Solving Satisfiability using Decomposition and the Most Constrained Subproblem

Eyal Amir and **Sheila McIlraith***
Department of Computer Science,
Gates Building, Wing 2A
Stanford University, Stanford, CA 94305-9020, USA
{eyal.amir,sheila.mcilraith}@cs.stanford.edu

Abstract

In this paper we provide SAT-solving procedures that use the idea of decomposition together with the heuristic of solving the most constrained subproblem first. We present two approaches. We provide an algorithm to find the most constrained subproblem of a propositional SAT problem in polynomial time. We use this algorithm iteratively to decompose a SAT problem into partitions. We also provide a polynomial-time algorithm that uses the idea of minimum vertex separators iteratively to provide different decompositions. We show how to solve SAT problems, using these algorithms to emphasize solving the most constrained subproblem first.

1 Introduction

Problem structure is used to speed up real-world problem solving in constraint satisfaction problems (CSPs), Bayes nets and propositional satisfiability problems (SAT) [9; 3; 20; 7; 1]. Typically, one transforms the given problem into a tree of subproblems that is then processed to yield the solution. To minimize the processing time, it is common to search for trees that minimize the size of the largest subproblems (i.e., TREEWIDTH). The main difficulty with these algorithms is that finding optimal decompositions is NP-hard, and it is unknown whether constant-factor approximations can be found in polynomial time [2; 4].

In this paper we provide SAT-solving procedures that use the idea of decomposition together with the heuristic of solving the most constrained subproblem first. Most notable among the contributions in this paper is a polynomial-time algorithm for finding the most constrained subproblem of a SAT problem. Here, we say a SAT problem is highly *constrained* if it has a high clause-to-variable (*c/v*) ratio. We use this algorithm iteratively to provide a decomposition of a given SAT problem in polynomial time. We also present a second decomposition algorithm that improves a greedy algorithm of [1]. It views the SAT problem as a graph and decomposes it by iteratively finding the minimum vertex-separators. We provide algorithms for SAT that exploit these decompositions and the constrainedness of each of the subproblems. Our approach generalizes the simple SAT-search heuristic that instantiates the most constrained variable first.

Our view on the constrainedness of SAT problems is supported by analyses that showed that, for many important distributions of theories, the expected number of models of a theory decreases as the c/v ratio increases (e.g., [19; 6]). Other works that use the c/v ratio and more sophisticated parameters to predict satisfiability and guide SAT-search algorithms include [16; 18]. Our view is also in line with the work on constrainedness of search of [12; 23] and is consistent with [15] that shows that the difficulty of showing unsatisfiability corresponds to the size of minimal unsatisfiable sub-problems. Following this line, we wish to find those subproblems that are most likely to have the least number of models.

In Section 2 we present an algorithm for decomposing a propositional theory using minimum vertex-separators iteratively. We also present an algorithm that uses the resulting tree of partitions to determine the satisfiability of the theory, using the c/v ratio of partitions to determine the order in which partitions are processed. Finally, we show how to order variables for instantiation by a backtracking search procedure. In Section 2.1, we present a procedure that finds the most constrained subproblem of a SAT problem. This is followed in Section 2.2 by an algorithm that iteratively finds the next most constrained subproblem, ultimately generating a decomposition of the theory into mutually exclusive partitions. We use this decomposition to determine the satisfiability of the theory in similar ways to the first decomposition. Finally, in Section 2.3 we present an algorithm that interleaves searching for the most constrained subproblem with searching for a solution to the SAT problem.

2 Partition-Based Satisfiability Checking

In this section we present the first of two approaches to partition-based SAT. We describe an algorithm for decomposing a theory into subtheories or partitions, and show how to solve SAT by processing these partitions in an order that depends on the c/v of each partition. We also use the partitions and their c/v to order variables for a SAT-search procedure.

2.1 Decomposing and Ordering Partitions

Procedure Split-Thy, presented in Figure 1, uses procedure Split2 to decompose a theory into a tree of partitions. It is

* Knowledge Systems Laboratory (KSL)

given a theory[1], $\mathcal{A}$, and limitations on the partition size (a lower limit, M) and the separators between partitions (an upper limit, l). Split2 initially considers the theory as one big partition, and at every recursive iteration it breaks one of the partitions in two. It represents the tree structure of the partitions in a global variable, G_{str}. This tree structure and the set of partitions, $\{\mathcal{A}_i\}_{i \leq p}$, is returned as the result of Split-Thy. An example of the input and the output is shown in Figure 2.

PROCEDURE Split-Thy($\mathcal{A}$, M, l)
$\mathcal{A}$ is a theory. M is the lower limit on the number of symbols in a partition. l is the upper limit on the number of symbols shared between partitions.

