Identifying Conflicts in Overconstrained Temporal Problems

Mark H. Liffiton, Michael D. Moffitt, Martha E. Pollack, and Karem A. Sakallah
Department of Electrical Engineering and Computer Science,
University of Michigan, Ann Arbor 48109-2122
{liffiton, mmoffitt, pollackm, karem}@eecs.umich.edu

Abstract

We describe a strong connection between maximally satisfiable and minimally unsatisfiable subsets of constraint systems. Using this relationship, we develop a two-phase algorithm, employing powerful constraint satisfaction techniques, for the identification of conflicting sets of constraints in infeasible constraint systems. We apply this technique to overconstrained instances of the Disjunctive Temporal Problem (DTP), an expressive form of temporal constraint satisfaction problems. Using randomly-generated benchmarks, we provide experimental results that demonstrate how the algorithm scales with problem size and constraint density.

1 Introduction

Many AI problems can be cast in terms of Constraint-Satisfaction Problems (CSPs), and a large number of systems have been developed to efficiently compute solutions to sets of constraints. However, not every set of constraints is satisfiable. This paper reports on a technique for efficiently identifying sets of conflicting constraints in an overconstrained problem. We focus on temporal constraint problems, which play a prominent role in planning, scheduling, and other applications of interest in AI. Our approach is particularly well-suited to temporal problems, in which conflicts among constraints can be resolved by weakening, rather than completely abandoning, constraints.

We build on our previous work [Liffiton and Sakallah, 2005] on identifying Minimally Unsatisfiable Subsets of constraints (MUSes). Given a set of constraints C, an MUS of C is a subset of C that is (1) unsatisfiable and (2) minimal, in the sense that removing any one of its elements makes the rest of the MUS satisfiable. Each MUS thus provides information about a conflict that must be addressed to solve the given CSP. In general, an arbitrary CSP may contain multiple MUSes, and all of them must be resolved by constraint relaxation before the CSP can be solved. Identifying the MUSes of a CSP makes it possible to reason about how to weaken conflicting constraints to make a solution feasible.

Prior work on MUSes has largely been restricted to algorithms for finding a single MUS. In addition, these algorithms have been specific to particular types of constraint systems, e.g., Boolean satisfiability ([Bruni and Sassano, 2001; Zhang and Malik, 2003; Oh *et al.*, 2004] and others) or linear programming [Chinneck, 1996], and the techniques developed often cannot be readily generalized to other types of constraints. In contrast, we explore a deep connection between maximal satisfiability and minimal unsatisfiability to develop techniques that can be used to find *all* MUSes of a constraint system, regardless of constraint type. In this paper, we demonstrate the approach by applying it to Disjunctive Temporal Problems (DTPs) [Stergiou and Koubarakis, 1998], a particularly expressive form of temporal constraints. Our previous work focused on finding MUSes of Boolean satisfiability instances, which have applications in circuit design and verification. DTPs provide a richer space for using MUSes, specifically because their constraints can be relaxed without any change in structure, something that is not possible with Boolean variables and constraints.

In Section 2 of the paper, we define DTPs and discuss the motivation for finding their MUSes further. Section 3 describes the relationship between maximal satisfiability and minimal unsatisfiability. Section 4 contains the specific algorithms we employed to exploit this relationship for finding MUSes of DTPs. Empirical results are given in Section 5, and Section 6 closes with conclusions and directions for future work.

2 Disjunctive Temporal Problems

A *Disjunctive Temporal Problem* (DTP) is a constraint satisfaction problem defined by a pair $\langle X, C \rangle$, where X is a set of real variables, designating time points, and C is a set of constraints of the form: $c_{i1} \vee c_{i2} \vee ... \vee c_{in}$, where each c_{ij} is of the form $x - y \leq b$; $x, y \in X$ and $b \in \Re$. DTPs are thus a generalization of Simple Temporal Problems (STPs), in which each constraint is limited to a single inequality [Dechter *et al.*, 1991]. A solution to a DTP is an assignment of values to time points such that all constraints are satisfied.

