch*. This requirement makes it clear that all searched nodes are distinct and so solving a subproblem always makes progress. For each fathomed node, we define the *sufficient reason for fathoming (SRF)* to be the nogood that the search algorithm knows is the reason for fathoming. In search algorithms that do not use conflict analysis, this is simply the set of assignments on the current search path, and in conflict analysis it is the identified nogood.

Definition 1. *An operator F is* correctness preserving *if, for every tree T, the partial assignment for every fathomed node in $F(T)$ is a superset of some SRF of a node in T. Furthermore, the SRF*

[1] In this stage of our work, we are explicitly representing the tree as a linked structure.

for every fathomed node in $F(T)$ is a superset of the SRF of some node in T.

Definition 2. *An operator F is* completeness preserving *if, for every tree T, there does not exist a full assignment that was fathomed in T but unfathomed in $F(T)$. Additionally, if there are no paths in T that double branch then there are no double branches in $F(T)$.*

2.1 Delete Operator

We call our first tree operator *delete* (Fig. 1). It causes the deletion of an irrelevant decisions and one of its subtrees. If a fathomed node f has an SRF that does not involve its parent decision, then we can delete the parent decision and f's sibling subtree U. Since f can be fathomed without assigning the parent's literal, we can safely prune U. This operation can be carried out in constant time by simply rearranging pointers.

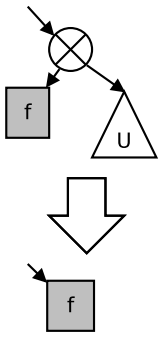

Figure 1: The delete *operation. f is a fathomed node that does not involve the exed variable and U is a subtree.*

Proposition 1. *The delete operator preserves completeness and correctness.*

Proof. Let f be the fathomed node in figure 1. The delete operator can be seen as correctness preserving immediately: since the parent never appeared in f's SRF, the partial assignment to f after the operation is still a superset of the SRF.

Delete preserves completeness as well. Let χ denote the partial assignment made to reach the crossed-out node in figure 1. The only complete assignments affected by this operator are in the subtree that extends χ. Every complete assignment that extends χ is fathomed after the operation, so every complete assignment fathomed before the deletion is also fathomed after the deletion. Since delete has only ever removed decisions, it cannot have introduced any double branches. Therefore it preserves completeness. □

2.2 Promote Operator

The *promote* operator—shown in figure 2—takes a branching decision and its parent, and switches their positions in the search tree. Their subtrees are then copied to repair the tree. This operation is linear in the size of the subtree that it is copying, so can be exponential in the size of the input if the subtree is large. We will later give some policies for applying these operators that heuristically control when these operators are applied, so as to avoid blowup.

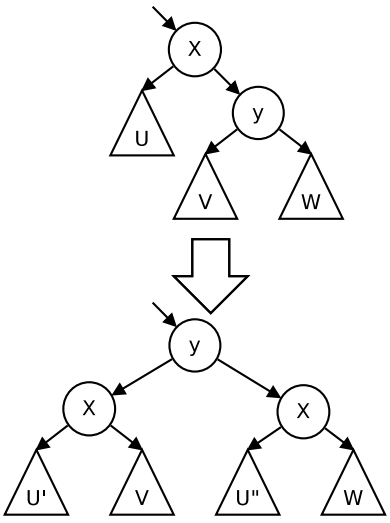

Figure 2: The promote *operation.*

Now the promoted variable (y in the example) may occur twice on some paths (once at the node marked y and again in the subtrees U' and U''). We thus conduct what we coin *double branch pruning*: at all places in U' and U'' where y occurs, we remove y as follows. We remove the search node y and bring up the subtree rooted at that child of y that agrees with the assignment made in y's first occurrence. The subtree rooted at y's other child is deleted because it is now inconsistent with the assignment made to y at y's first occurrence. Because this double branch pruning could involve searching the entire subtree, this operation is linear in the size of the subtree. So, in principle it could be exponential in the size of the input, but as mentioned above, our policies described later will curtail this blowup.

We denote the pruned trees by $U \setminus \{y = 0\}$ (which is U, except that all $y = 0$ subtrees have been pruned) and $U \setminus \{y = 1\}$. We denote by T the subtree whose root changes in the promote operation. We denote by $\mathbb{T}$ the entire search tree.

Proposition 2. *The promote operator preserves completeness and correctness.*

Proof. **Correctness:** Consider a fathomed node $f \in \mathbb{U}$, where $\mathbb{U}$ is the new tree. Since each f belongs to a subtree that is a copy of a subtree in $\mathbb{T}$, it has an associated node $f' \in \mathbb{T}$. Then f must be in one of three cases:

- $f \notin T$: the only changes made to the search tree are local to the subtree T, so the path to f is the same as the path to f';
- $f \in V$ or $f \in W$: Without loss of generality, let $f \in V$. The partial assignment at the root of the copy of V in $\mathbb{U}$ is identical to original V in $\mathbb{T}$, so the partial assignment at f is the same as the partial assignment to f';
- $f \in U \setminus y = 0$ or $f \in U \setminus y = 1$: partial assignment at the root of either copy of U in $\mathbb{U}$ is a (proper) superset of the assignments made in $\mathbb{T}$. U itself is unchanged, so the partial assignment at f is a superset of the assignment at f'.