1. Let $G(V, E)$ be an undirected graph with $V = L(\mathcal{A})$ and $E = \{(l_1, l_2) \mid \exists C \in \mathcal{A}\ l_1, l_2 \in L(C)\}$.
2. Let $G_{str}(V_{str}, E_{str})$ be an undirected graph with $V_{str} = \{\{V\}\}$ and $E_{str} = \emptyset$.
3. Run Split2(G, M, l, nil, nil).
4. For every $v \in V_{str}$, let $\mathcal{A}_v = \{C \in \mathcal{A} \mid L(C) \subset v\}$. Return $\{\mathcal{A}_v\}_{v \in V_{str}}$ and G_{str}.

PROCEDURE Split2(G, M, l, a, b)
$G = (V, E)$ is an undirected graph. M, l as above. a, b are in V or are nil.

1. If $|V| < M$, then return V.
2. (a) If $a, b = nil$, find a minimum vertex separator, R, in G. (b) Otherwise, if $b = nil$, find a minimum vertex separator, R, of a in G. (c) Otherwise, find minimum vertex separators, R_a of a in G, and R_b of b in G. Let R be the smaller of R_a, R_b.
3. If $R = V$ or $|R| > l$ then return V.
4. Let G_1, G_2 be the two subgraphs of G separated by R, with R included in both subgraphs.
5. Let $V_{str} \leftarrow V_{str} \setminus \{\{V\}\} \cup \{\{V_1\}, \{V_2\}\}$ and $E_{str} \leftarrow E_{str} \cup \{(\{V_1\}, \{V_2\})\}$. Change the edges that connected to $\{V\}$ to connect to one of $\{V_1\}, \{V_2\}$.
6. Create G_1', G_2' from G_1, G_2, respectively, by aggregating the vertices in R into a single vertex r, removing all self edges and connecting r with edges to all the vertices connected by edges to some vertices in R.
7. Run $Split2(G_1', M, l, r, a)$, $Split2(G_2', M, l, r, b)$. Replace r in the nodes of V_{str} by the members of R.

Figure 1: An algorithm for generating partitions of axioms.

If $\mathcal{A}$ is a propositional theory, let $L(\mathcal{A})$ be the set of its propositional symbols. Split-Thy creates a graph called the *symbols graph* that represents the connections between the symbols in $L(\mathcal{A})$. Each symbol in $L(\mathcal{A})$ is represented by a node in this graph, and two nodes are connected iff their respective symbols appear together in a clause in $L(\mathcal{A})$. Figure 3 (top) illustrates the symbols graph of theory $\mathcal{A}$ from Figure 2 and the connected symbols graphs (bottom) of the individual partitions $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$. Notice that each axiom creates a clique among its constituent symbols.

[1] In this paper, a theory is a set of propositional clauses.

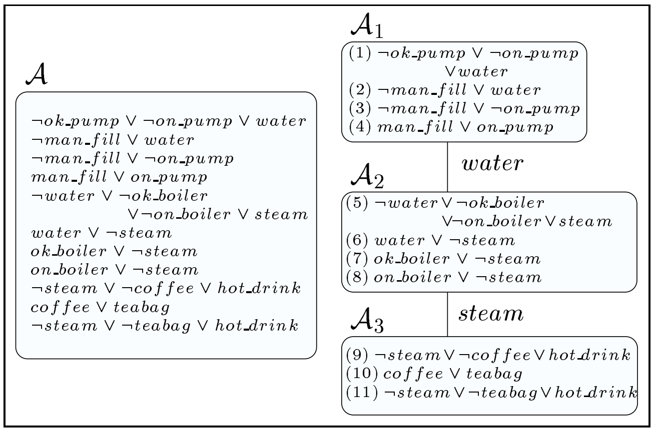

Figure 2: A partitioning of $\mathcal{A}$ and intersection graph.

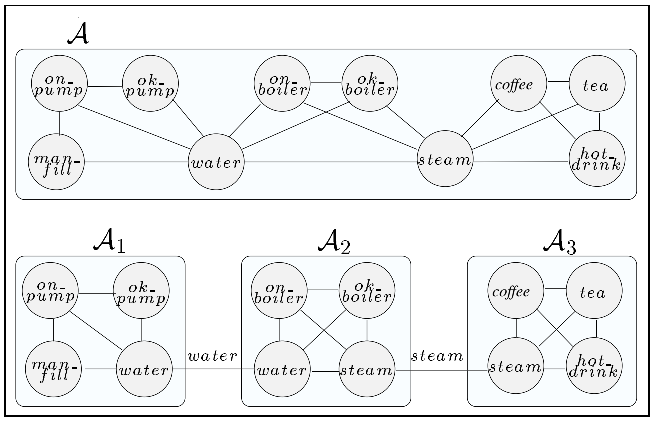

Figure 3: Decomposing $\mathcal{A}$'s symbols graph.