Several algorithms have been developed for solving DTPs [Stergiou and Koubarakis, 1998; Armando *et al.*, 1999; Oddi and Cesta, 2000; Tsamardinos and Pollack, 2003]. Typically, these algorithms transform the problem into a *meta-CSP*, in which the original DTP is viewed as a collection of alternative STPs. Using this approach, the algorithm selects a single disjunct from each constraint of a given DTP. The re-

sulting set forms an STP, called a *component STP*, which can then be checked for consistency in polynomial time using a shortest-path algorithm. Clearly, a DTP D is consistent if and only if it contains at least one consistent component STP. Furthermore, any solution to a consistent component STP of DTP D is also a solution to D itself. Consequently, it is standard in the DTP literature to consider any consistent component STP to be a solution of the DTP to which it belongs.

A number of pruning techniques can be used to focus the search for a consistent component STP of a DTP. These include conflict-directed backjumping, removal of subsumed variables, and semantic branching. The DTP solver Epilitis [Tsamardinos and Pollack, 2003] integrated all these techniques, in addition to no-good recording. Epilitis (and other traditional DTP solvers) perform total constraint satisfaction—that is, their objective is to find a solution that satisfies all the constraints of a DTP. In the event that a DTP is inconsistent, these solvers are capable of detecting such infeasibility, but incapable of providing partial solutions that come close to satisfying the problem. In response, the DTP solver Maxilitis [Moffitt and Pollack, 2005] was designed to find solutions that maximize the number of satisfied constraints.

Sometimes the number of satisfied constraints is not the best measure of a solution; it may instead be important to reason about exactly which constraints are violated, as well as the extent to which they have been violated. We illustrate with the following very small DTP, which we use as an example throughout the paper:

$$C_1 : \{c_{11} : (x - y \leq 1)\}$$
$$C_2 : \{c_{21} : (y - x \leq -2)\}$$
$$C_3 : \{c_{31} : (y - x \leq -3)\} \vee \{c_{32} : (z - y \leq 1)\}$$
$$C_4 : \{c_{41} : (y - z \leq -2)\}$$

Note that this DTP is unsatisfiable. The set of constraints $\{C_1, C_2\}$ is inconsistent, and the same is true for $\{C_1, C_3, C_4\}$. A standard DTP solver would simply report that the problem was unsatisfiable, while a system like Maxilitis would find a solution that maximizes the number of satisfied constraints—in this case, a solution that satisfies $\{C_2, C_3, C_4\}$ and not C_1. But for many applications, it might be important instead to maintain some control over the relationship between x and y that is expressed in C_1. If we knew the source of the conflict involving C_1, we could reason about how to weaken C_1 and the constraints with which it conflicts. For example, we could decide to change the bound on C_1 to 2, or to change the bounds on both C_1 and C_2 to 1.5. We could reason about similar relaxations of C_1, C_3, and C_4 to make them jointly feasible. But in order to do this, we must have identified which constraints are in conflict with one another, and this is precisely what the MUSes tell us.

This ability to relax constraints without completely removing them makes DTPs a much more interesting domain for finding MUSes than, for example, systems of Boolean constraints. An infeasible Boolean satisfiability instance may be made feasible by completely removing constraints, and such changes are indicated directly by techniques that identify a maximal satisfiable subset of constraints. An infeasible DTP, however, can be made consistent by altering the bounds of inequalities, without modifying the structure of the problem. MUSes provide information needed to determine these changes.

3 Maximal Satisfiability and Minimal Unsatisfiability

Our techniques for extracting MUSes are derived from a deep relationship between maximal satisfiability and minimal unsatisfiability. The Maximal Constraint Satisfaction problem (Max-CSP) is an optimization problem on a constraint system C that has the goal of finding an assignment to the variables of C that satisfies as many constraints as possible; this is the solution that Maxilitis finds (e.g., $\{C_2, C_3, C_4\}$ for our example problem).