Since in every case the partial assignment at f is a superset of the assignment at f', and the SRF for f' is a subset of the assignment at f', the assignment at f must be a superset of the SRF for f'. Additionally, the SRF for f is identical to the SRF for f'. Therefore, promote preserves correctness.

Completeness: Let μ be the partial assignment made to reach the root of U in T. Let ν be the partial assignment for the root of V and ω be the root of W. For every full assignment, ϕ, fathomed in T, there are two possible cases:

- If ϕ extends ν or ω, it must extend a fathomed node in either V or W, and since the trees V and W were copied exactly, this full assignment is fathomed after promotion too.
- If ϕ extends μ then, without loss of generality, it assigns $y = 0$ and must extend a fathomed partial node z in U with partial assignment ζ. Therefore, ϕ extends $\zeta \cup \{y = 0\}$.
  - If the partial assignment to z does not involve y, it is unaffected by double branch pruning, and is copied to both $U \setminus y = 0$ and $U \setminus y = 1$. Therefore $\zeta \cup \{y = 0\}$ is a copy of the fathomed node z in $U \setminus y = 1$, and so ϕ is fathomed.
  - If the partial assignment to z does involve y, then $\zeta = \zeta \cup \{y = 0\}$ and z is copied to $U \setminus \{y = 1\}$ (it will be removed by double branch pruning in $U \setminus \{y = 0\}$). Then $\zeta \cup \{y = 0\}$ is the copy of z in $U \setminus \{y = 1\}$, so ϕ is fathomed.

Promote does not introduce any double branches because any double branches are explicitly pruned. Thus promote preserves completeness. □

While promote duplicates one of the subtrees, U, and thus potentially makes the search tree larger, it adds an assignment before these copies. Intuitively, we expect this operation could be beneficial whenever at least one literal of y is important to fathoming decisions for U.

2.3 Conflict-Directed Backtracking (CDBT)

CDBT can now be expressed as a compound of delete and promote operations. Consider a newly discovered fathomed node f, and imagine that we have discovered (using some conflict analysis procedure) that there exists a conflict that is a proper subset of the partial assignment on the path. There are three general cases: 1) the parent of f is not in the SRF, 2) the parent of f is in the SRF, but the grandparent is not, or 3) both the parent and grandparent are in the SRF.

In the first case, we can repeatedly apply the delete operator until we are in the second or third case. This straightforwardly mimics the non-chronological backtracking phase of CDBT.

In the second case, we can promote the parent decision. Then, since the old grandparent decision is now f's parent decision, and does not belong to f's newly discovered SRF, it can be deleted. This two-step process can be repeated until we are in the third case. This phase of the procedure operates just like *failure-driven assertion (FDA)* in CDBT [Marques-Silva and Sakallah, 1999]: at the end of this phase the parent branching decision will be placed immediately below the second-deepest variable in the SRF with the fathomed node (f) blocking off one of its two branches. This makes the parent decision an assertion.

In the third case, we do nothing.

Now we can try to understand when CDBT is helpful. To answer this, we introduce the concept of *assignment monotonicity*. It means the remaining node count of the underlying solver (on top of which we do tree restructuring) decreases as more variable assignments are made.

Definition 3. *A (potentially randomized) tree search algorithm A on problem instance I is* assignment monotonic *if, for every subproblem S (that arises), adding any assignment to S (weakly) decreases expected (over randomizations) number of remaining search nodes needed to finish the tree rooted at S.*

We show in an appendix that the search algorithm presented in this paper seems to be assignment monotonic on our problem instances used in our evaluation.

If the underlying solver (*i.e.*, some choice of branching heuristic, node selection heuristic, cutting plane generator, etc.) is assignment monotonic on the problem instance, then CDBT is always a good idea:

Proposition 3. *Assuming that the underlying solver is assignment monotonic on the given problem instance, CDBT weakly decreases the expected number of remaining search nodes.*

Proof. CDBT weakly decreases the amount of remaining work since every subtree, except the fathomed node f, is weakly deeper after the CDBT operation. □

2.4 Path Contraction

We showed that CDBT can be expressed as a compound operation of interleaved promote and delete operations. However CDBT is only one member of a rich space of compounds of these two. In this section we propose a more aggressive compound, *path contraction*, which generalizes CDBT.

Again, consider a newly discovered fathomed node f and its associated SRF. We mark each decision node on the path from the root to f to indicate whether or not it is a member of the SRF. This partitions the path into alternating *relevant segments* and *irrelevant segments*. The decisions in a relevant segment are in the SRF, while the decisions in an irrelevant segment are not. The segment between the fathomed node and the first relevant decision is the 1^{st} irrelevant segment (it will be empty when the parent of f is in f's SRF), the subsequent relevant segment is the 1^{st} relevant segment, the next irrelevant segment is the 2^{nd} irrelevant segment, and so on. Figure 3 gives an example of an annotated path.