Split2 partitions the theory $\mathcal{A}$ by taking as input its symbols graph, $G = (V, E)$, the two limiting parameters, M and l, and nodes $a, b \in V$ that are initially set to nil. Split2 updates the global variable G_{str} to represent the progressing decomposition. In each recursive call, Split2 finds a minimum vertex separator of a, b in G (i.e., a minimum-size set of vertices that crosses every path between a, b). If one of a, b or both are nil, it finds the overall minimum vertex separator between all vertices and the non-nil vertex (or all other vertices). This separator splits G into two graphs, G_1, G_2, and the process continues recursively. An algorithm for finding minimum vertex-separators is given in [10].

Theorem 2.1 *Algorithm Split-Thy takes time $O(n^{\frac{5}{2}} * m)$, for n propositions and m clauses.*

A similar algorithm to Split-Thy is described in [1]. Split-Thy improves on this algorithm by allowing decompositions into trees of partitions that the algorithm of [1] could not generate. In addition, it is not clear how to generate the tree of partitions from that algorithm, whereas in our algorithm the tree is explicit in the output.

2.2 Ordered Partition-Based SAT

Given a partitioning of a theory, $\mathcal{A}$, we run an ordered partition-based SAT procedure that aggregates computation on each of the partitions, and then uses the results of this computation to decide the satisfiability of $\mathcal{A}$. The procedure is detailed in Figure 4. It takes as input a theory, $\mathcal{A}$, and returns as output all its models, or FALSE if there are no models. The procedure can be made to return a single model by changing step 8 to leave only one entry in T_{i_1} at each iteration. This will save space and time in case there are too many models of $\mathcal{A}$ and only one is needed. Applications that require all the models include verification, nonmonotonic reasoning and knowledge compilation.

We describe the algorithm using database notation [22]. $\pi_{p_1,...,p_k}T$ is the *projection* operation on a relation T. It produces a relation that includes all the rows of T, but only the columns named $p_1, ..., p_k$ (suppressing duplicate rows). $S \bowtie R$ is the *natural join* operation on the relations S and R. It produces the cross product of S, R, selecting only those entries that are equal between identically named fields (checking $S.A = R.A$), and discarding those columns that are now duplicated (e.g., $R.A$ will be discarded).

PROCEDURE CV-Compile-Join($\mathcal{A}$)
$\mathcal{A}$ a propositional clausal theory.

1. Let $\{\mathcal{A}_v\}_{v\in V}$ and $G(V, E) = G_{str}$ be the results returned by Split-Thy($\mathcal{A}$, 1, n).
2. $\forall v \in V$, $L(v) = \bigcup_{(v,w)\in E}(L(\mathcal{A}_v) \cap L(\mathcal{A}_w))$.
3. Let $cv(\mathcal{A}_v) = \frac{\#clauses(\mathcal{A}_v)}{\#variables(\mathcal{A}_v)}$ for all $v \in V$. Let $i_1, ..., i_p$ be indexes such that $cv(\mathcal{A}_{i_j}) \geq cv(\mathcal{A}_{i_{j+1}})$.
4. For each j from 1 to p (with this order):
   – For every truth assignment A to $L(i_j)$:
     - If $\exists(i_a, i_j) \in E$ $(a < j \wedge A \cap L(\mathcal{A}_a) \in T_{i_a})$, then perform SAT-search on $\mathcal{A}_{i_j} \cup A$. If it is successful, insert A to the table T_{i_j}.
5. If $T_{i_j} = \emptyset$, for some $j \leq p$, return FALSE.
6. Let $dist(v, w)$ $(v, w \in V)$ be the length of the shortest path between v, w in G. Let $v \prec w$ iff $dist(v, i_1) < dist(w, i_1)$ and $(v, w) \in E$.
7. Iterate over $v \in V$ in reverse $dist(v, i_1)$-order (highest to lowest), excluding i_1: Take $w \in V$ that satisfies $w \prec v$.
   - $T_w \leftarrow T_w \bowtie (\pi_{L(w)}T_v)$ (*Join* T_w with those columns of T_v that correspond to $L(v)$).
     If $T_w = \emptyset$, return FALSE.
8. Iterate over $v \in V$ in $dist(v, i_1)$-order (lowest to highest), excluding i_1: Let $T_{i_1} \leftarrow T_{i_1} \bowtie T_v$.
9. Return T_{i_1}.

Figure 4: An algorithm for SAT of a propositional theory, ordering the processing using the c/v ratio.