While Max-CSP is defined in terms of the cardinality of a satisfiable subset of constraints, the definition can be relaxed to have *inaugmentability* as the goal instead. Thus, while we can define Max-CSP(C) as $\{m \subseteq C : |m| \text{ is maximal}, \ m \text{ is satisfiable}\}$, we can define a new problem, *Maximally Satisfiable Subset* (MSS). The definition of the set of MSSes follows, with the set of MUSes defined similarly for comparison:

$$\mathrm{MSSes}(C) = \left\{ \begin{array}{l} m \subseteq C : m \text{ is satisfiable, and} \\ \forall c \in (C \backslash m), \ m \cup \{c\} \text{ is unsatisfiable} \end{array} \right\}$$

$$\mathrm{MUSes}(C) = \left\{ \begin{array}{l} m \subseteq C : m \text{ is unsatisfiable, and} \\ \forall c \in m, \ m \backslash \{c\} \text{ is satisfiable} \end{array} \right\}$$

Each MSS is a subset of C that is satisfiable and inaugmentable; adding any other constraints in C to such a subset will render it unsatisfiable. Notice from the definition that MSSes(C) and MUSes(C) are essentially duals of one another! An MSS is satisfiable and cannot be made larger, and an MUS is *un*satisfiable and cannot be made *smaller*. This relationship is more than cosmetic; we will show how one set is actually an implicit encoding of the other. Note also that any Max-CSP solution is also an MSS; maximal cardinality implies inaugmentability. However, MSSes may not all have maximal cardinality, as illustrated in the example DTP. One MSS of the example is $\{C_2, C_3, C_4\}$, corresponding to the Max-CSP solution. $\{C_1, C_4\}$ is another MSS (adding either C_2 or C_3 makes it unsatisfiable), though smaller.

Now consider C_1, the constraint not included in the Max-CSP solution to our example. Removing C_1 from the DTP makes the remaining system satisfiable. In general, given any MSS, the set of constraints not included in that MSS provides an irreducible "fix" for the original infeasible system; removing these constraints makes it satisfiable. Therefore, we define a "CoMSS" as the complement of an MSS, and the set CoMSSes(C) as:

$$\mathrm{CoMSSes}(C) = \{m \subseteq C : (C \backslash m) \in \mathrm{MSSes}(C)\}$$

This complementary view of MSSes(C) reveals a more meaningful connection between maximal satisfiability and minimal unsatisfiability. Recall that the presence of any MUS in a constraint system C makes C infeasible. Therefore, to make C feasible, every MUS in C must be neutralized by relaxing or removing at least one constraint from it. Any

CoMSSes(C)	C_1	C_2	C_3	C_4
$\{C_1\}$	X			
$\{C_2, C_3\}$		X	X	
$\{C_2, C_4\}$		X		X

$$\begin{aligned}\text{MUSes}(C) &= (C_1)(C_2 \vee C_3)(C_2 \vee C_4) \\ &= C_1C_2 \vee C_1C_3C_4 \\ &= \{\{C_1, C_2\}, \{C_1, C_3, C_4\}\}\end{aligned}$$

MUSes(C)	C_1	C_2	C_3	C_4
$\{C_1, C_2\}$	X	X		
$\{C_1, C_3, C_4\}$	X		X	X

$$\begin{aligned}\text{CoMSSes}(C) &= (C_1 \vee C_2)(C_1 \vee C_3 \vee C_4) \\ &= C_1 \vee C_2C_3 \vee C_2C_4 \\ &= \{\{C_1\}, \{C_2, C_3\}, \{C_2, C_4\}\}\end{aligned}$$

Figure 1: Covering Problems Linking CoMSSes(C) and MUSes(C)

CoMSS is one such set: a CoMSS m is an irreducible set of constraints whose removal makes C satisfiable. Thus, every CoMSS contains at least one constraint from every MUS of C. For temporal problems, achieving satisfiability may not require removing every constraint in a CoMSS, but it does require relaxing all of them. Of course, removing a constraint is equivalent to extreme relaxation, in which the bound of an inequality in the constraint is set to infinity.