We can eliminate the 1^{st} irrelevant segment by deletion operations alone. The 2^{nd} irrelevant segment can be eliminated by a mixture of promote and deletion operations as in our expression of CDBT. However, unlike in CDBT, we can eliminate the 2^{nd} irrelevant segment even when the 1^{st} relevant segment has more than one decision. To do this we promote members of the 1^{st} relevant segment until the lowest member of the 2^{nd} irrelevant segment is the parent of the fathomed node f, and then we apply the delete operator (Fig. 4). This decreases the length of the 2^{nd} irrelevant segment by one. If the size of the 2^{nd} irrelevant segment becomes zero, then the 1^{st} and 2^{nd} relevant segments fuse to form a new 1^{st} relevant segment, and the old 3^{rd} irrelevant segment becomes the new 2^{nd} irrelevant segment. This process can be repeated as long as there are irrelevant segments between the root and f.

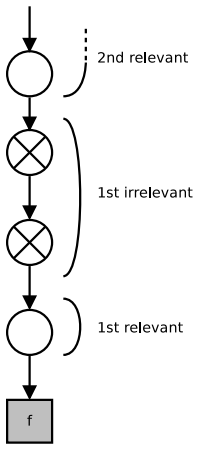

Figure 3: An example of a path with three annotated segments.

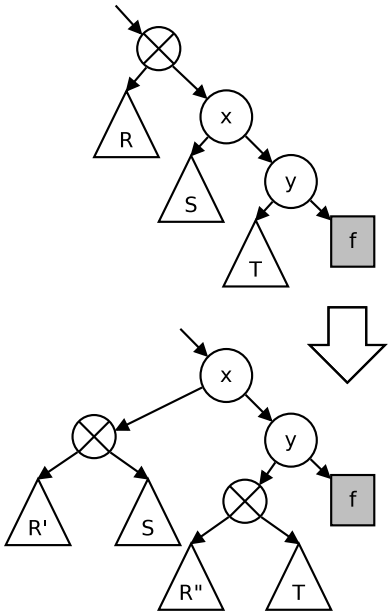

Figure 4: The path contraction operation. Here, $R' = R \setminus \{x = 1\}$ and $R'' = R \setminus \{x = 0, y = 1\}$ are pruned copies of R.

2.4.1 Basic Path Contraction

While CDBT is uniquely defined for a newly discovered SRF, there are a host of path contractions that apply. For example, there is a choice of how close to the root we wish to contract up to. Therefore, we need to introduce contraction policies to govern which path contractions we will take. Our *basic path contraction policy* contracts up to the n^{th} irrelevant segment, but stops before any relevant segment that has more than k decisions. (CDBT is then identical to a basic path contraction policy with $n = 2$ and $k = 1$.) The idea of this policy is to restrict, in a relatively static way, how aggressively we apply path contractions, thus limiting how many times each subtree is copied.

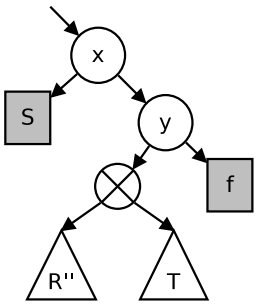

Figure 5: The result of single-copy contraction (SCC).

Path contractions allowed by CDBT are usually (and always under assignment monotonicity) helpful and thus it seems reasonable to take them whenever available. Not every (even basic) path contraction policy will be beneficial: some can actually increase remaining node count.

2.4.2 Single-Copy Contraction (SCC)

We now prove that there are policies beyond CDBT that guarantee a decrease in the number of expected nodes (under assignment monotonicity). One such policy—which we coin *single-copy contraction (SCC)*—demotes down an irrelevant decision d through a relevant segment if and only if we can show that of all the copies of d's off-path subtrees (R, S, and T in figure 4) at most one will remain after subsequent deletions.

For example in figure 4, SCC will only eliminate the exed decision if it can show that the off-path subtree, R, will be copied at most once. In this example, this will amount to either S or T being a closed leaf with an SRF that does not involve the exed variable. Figure 5 shows what happens to the top tree of figure 4 if S is such a fathomed leaf.

SCC always places every subtree of the original tree weakly deeper in the new search tree. The following result follows immediately. Note that this means that SCC can be shown to cause no harm under the same condition (assignment monotonicity) as CDBT despite SCC being a generalization of CDBT.

Proposition 4. *Assuming that the underlying solver is assignment monotonic on the given problem instance, SCC weakly decreases the expected number of remaining search nodes.*

3 Experiments

Our experiments compare *None* (a policy that rejects all path contractions), *CDBT*, *SCC*, *SCC+*, and two basic policies—*Basic1* ($n = 3$ and $k = 1$) and the more aggressively tuned *Basic2* ($n = 6$ and $k = 2$). The two basic policies were two of the most successful parameter settings from an initial round of tuning n and k, but further automated or manual tuning would certainly be an interesting direction for future research. *SCC+* is like *SCC*, except that it is more aggressive. It does not count a copy toward the one-copy limit if a new assignment has been made to the copy that is contained in over 20% of the nogoods in the entire tree. *SCC+* thus encourages moving those branching decisions up that occur frequently in nogoods.