This procedure uses the c/v ratio of the symbols and clauses in each partition to determine an order on the partitions. It performs a *compilation* process for each of the partitions, and then *joins* the results of the compilations, determining SAT. In this algorithm we use a subroutine for SAT-search that can be any SAT procedure that answers TRUE/FALSE. If this subroutine is complete, then our algorithm is complete.

This process is similar to the one exploited by [1]. However, CV-Compile-Join follows the order on the partitions during the compilation process and avoids unnecessary SAT-searches. This guarantees that harder sub-problems (that are more likely to be unsatisfiable) are tackled first, pruning the search/compilation space for the partitions that follow. CV-Compile-Join also returns all the models of $\mathcal{A}$ (or a single model, if desired), whereas the procedure described in [1] only reports TRUE or FALSE.

Theorem 2.2 *Procedure* CV-Compile-Join *is sound and complete for SAT checking, if its SAT-search subroutine is sound and complete. If the largest partition generated in step 1 of the procedure has k propositional symbols, then the algorithm (up to step 8) has a worst-case time bound of*

$$O(n^{\frac{5}{2}} * m + p \cdot 2^{|L(i)|} \cdot f_{SAT}(k - |L(i)|))$$

for $\mathcal{A}$ having n propositional symbols and m clauses, i is the index of the largest partition, p is the number of partitions, and f_{SAT} is the time to compute SAT.

2.3 Order for Dynamic Backtracking

One way to apply our decomposition algorithm is to order the variables for a backtracking SAT procedures (e.g., DPLL [8], dependency-directed backtracking [21] dynamic backtracking [13]). Those variables participating in the most constrained partition should come first, then those appearing in the most constrained partition of the remaining partitions and so forth.

Figure 5 presents a simple algorithm that orders the variables for dynamic backtracking [13]. CV-Backtrack takes as input a propositional clausal theory $\mathcal{A}$. It finds a decomposition of $\mathcal{A}$, sorts the partitions according to c/v ratios and sort the variables in accordance to this partitioning. It uses this order on variables to call a dynamic backtracking SAT-search procedure. It returns a model of $\mathcal{A}$ or FALSE.

PROCEDURE CV-Backtrack($\mathcal{A}$)
$\mathcal{A}$ a propositional clausal theory.

1. Let $\{\mathcal{A}_v\}_{v\in V}$ and $G(V, E) = G_{str}$ be the results returned by Split-Thy($\mathcal{A}$, 1, n).
2. Let $cv(\mathcal{A}_v) = \frac{\#clauses(\mathcal{A}_v)}{\#variables(\mathcal{A}_v)}$ for all $v \in V$. Let $i_1, ..., i_p$ be indexes such that $cv(\mathcal{A}_{i_j}) \geq cv(\mathcal{A}_{i_{j+1}})$.
3. For all $q \in L(\mathcal{A})$, let $P(q)$ be a partition $\mathcal{A}_i$ with maximal $cv(\mathcal{A}_i)$ such that $q \in L(\mathcal{A}_i)$.
4. Sort $L(\mathcal{A})$ into $q_1, ..., q_n$ such that if $a < b \leq n$, then $cv(P(q_a)) \geq cv(P(q_b))$.
5. Return the result of running dynamic-backtracking on $\mathcal{A}$ with the order $q_1, ..., q_n$ on the propositional symbols.

Figure 5: An algorithm for SAT of a propositional theory, using decomposition, dynamic backtracking and c/v ordering.

3 Finding the Most Constrained Subproblem

Section 2 proposed one approach to partitioning a theory and performing partition-based SAT checking. In this section, we present an alternate method of partitioning the theory by finding the most constrained subproblem. In the section that follows we use this algorithm for SAT checking. As mentioned above, we take *increasing constrainedness* of a SAT problem to indicate an expectation of a *decreasing number of models*. Thus, we approximate the problem of *finding the most constrained subproblem* using the problem of *finding the subproblem with the highest c/v ratio*.

PROCEDURE Densest-Subproblem($\{c_i\}_{i\leq m}$)
$\{c_i\}_{i\leq m}$ clauses over n propositions enumerated $1,...,n$.