Every CoMSS is inherently a solution to a covering problem on MUSes(C). Specifically, it is a solution to the HITTING-SET problem [Garey and Johnson, 1990] on MUSes(C). Given a set C, and a collection of subsets of C, a hitting set of the collection is a subset of C that contains at least one element from each subset in the collection. In this case, the set C is the set of constraints in a constraint system, the collection of subsets of C is the set MUSes(C), and the hitting set of MUSes(C) is a CoMSS of C.

Additionally, in line with the "duality" of MSSes and MUSes, every MUS is a hitting set of the set of CoMSSes. This is the key to extracting all MUSes of a constraint system: first compute the set of CoMSSes of C, then extract MUSes by finding hitting sets of the CoMSSes.

Figure 1 illustrates the relationship with covering problems that link CoMSSes(C) and MUSes(C) for our example DTP. The left table represents the covering problem of finding MUSes that cover the CoMSSes to generate MUSes(C). Each column is a constraint from the DTP, and each row is a single CoMSS. We can say that a constraint "covers" a CoMSS (marked with an 'X' in that row) if it is contained in the CoMSS. Each MUS is then an irreducible subset of the columns that covers all of the rows. In the example, these are found in a manner similar to Petrick's Method from Boolean logic minimization: each row becomes a disjunction of the columns that cover that row, and the disjunctions are conjoined and simplified by the distributive rule. The right half of the figure shows how CoMSSes(C) can be generated in the exact same manner from MUSes(C).

If we are interested primarily in finding an arbitrary set of constraints to relax in order to make an overconstrained set C feasible, then we could simply compute CoMSSes(C): each element of CoMSSes(C) represents one set of constraints that, if sufficiently relaxed, will result in C being feasible. But if we instead want to reason about how to weaken particular constraints, we need to identify the individual sets of conflicting constraints, i.e., MUSes(C). In practice, it is easier to find maximally satisfiable subsets (and thus, CoMSSes) than to find MUSes directly. This follows from the relative simplicity of problems in NP (i.e., SAT) as compared to those in Co-NP (i.e., UNSAT). Thus, our approach will be to compute CoMSSes(C) and then find its irreducible hitting sets.

Simplified-Musilitis-CoMSSes(*A, U*)

1. If ($U = \oslash$)
2. *newCoMSS* ← $\{C | (C \leftarrow \epsilon) \in A\}$
3. *CoMSSes* ← *CoMSSes* ∪ {*newCoMSS*}
4. *backjump-to-*ϵ ← *true*
5. Return
6. C ← **Select-Constraint**(*U*), *U'* ← *U* – {C}
7. For all values $c \in d(C)$
8. *A'* ← *A* ∪ {$C \leftarrow c$}
9. If **consistent**(*A'*)
10. **Simplified-Musilitis-CoMSS**(*A', U'*)
11. If (*backjump-to-*ϵ = *true*) Return
12. *A'* ← *A* ∪ {$C \leftarrow \epsilon$}
13. Unless (**Assignment-Subsumed**(*A',CoMSSes*))
14. **Simplified-Musilitis-CoMSS**(*A', U'*)
15. *backjump-to-*ϵ ← *false*

Figure 2: A simplified version of an algorithm for finding the CoMSSes of a DTP

4 Algorithm Details

We now describe the details of our algorithm for finding MUSes. We use a serial decomposition of the task. First we find all of the CoMSSes of a DTP using an algorithm that borrows heavily from Maxilitis; this algorithm could be readily generalized for other types of constraints. Second, we use techniques for extracting MUSes that operate completely independently of how the CoMSSes were generated and the types of constraints involved. The two phases combine to form a solver, which we name Musilitis, that is capable of diagnosing the infeasibility of any given DTP. The solver is both sound, in that the sets of constraints it returns are all MUSes, and complete, in that it will find all MUSes of the given constraint system.