We benchmark on infeasible instances because they are essentially equivalent to optimization—where one needs to prove that a better solution does not exist—and because they are often a more

difficult class of CSPs. Also, in the integer programming community a folk wisdom has recently begun to emerge that finding good feasible solutions and proving optimality are tasks that call for very different types of algorithms. The two types of algorithms can be run in parallel (or time sliced) to tackle both tasks on any given problem instance. Such approaches have also recently emerged in AI (e.g. [Kroc et al., 2009]).

In order to keep our experiments as free as possible from confounding factors, our implementation of branch-and-bound is simple. Our solver branches using the most well-known [Nemhauser and Wolsey, 1999] variable-selection method in integer programming: it branches on the variable whose LP value is most fractional. Random tie breaking induces a performance distribution over runs on any given instance. As our node selection heuristic, we select the left-most open node in the tree, where the 'left-branch' of a decision is the assignment of value zero. While node ordering is important for solver performance due to fathoming by bound, it is less important in our case because the instances are infeasible, so there are no feasible incumbents and thus no fathoming by bound. We wrote our own branch-and-bound code in C++ and used CPLEX 10.0.1 to solve the LPs at the search nodes.

Whenever our solver encounters a fathomed node, it uses a duality-based conflict analysis procedure[2] to find an SRF and applies a set-covering cut (an additional linear inequality constraint) that encodes the nogood into the integer program [Achterberg, 2007]. These are the only cuts we add.

We report node count as an indication of the performance of each policy. Node count is a reasonable metric for testing performance of first-cut implementations. We also report timing results. All experiments were run on two dual quad-core AMD 2376 CPU machines (2.GHz, 4×512KB L2 Cache) with 32GB memory running Linux (2.6.18). Run time is capped at 10 minutes. To check the significance of the difference between empirical node-count and run-time distributions of two solvers run on a problem instance, we use the one-sided Kolmogorov-Smirnov goodness-of-fit test with $\alpha = 0.05$.

3.1 Infeasible Coloring

We first experimented on random graph 4-coloring problems parameterized by N, the number of vertexes, and e, the expected number of neighbors for each vertex. To ensure infeasibility, we inserted a random 5-clique in every problem instance. We generated five instances from each of nine subclasses. Each subclass is either *small* (50 vertexes), *medium* (75 vertexes), or *large* (100 vertexes) and is either *sparse* ($e = 2.5$), *moderate* ($e = 3$), or *dense* ($e = 3.5$).

We encoded each instance as a BIP using binary variables to indicate a vertex having a particular color. Inequalities are generated to ensure that no adjacent vertexes can have the same color and that every vertex has at least one color. The objective function awards unit value for every successfully colored variable. Each of the 45 instances was run 25 times for each of the algorithms.

Table 1 shows the number of problem instances for each subclass where there was no other solver that was significantly better. The results show that no solver is significantly better than

[2]Notice that this duality-based approach to conflict analysis is distinct from a conflict graph based approach (e.g., Marques-Silva and Sakallah [1999]).

Basic2 on any problem instance. Table 2 shows that *CDBT* and all four path contraction policies offered a significant advantage over *None* on most instances, and that *None* never offered a significant advantage over any of the other policies.

	None	**CDBT**	**SCC**	**SCC+**	**Basic1**	**Basic2**
color_L_dense	·	·	·	·	·	**5**
color_L_mod	·	·	·	1	·	**5**
color_L_sparse	·	·	·	1	1	**5**
color_M_dense	·	·	·	·	·	**5**
color_M_mod	·	·	·	·	·	**5**
color_M_sparse	·	1	1	1	1	**5**
color_S_dense	·	·	·	·	·	**5**
color_S_mod	1	2	2	2	2	**5**
color_S_sparse	·	1	2	4	3	**5**

Table 1: Number of coloring instances for which there did not exist another policy that had significantly better node counts. Higher numbers are better.

	None	**CDBT**	**SCC**	**SCC+**	**Basic1**	**Basic2**
None	·	·	·	·	·	·
CDBT	39	·	·	1	·	·
SCC	41	13	·	1	·	·
SCC+	41	32	26	·	10	·
Basic1	41	31	23	4	·	·
Basic2	**44**	**40**	**40**	**36**	**38**	·

Table 2: Number of coloring instances for which the row policy had significantly better node counts than the column.

Performance improvements are particularly impressive on some instances. *Basic2* was able to reduce median node counts by up to 82%. *Basic2* reduced the medians by at least 50% for 17 of the 45 problem instances. *Basic1* was able to do this on 12 instances, *SCC+* on 11, *SCC* on 4, and *CDBT* on 2. *Basic2* was the only policy able to decrease the median by over 70% and did so on 6 instances. No policy increased the median.