1. Define $G = (V, E, c)$ to be a bipartite network as follows ($L(c)$ is the set of propositions showing in c):
   (a) $V_1 \leftarrow \{1,...,n\}$, $V_2 \leftarrow \{L(c_1),...,L(c_m)\}$, $V_3 \leftarrow \{S, T\}$ (source and sink nodes), $V \leftarrow V_1 \cup V_2 \cup V_3$.
   (b) $E \leftarrow \{(S,i) \mid i \in V_1\} \cup \{(h,T) \mid h \in V_2\} \cup \{(i,h) \mid h \in V_2,\ i \in h\}$.
   (c) Let the capacities of the edges be $c(S,i) \leftarrow \lambda$, $c(i,h) \leftarrow \infty$, $c(h,T) \leftarrow |\{c_i|L(c_i) = h\}|$. All other vertex pairs have capacity 0.
2. Let $f(x) = \sum_{h\in V_2} c(h,T)\prod_{i\in h} x_i$ and $g(x) = \sum_{i=1}^{n} x_i$
3. $\langle x^*, \lambda^*\rangle \leftarrow$ Find-Max-Fraction(G, f, g, n).
4. Let $V^* = \{i \mid i \in V_1,\ x_i = 1\}$. Let $C = \{c_i \mid i \leq n,\ vars(c_i) \subseteq V^*\}$. C is the desired set of clauses.

Figure 6: An algorithm that finds the subproblem (subset of clauses) that has the highest c/v ratio.

Our *Densest-Subproblem* (*D-S*) algorithm is described in Figure 6. It takes as input a theory (a set of clauses), and returns as output a set of clauses that has the highest c/v ratio among all subtheories of the input theory. This algorithm transforms the optimization of the ratio of clauses-to-variables to a *zero-one fractional-programming problem*. It solves this problem using procedure Find-Max-Fraction (Figure 7) and translates the problem back to find the desired set of clauses. In the description that follows we assume some familiarity with the classic concepts of network flow. Those notions that we do not define here can be found in basic algorithms textbooks, such as [5].

A *fractional-programming problem* is a discrete or network optimization problem with fractional objectives. For example, we may have an optimization problem where we wish to find a vector, x, that is a maximum point for $f(x)/g(x)$. A solution is a fixed-point $x*$ of $\lambda(x*) = \max_x\{\lambda(x) = f(x)/g(x)\}$. A *zero-one fractional-programming problem* is further restricted to have

$$\begin{aligned} f(x) &= \textstyle\sum_{P\in A} a_P \prod_{i\in P} x_i + \sum_{i=1}^{n} a_i x_i \\ g(x) &= \textstyle\sum_{Q\in B} b_Q \prod_{i\in Q} x_i + \sum_{i=1}^{n} b_i x_i \end{aligned} \tag{1}$$

for some sets A, B, coefficients a_i, b_i for $i = 1,...,n$ and coefficients a_P, b_Q for $P \in A, Q \in B$, respectively.

To find a solution to such a problem we specialize and slightly modify the algorithm proposed by [11]. The original algorithm may yield an empty set of clauses, which we wish to avoid. The algorithm uses the *push-relabel* max-flow algorithm of Goldberg and Tarjan (see [5]), the *parametric max-flow algorithm* of [11], the *fractional programming* algorithm of [14] and the translation of [17] of the selection problem into a flow problem. Together, they provide a general solution to *zero-one fractional programming*, as shown in [11].

Our combined algorithm is shown in Figure 7. Find-Max-Fraction takes as input a bipartite network (see an example in Figure 8), functions f and g and an integer $n > 0$. It returns as output a fixed point $\langle x^*, \lambda^*\rangle$, where x^* is the maximum vector and $\lambda^* = f(x^*)/g(x^*)$. We do not repeat the push-relabel (*Preflow* in [11]) algorithm here and the reader is referred to [5].

PROCEDURE Find-Max-Fraction(G, f, g, n)
$G = (V, E, c)$ a bipartite network. f and g functions. $n > 0$.

1. Set $x^0 = \langle 1,...,1\rangle$. Compute $\lambda^0 = f(x^0)/g(x^0)$. Set $k = 0$ and $N = |V|$. Set the preflow $\overrightarrow{f}(S,v) = c(S,v)$ for all v connected to S and $\overrightarrow{f}(v,w) = 0$ everywhere else. Set $d(S) = N$ and $d(v) = 0$ for all $v \neq S$ (for push-relabel).
2. Define G^R as G with all arcs reversed and source and sink switched.[a] Define $\overleftarrow{f}$ as the opposite flow to $\overrightarrow{f}$.
3. Set λ in G^R to λ^k.
4. $\overleftarrow{f} \leftarrow$ push-relabel(G^R) (reuse the previous $\overleftarrow{f}$ and d).
5. Compute a minimum cut $(X, \bar{X})$ from $\overrightarrow{f}$ with $T \in X$:
   $\Gamma \leftarrow \{v \mid (v,T) \in E,$ no residual path from S to $v\}$.
   $X \leftarrow \{x \mid (S,x) \in E, \exists v \in \Gamma\ (x,v) \in E\} \cup \Gamma\ \cup \{T\}$.
6. $\forall i \leq n$: if $i \in X$, set $x_i^{k+1} = 1$; otherwise $x_i^{k+1} = 0$.
7. If $f(x^{k+1}) = \lambda^k * g(x^{k+1})$, return $\langle x^k, \lambda^k\rangle$.
8. Let $\lambda^{k+1} = f(x^{k+1})/g(x^{k+1})$. Set $k \leftarrow k+1$ and go to 3.

[a] This allows *push-relabel* to reuse previously computed flows.

Figure 7: Find x^* with $\lambda(x*) = \max_x\{\lambda(x) = \frac{f(x)}{g(x)}\}$.

Find-Max-Fraction iteratively refines λ and x until a fixed point is achieved. At each loop iteration, the maximum flow $\overrightarrow{f}$ is computed for G using the push-relabel algorithm. Then, we check to see if we got a fixed-point. If we did not, we increase λ according to the newly found flow, and repeat the loop. The algorithm is guaranteed to terminate after no more than $N = |V|$ repetitions of the loop.

D-S translates our original problem to a zero-one fractional-programming problem by letting $f(x)$ correspond to the number of clauses and $g(x)$ correspond to the number of variables. In equation (1) we let $x_i = 1$ indicate that proposition i is in our set of variables, and $x_i = 0$ indicate it is not in our set. We take $B = \emptyset$ and $\forall i.b_i = 1, a_i = 0$. Finally, A is a set that includes sets of variables. Each set in A corresponds is the set of variables of a clause. For each P in A, a_P is set to the number of clauses that use exactly the propositions of P. $f(x)$ is then the number of clauses who's variables are from our set, and $g(x)$ is the number of variables in our set. Notice that possibly $L(c_i) = L(c_j)$. In that case, c_i and c_j will have a single vertex, since V_2 is a set.

The bipartite network represents this maximization problem by having one vertex for each set $P \in A$, one vertex for each propositional symbol $i \in \{1, ..., n\}$, and two additional vertices, a source, S, and a sink, T (see Figure 8). The network has an arc (v, T) of capacity a_P for each vertex v corresponding to a set $P \in A$, an arc (S, i) of capacity λ for every $i \in \{1, ..., n\}$, and an arc (i, v) of infinite capacity for every vertex corresponding to a set $P \in A$ that has i as one of its elements.

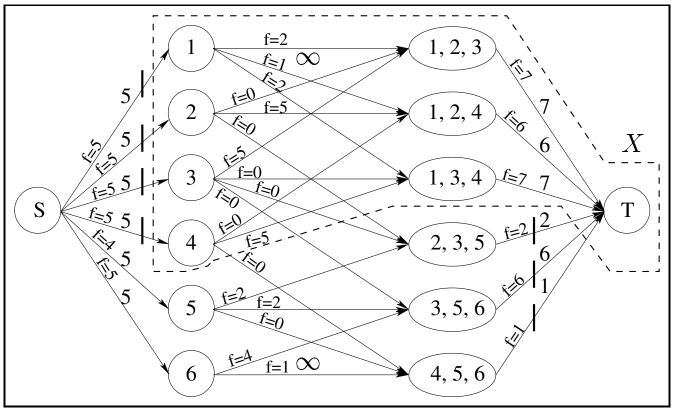

Figure 8: Maximum flow on a bipartite network G and the cut $(X, \bar{X})$ for $\lambda = 5$ (every edge has capacity and flow ('f=...')).

An example of the final loop iteration for a network G with $\lambda = 5$ is shown in Figure 8. The flow $\overrightarrow{f}$ (in the figure, 'f') produces $x^k = \langle 1, 1, 1, 1, 0, 0 \rangle$. This x^k produces $\lambda^k = f(x^k)/g(x^k) = 5$, so it is the fixed point. Thus, in this example, the highest c/v (which is $\lambda^k = 5$) is achieved by taking the propositional symbols corresponding to $1, 2, 3, 4$ and the clauses that include only symbols from this set.

Proposition 3.1 *Procedure Densest-Subproblem returns a subset of the variables (and the clauses represented in the vocabulary of those variables) that has the highest c/v ratio. The running time of procedure D-S is $O(k * m^2 * log(m))$, if each clause is of length $\leq k$.*

As is, the algorithm outputs the largest sub-problem with this c/v ratio. It can be changed to output a minimal sub-problem with this c/v ratio or a maximal/minimal one that includes a specified set of clauses or propositional symbols. To find a sub-problem with highest c/v that includes a clause C and is minimal in size, change step 5 in procedure Find-Max-Fraction to set

$$\Gamma \leftarrow \{v \mid (v, T) \in E, \text{ exists a residual path from } v \text{ to } T\},$$

and increase the capacity of (v, T) by 1 for the node v corresponding to the clause C. To find a sub-problem with highest c/v that is of minimal size, find the minimal X, iterating the last procedure over all nodes v that appear in the original Γ (before our modification above). For lack of space we do not bring more details of these modifications here.