Basic2 timed out least frequently—on only 2.0% of runs. *Basic1* timed out on 6.7%, *SCC* on 7.4%, *SCC+* on 7.8%, *CDBT* on 10.7%, and *None* on 15.5%. Even though our code was not optimized for run time, *Basic2* was able to reduce median run time by up to 93.2%—it reduced run time by more than 80% on 9 instances (Tables 3 and 4). *Basic1* also reduced run time by more than 80% on 9 instances, *SCC+* on 7, *SCC* on 3, and *CDBT* on 2. *SCC* and *Basic2* each significantly increased run time once (on the same instance), but the increase was less than 2.1%.

In summary, on the coloring problems, *Basic2* was the best of the tested policies overall, and yielded large improvements. This provides strong justification for our approach.

	None	CDBT	SCC	SCC+	Basic1	Basic2
color_L_dense	·	2	1	2	1	**5**
color_L_mod	·	·	·	2	1	**5**
color_L_sparse	·	·	·	2	2	**5**
color_M_dense	·	·	·	1	1	**5**
color_M_mod	·	·	·	2	1	**4**
color_M_sparse	·	1	·	2	1	**5**
color_S_dense	·	1	1	1	1	**4**
color_S_mod	1	2	·	3	**4**	2
color_S_sparse	·	4	3	**5**	**5**	4

Table 3: Number of coloring instances for which there did not exist another policy had significantly better run time. Higher numbers are better.

	None	CDBT	SCC	SCC+	Basic1	Basic2
None	·	·	1	·	·	1
CDBT	39	·	6	1	1	3
SCC	40	7	·	·	·	·
SCC+	41	27	31	·	7	1
Basic1	41	30	32	8	·	**5**
Basic2	**43**	**34**	**34**	**24**	**25**	·

Table 4: Number of coloring instances for which the row policy had significantly better run time than the column.

3.2 Random Unsatisfiable CNF

We also experimented on random 3SAT CNF problems parameterized by V, the number of variables and C, the number of clauses. There are three subclasses: `unsat0*` has $V = 100$ and $C = 430$, `unsat1*` has $V = 100$ and $C = 400$, and `unsat2*` has $V = 100$ and $C = 370$. We start building each instance by inserting a small randomly-generated infeasible kernel to the formula. The remaining clauses are picked randomly from the set of literals, rejecting any that are duplicates.

We encode each instance as a BIP using the standard formulation: there is a one-to-one mapping of binary variables in the BIP and in the UNSAT problem and a set-covering constraint to ensure that every clause is satisfied. We use an objective value that awards unit value for every positive literal in the assignment. We ran each of the 30 instances 50 times for each of the algorithms.

Tables 5 and 6 summarize the node-count results. The conclusions are less clear than in coloring, but *Basic1* is now the most successful of the policies tested. There was a significantly better policy on only three instances—in all three cases *Basic2* was the better algorithm. *Basic2* had extremely variable performance, and *None* was significantly better than it on 18 instances. *Basic2*'s performance on these 18 instances can be attributed to timing out—*Basic2* timed out on 25.5% of its runs and was the only policy to time out.

	None	**CDBT**	**SCC**	**SCC+**	**Basic1**	**Basic2**
unsat0*	3	5	6	6	**10**	2
unsat1*	3	4	5	**7**	**7**	5
unsat2*	·	2	2	3	**10**	4

Table 5: Number of 3SAT instances for which there did not exist another policy that had significantly better node counts. Higher numbers are better.

	None	**CDBT**	**SCC**	**SCC+**	**Basic1**	**Basic2**
None	·	·	·	·	·	18
CDBT	3	·	·	·	·	18
SCC	4	·	·	·	·	18
SCC+	10	7	2	·	·	18
Basic1	**23**	14	**14**	**12**	·	**19**
Basic2	17	**15**	**14**	8	**3**	·

Table 6: Number of 3SAT instances for which the row policy had significantly better node counts than the column.

Basic2 was the only policy to significantly increase median run time (Tables 7 and 8), and it increased the median by over 10% on 16 of the 30 instances. *Basic1* was the most successful policy in significantly decreasing run time, and it decreased the median significantly on 20 instances. *SCC+* decrease on 10, *Basic2* on 7, *SCC* on 4, and *CDBT* on 4. The median decreases were large on some instances: *Basic1* decreased the median by over 50% on 13 instances, *Basic2* on 7, *SCC+* on 3, *SCC* on 1, and *CDBJ* on none.

	None	CDBT	SCC	SCC+	Basic1	Basic2
unsat0*	5	6	6	7	**10**	1
unsat1*	4	4	7	8	**9**	2
unsat2*	·	3	3	4	**10**	1

Table 7: Number of 3SAT instances for which there did not exist another policy that had significantly better run time. Higher numbers are better.

	None	CDBT	SCC	SCC+	Basic1	Basic2
None	·	·	·	·	·	23
CDBT	4	·	·	·	·	23
SCC	4	·	·	·	·	24
SCC+	10	3	3	·	·	24
Basic1	**20**	**14**	**13**	**9**	·	**26**
Basic2	7	7	3	3	**1**	·

Table 8: Number of 3SAT instances for which the row policy had significantly better run time than the column.

However, on runs where *Basic2* did not time out, it tended to be very successful at reducing median node count. Of the 30 instances, *Basic2* reduced median node count by over 50% on 10 instances. *Basic1* managed this on 7 instances, *SCC+* on 3, and both *SCC* and *CDBT* on none. If we drop the threshold to 25% reduction, *Basic1* improved performance on 21 instances, *Basic1* on 14, *SCC+* on 12, *SCC* on 8, and *CDBT* on 7. In summary, on the 3SAT problems, *Basic1* frequently yielded modest improvements while *Basic2* sometimes yielded large improvements.

4 Conclusions and Future Research

We introduced two operators—delete and promote—for restructuring a search tree during search. They can be combined arbitrarily while maintaining correctness and completeness. This perspective opens a new space of possibilities, and conflict-directed backtracking (CDBT) is just a point in this space. We introduced another family of methods in this space, path contraction. We proved that (under assignment monotonicity) certain forms of path contraction always reduce search tree size as we showed does CDBT. Experimentally, more aggressive path contraction performs even better. Our techniques yield significant speed improvements and reduce node count by up to 82%.

Future research includes studying different policies for path contraction, and other methods in our space, on a wider variety of problems. We studied some policies, but the space is rich and undoubtedly there are better algorithms that make more complete use of the conflict analysis information. This space seems fertile also for automated algorithm design and parameter tuning. Additionally, it would be interesting to identify what instance properties lead to good path contraction performance.

There are further correct operators that could be developed, for example *forgetting*, an information-losing operation where we delete a subtree (for example, in order to avoid increasing overall current tree size as a result of a promote operation). Forgetting is essentially a local restart. This operator

helps avoid moving around large subtrees with few fathoming conclusions. A information-losing operation is necessary to generalize approaches like dynamic backtracking [Ginsberg, 1993, Ginsberg and McAllester, 1994] and may be important for efficient restructuring policies.

Finally, what guarantees can be made about these operations and policies? Could we bound, for example, the amortized time spent on path contractions? If we incorporate lossy operators, like forgetting, what policies are efficient and still ensure that search is complete?

Hopefully, research in these directions will stimulate work on other forms of tree reorganization as well.

References

T. Achterberg and T. Berthold. Hybrid Branching. In *CPAIOR*, pages 311–312, 2009.

Tobias Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1): 4–20, March 2007.

Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, January 2005.

Andrew Gilpin and Tuomas Sandholm. Information-theoretic approaches to branching in search. In *IJCAI*, pages 2286–2292, Hyderabad, India, 2007.

M.L. Ginsberg. Dynamic Backtracking. *JAIR*, 1:25–46, 1993.

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

E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

C.P. Gomes, B. Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *AAAI*, Madison, WI, 1998.

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

George Katsirelos and Fahiem Bacchus. Generalized Nogoods in CSPs. In *AAAI*, Pittsburgh, PA, 2005.

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

F. Kilinc-Karzan, G. Nemhauser, and M. Savelsbergh. Information-Based Branching Schemes for Binary Linear Mixed Integer Problems. Draft, `http://www.optimization-online.org`, 2009.

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

C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Nogood recording from restarts. In *Proceedings of IJCAI*, volume 7, pages 131–136, 2007.

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

J P Marques-Silva and K A Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, 2001.

George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1999.

Tuomas Sandholm and Robert Shields. Nogood learning for mixed integer programming. Technical Report CMU-CS-06-155, Carnegie Mellon University, 2006.

Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. CABOB: A fast optimal algorithm for winner determination in combinatorial auctions. *Management Science*, 51(3): 374–390, 2005. Early version in IJCAI-01.

A Appendix: Assignment Monotonicity

One of the fundamental assumptions for those theoretical results in this paper that show that some of our policies never hurt (Propositions 3 and 4) is that the underlying solver (*i.e.* the solver without any path contraction) is *assignment monotonic*. In other words, assigning any additional literal to a problem instance weakly decreases the number of expected nodes left in the search. This assumption is intuitively appealing since assigning an additional literal weakly decreases the set of feasible full assignments.

However, we may worry that while the linear program relaxation polytope after assignments contains weakly fewer feasible points, the structure of the polytope has also changed which may be 'misleading' to our branching heuristics and node selection heuristics. For example, assigning an additional literal may turn a problem instance that can be pruned at the root into a instance that requires non-trivial search. As an example of this, consider the 2-dimensional IP

$$\max\left[2x_1 + x_2 \mid 2x_1 + x_2 \leq 1, x_1, x_2 \in \{0, 1\}\right].$$

There is an integer solution to the root relaxation (namely $(0, 1)$ but the LP after assigning $x_2 = 0$ is fractional: the optimal solution is $\left(\frac{1}{2}, 0\right)$. Therefore, any solver that prunes integer solutions can display non-monotonic behavior.