4 Applying *D-S* to Satisfiability

In this section we use the D-S procedure to create partitions and ordering of variables. *D-S-var-order* (Figure 9) makes iterative use of procedure D-S. It uses procedure D-S to order the variables for a backtracking search through the SAT search space. Those variables participating in the most constrained subproblem come first, then those appearing in the most constrained subproblem of the remaining axioms and so forth. After decomposition, we use CV-Compile-Join (Figure 4) or CV-Backtrack (Figure 5) to reason with these partitions or variable ordering (as in step 4 in Figure 9).

PROCEDURE D-S-Var-Order($\{c_i\}_{i \leq m}$)
$\{c_i\}_{i \leq m}$ clauses over n propositions enumerated 1, ..., n.

1. Let $V = \{1, ..., n\}$, $C = \{c_i\}_{i \leq m}$, $i = 1$.
2. While $V \neq \emptyset$,
   (a) Run D-S(C), returning C_i and V_i.
   (b) Set $C \leftarrow C \setminus C_i$; $V \leftarrow V \setminus V_i$; $i \leftarrow i + 1$.
3. Order the variables in $\{1, ..., n\}$ such that if $x \in V_i$ and $y \in V_j$ and $i < j$ then $x \prec y$.
4. Use the order $\prec$ in a dynamic-backtracking search of SAT on $\mathcal{A} = \{c_1, ..., c_m\}$.

Figure 9: Order variables according to most constrained subproblems, and perform SAT search using this order.

Using D-S-Var-Order can be considered a *static decomposition* approach to solving SAT problems. After a subproblem is extracted, the rest of the clauses are considered as a fresh new problem. However, the most constrained subproblem found in the second and subsequent iterations of procedure D-S-Var-Order can depend on the way we use the first subproblem. In backtracking SAT procedures the variables are set one by one to some truth values (interleaving such setting choices with search-pruning procedures such as unit resolution and tautology elimination), backtracking (to some point) when there is no way to reach a satisfying truth assignment. If we set the variables in the first subproblem V_1 before any other variables, the second iteration of finding the current most constrained subproblem can use this setting.

Procedure *Dynamic-DS-SAT*, shown in Figure 10 presents a *dynamic ordering* approach. It first selects V_1, the first set of variables, allows the SAT-search to instantiate the variables of V_1, eliminates those clauses that are satisfied by the instantiation, and eliminates the variables of V_1 from the rest of the clauses. $C[A]$ denotes the set of clauses C after removing clauses that are satisfied by the truth assignment A and removing symbols that appear with opposite polarity to the one sanctioned by A from the rest of the clauses. Dynamic-DS-SAT is then applied to search for the current most constrained subproblem V_2 in the updated set of clauses.

To use procedure Dynamic-DS-SAT, we amend procedure D-S to allow our notion of *constrainedness* to depend on the length of each clause. We use the observation that a clause of length k has an expectation for $\frac{2^k - 1}{2^k}$ satisfying models. This allows us to approximate the constraining of such a clause to be $\frac{2^k}{2^k - 1}$. E.g., for $k = 1$ we get constraining of 2, and for $k = 2$, we get $\frac{4}{3}$. Change step 2 in procedure D-S (Figure 6) to reflect this *weighing of clauses* by replacing it with

PROCEDURE Dynamic-DS-SAT($\{c_i\}_{i\leq m}$)
$\{c_i\}_{i\leq m}$ clauses over n propositions enumerated $1, ..., n$.

1. Let $V = \{1, ..., n\}$, $C = \{c_i\}_{i\leq m}$, $i = 1$, $A = \emptyset$.
2. Perform indefinitely:
   (a) Run D-S(C), returning C_i and V_i.
   (b) Run backjumping search of SAT on $C[A]$, instantiating only V_i.
   (c) If the search did not conclude successfully,
   - If $i = 1$, return FALSE (not satisfiable).
   - Otherwise, backjumping needs to change a variable that is in V_j ($j < i$). Set $V \leftarrow V \cup V_j \cup ... \cup V_{i-1}$, and remove the truth assignments for $V_j \cup ... \cup V_{i-1}$ from A. Set $i \leftarrow j$. Go to step 2b.

   (d) If the search concluded successfully (no backtracking),
   - If $V = \emptyset$, return TRUE (satisfiable).
   - Otherwise, add the the truth assignment found to A, let $V = V \setminus V_i$, let $i \leftarrow i + 1$ and go to step 2a.

Figure 10: An algorithm for SAT of a propositional theory, using backjumping and dynamic c/v ordering.

2. For each $i \leq m$, let $w(c_i)$ be $\frac{2^{k_i}}{2^{k_i}-1}$, for k_i being the number of propositions in c_i. Let
$$f(x) = \sum_{h\in V_2} w(c_i)c(h,T)\prod_{i\in h} x_i$$
$$g(x) = \sum_{i=1}^{n} x_i$$

The *dynamic ordering* approach has the benefit that we get a better estimate of the most constrained subproblem at each stage. However, it comes in price of adding the time of running D-S multiple times throughout the search.

5 Summary and Discussion

We presented algorithms that use the c/v ratio to guide SAT solvers in choosing and ordering subproblems, with the objective of solving the most constrained subproblem first. The rationale behind our approach is to use the observation that real-world theories are not uniformly distributed and are often comprised of loosely coupled subtheories of varying constrainedness. We were also motivated by the need to generalize the simple search heuristic that instantiates the most constrained variable first. The satisfiability of $\mathcal{A}$ is correlated to its most constrained subproblem. As such, to determine that a theory is unsatisfiable, our approach is to focus computational effort on the most constrained subproblem.

To this end, we provided a fast algorithm for decomposing SAT problems into subproblems that are loosely connected, ordering the subproblems according to c/v ratio, and then solving the original SAT problem using this order. Our SAT procedure returns all the models, if needed. We also provided an algorithm that finds the subproblem with highest c/v ratio in a low polynomial time. We used it iteratively to decompose given SAT problems, then performed SAT on the separate subproblems. We also proposed using the decompositions to provide variable orders for backtracking algorithms.

Our algorithms are suitable for structured real-world problems, such as the commonsense theories found in the DARPA High Performance Knowledge Base (HPKB) program. Our results show that proper decomposition of such problems can produce a very fast solution, even if the problems are large. This work also generalizes and justifies heuristic approaches to SAT search that choose and instantiate the most constrained variable first, in an iterative fashion.

The work presented in this paper is still in a preliminary stage. In the future we plan to experimentally evaluate our algorithms on a suite of structured problems, to confirm the efficacy of our algorithms in practice.

References

[1] Eyal Amir and Sheila McIlraith. Paritition-based logical reasoning. In *Proc. KR '2000*, pages 389–400. MK, 2000.

[2] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embebdings in a K-tree. *SIAM J. Alg. and Discr. Meth.*, 8:277–284, 1987.

[3] Ann Becker and Dan Geiger. A sufficiently fast algorithm for finding close to optimal junction trees. In *Proc. UAI '96*, pages 81–89. MK, 1996.

[4] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Mathematical Foundations of Computer Science 1997*, volume 1295, pages 19–36. Springer, 1997.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1989.

[6] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proc. AAAI '93*, pages 21–27, 1993.

[7] Adnan Darwiche. Utilizing knowledge-based semantics in graph-based algorithms. In *Proc. AAAI '96*, pages 607–613, 1996.

[8] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *CACM*, 5:394–397, 1962.

[9] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[10] Shimon Even. *Graph Algorithms*. Computer Science Press, 1979.

[11] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30–55, 1989.

[12] Ian Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The constraindness of search. In *Proc. AAAI '96*, pages 246–252, 1996.

[13] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[14] J. R. Isbell and H. Marlow. Attrition games. *Naval Research Logistics Quarterly*, 2:71–93, 1956.

[15] Dorothy L. Mammen and Tad Hogg. A new look at the easy-hard-easy pattern of combinatorial search difficulty. *Journal of Artificial Intelligence Research*, 6:47–66, 1997.

[16] David M. Pennock and Quentin F. Stout. Exploiting a theory of phase transition in three-satisfiability problems. In *Proc. AAAI '96*, pages 253–258, 1996.

[17] J. M. W. Rhys. A selection problem of shared fixed costs and network flows. *Management Science*, 17(3):200–207, 1970.

[18] Tuomas W. Sandholm. A second order parameter for 3sat. In *Proc. AAAI '96*, pages 259–265, 1996.

[19] Bart Selman, David Mitchell, and Hector Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, pages 17–29, 1997. Selman, B.Mitchell, D.Levesque, H.

[20] Kirill Shoikhet and Dan Geiger. A practical algorithm for finding optimal triangulations. In *Proc. AAAI '97*, pages 185–190. MK, 1997.

[21] Richard M. Stallman and Gerald Jay Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[22] Jeffrey D. Ullman. *Principles of Database and knowledge-base systems*, volume 1. Computer Science Press, 1988.

[23] Toby Walsh. The constraindness knife-edge. In *Proc. AAAI '98*, pages 406–411, 1998.