We can construct solvers that are assignment monotonic by removing all heuristics and pruning. Explicitly enumerating and checking all potential complete assignments is clearly assignment monotonic since asserting literals weakly decreases the number of complete assignments.

While this shows that there *exist* monotonic solvers, enumeration is not a practical solver. Do practical solvers exhibit assignment monotonicity? In particular, what about the underlying solver used in this study—is it monotonic? We ran some experiments to see whether this hypothesis is tenable.

A.1 Assignment Monotonicity Experiment with the Most-Fractional-Variable (MFV) Branching Rule

In this experiment we checked the solver used in the other experiments of this paper for assignment monotonic behavior. It uses depth-first search, an LP oracle to indicate which variables are fractional, and applies conflict-analysis-based generalized upper bound cuts to prune parts of the tree that conflict-analysis proves to be infeasible. It uses the *most-fractional variable* (MFV) heuristic for branching.

To conduct the assignment monotonicity experiment we first conducted 150 runs on 16 different problem instances to get a baseline node-count distribution. Then we ran an addition 150 runs for each instance where we assigned an additional literal (selected uniformly from all possible literals) to obtain a modified distribution.

If a solver is assignment monotonic on a particular problem instance, then we would see a significant negative correlation between the node count and assigning an additional random literal. (This is a necessary empirical condition for assignment monotonicity, not a sufficient one.) To test for correlation we use Spearman's ρ-test with a significance level of $\alpha = 0.05$.

There is evidence that the MFV solver is assignment monotonic on 7 of the 16 problem instances. Indeed, every instance that displayed significant correlation had $\rho < 0$. These results

are summarized in table 9. Some of the correlation was very strong, such as *color_M_mod_01* (figure 6). Notice that the modified distribution is close to stochastically dominating the baseline distribution; this is strong evidence for the MFV solver being assignment monotonic on this instance.

	ρ	p **-value**
color_L_mod_00	$-0.17^{\dagger}$	0.0025
color_L_mode_01	$-0.13^{\dagger}$	0.022
color_M_mod_02	$-0.17^{\dagger}$	0.0033
color_M_dense_00	$-0.14^{\dagger}$	0.017
color_M_dense_01	-0.1	0.082
color_M_dense_02	$-0.13^{\dagger}$	0.024
color_M_mod_00	-0.093	0.11
color_M_mod_01	$-0.4^{\dagger}$	$8.4e-13$
color_M_mod_02	0.051	0.38
unsat00	0.099	0.088
unsat01	-0.033	0.57
unsat02	-0.095	0.1
unsat03	-0.023	0.69
unsat04	$-0.16^{\dagger}$	0.0067
unsat05	-0.017	0.77
unsat06	-0.092	0.11
unsat07	-0.052	0.37

Table 9: Correlation between setting a random literal and node count for the MFV solver. Significant results are noted with a †.

A.2 Assignment Monotonicity Experiment with Strong Branching

While the MFV solver turned out to exhibit assignment monotonicity, in this section we show that, surprisingly, the same solver with another kind of popular branching rule, *strong branching*, often does not, depending on the instance. In strong branching, the solver does a 1-step lookahead for each variable and then selects the variable to branch on that reduces the average bound at the two children the most. (We run the LP solver for 25 iterations (pivots) at each of the two children. Such capping of the number of iterations is a common technique used in strong branching so the lookahead does not become too costly.)

To conduct the assignment monotonicity experiment we again first conducted 150 runs on the 16 different problem instances to get a baseline node-count distribution. Then we ran an addition 150 runs for each instance where we assigned an additional literal (selected uniformly from all possible literals) to obtain a modified distribution.

There is evidence for the strong-branching solver being assignment monotonic on some problem instances but not others (Tab. 10). There is significant correlation in 10 of the 16 instances. Correlation is negative in 5 of these, and positive in the other 5. Some of the positive correlation is quite high, for example, in instance *unsat04*.

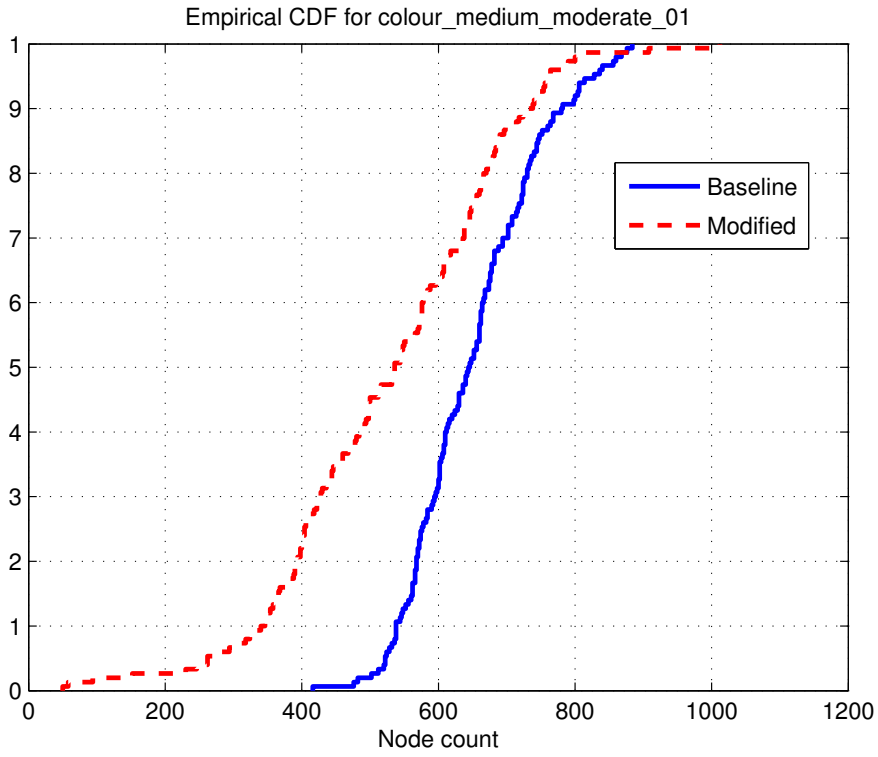

Figure 6: Node-count distribution for MFV on *color_M_mod_01*. The Baseline distribution is the solid curve, and the modified distribution is the dashed curve.

	ρ	p **-value**
color_L_mod_00	-0.058	0.32
color_L_mod_01	-0.082	0.16
color_L_mod_02	-0.1	0.082
color_M_dense_00	-0.1	0.082
color_M_dense_01	-0.058	0.32
color_M_dense_02	$-0.17^{\dagger}$	0.004
color_M_mod_00	-0.1	0.082
color_M_mod_01	-0.082	0.16
color_M_mod_02	$-0.13^{\dagger}$	0.024
unsat00	$0.84^{\dagger}$	$5.4e-80$
unsat01	$-0.14^{\dagger}$	0.013
unsat02	$0.26^{\dagger}$	$5.8e-06$
unsat03	$-0.81^{\dagger}$	$5.7e-71$
unsat04	$0.68^{\dagger}$	0
unsat05	$-0.31^{\dagger}$	$4.8e-08$
unsat06	$0.18^{\dagger}$	0.0018
unsat07	$0.44^{\dagger}$	$5.6e-16$

Table 10: Correlation between setting a random literal and node count for the strong-branching solver. Significant results are noted with a †.

Figure 7 shows the node-count distribution for *unsat04*. This shows that the vast majority of randomly assigned literal dramatically increase the amount of remaining nodes in the search tree. The degree of the increase is surprising: some of the assignments increase the node count by more than 2000%. This is strong evidence that the strong-branching solver is not assignment monotonic on this instance.

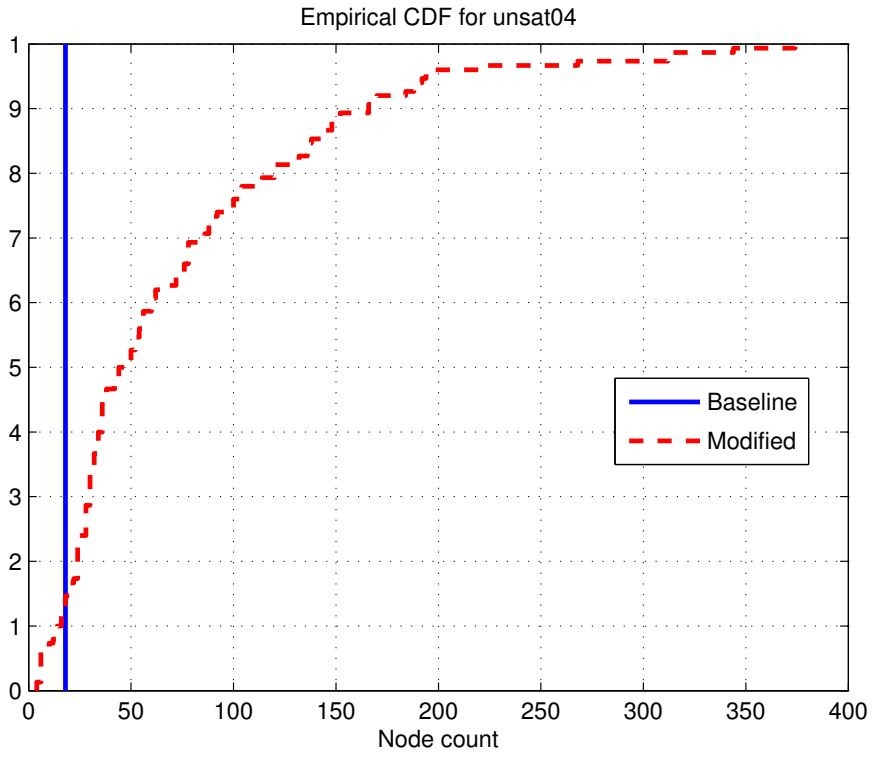

Figure 7: Node-count distribution for strong branching on *unsat04*. The Baseline distribution is the solid vertical line, and the modified distribution is the dashed curve.