4.1 Finding CoMSSes(C)

In Figure 2, we present a simplified version of an algorithm for finding the set of all CoMSSes of a DTP. This backtrack-

ing search resembles in many ways the branch-and-bound algorithm employed by Maxilitis, in that it effectively searches through the space of all partial assignments of disjuncts to constraints. (Recall that we convert each disjunctive constraint in the original DTP into a meta-level variable, whose domain is the set of disjuncts.) The set A contains the current partial assignment, and the set U contains those constraints that have not yet been instantiated. Whenever a constraint is instantiated with an empty assignment (i.e., it is purposefully chosen to be left unsatisfied), such an assignment is labeled by the symbol ϵ.

To understand the algorithm, consider an example with 50 constraints. Suppose that we begin the search and find a solution in which two constraints, say C_6 and C_{28}, have been left unsatisfied. At this point, the search can prune any other solutions that would necessarily leave C_6 and C_{28} unsatisfied, as such candidates would be incapable of leading to an irredundant CoMSS. We handle this in our algorithm by means of a backjump flag that causes the search to immediately backjump to the deepest point at which the assignment to one of the unsatisfied constraints was made.

Lines 1 through 5 handle the case when a particular solution with all constraints instantiated has been found. The set of unsatisfied constraints (those assigned ϵ) is a CoMSS, and it is added to a global list of CoMSSes (initialized to $\oslash$). At this point, the flag *backjump-to-*ϵ is toggled to avoid any further search within the current subtree whose solutions would be subsumed by the CoMSS just found.

If one or more constraints remain uninstantiated, line 6 selects a constraint to satisfy, and lines 7 – 11 attempt various disjuncts with which to satisfy it. However, if one of the recursive calls finds a solution, the *backjump-to-*ϵ flag may be enabled, causing the algorithm to bubble up the search tree until the deepest assignment of ϵ is reached.

Line 12 is where the algorithm attempts the empty assignment. If this assignment is not subsumed by any CoMSSes found in prior solutions (line 13), the algorithm will pursue this partial assignment (line 14), afterwards resetting the backjump flag in the event that a solution below was discovered (line 15). It should be noted that this version of the algorithm is not guaranteed to find only irredundant CoMSSes, but any redundancies can be easily removed in a postprocessing step.

A slight variation on this approach is to find CoMSSes in increasing order of size. By blocking CoMSSes found in increasing order, the algorithm ensures that all subsequent CoMSSes found are irredundant. Such an incremental algorithm can easily be achieved by setting an upper bound on the number of constraints violated, and repeatedly calling the given algorithm, each time increasing the upper bound by 1 until the problem is no longer satisfiable. While many searches are performed, the pruning allowed by the smaller CoMSSes may potentially save time otherwise spent finding redundant CoMSSes. This incremental algorithm proved to be extremely effective in Maxilitis, where the goal was simply that of maximal constraint satisfaction.

If implemented exactly as shown, the algorithm shown in Figure 2 would perform rather poorly. The pseudocode does not illustrate the use of forward checking, a vital technique used in all DTP solving algorithms. It also does not demonstrate the use of two pruning techniques called *removal of subsumed variables* and *semantic branching* [Armando *et al.*, 1999] that are known to greatly improve the power of DTP solvers. Luckily, these methods remain available when finding CoMSSes of a DTP, and they are integrated into our Musilitis solver. Other common issues, such as variable and value ordering heuristics, also play a large role in guiding the search, although we do not discuss them here due to space limitations.

Musilitis-ExtractMUS(*CoMSSes*)

```
1.  MUS ← ⊘
2.  While (CoMSSes is not empty)
3.      curCoMSS ← pop(CoMSSes)
4.      newConstraint ← pop(curCoMSS)
5.      MUS ← MUS ∪ newConstraint
6.      For all testConstraint ∈ curCoMSS
7.          For all testCoMSS ∈ CoMSSes
8.              If testCoMSS contains testConstraint
9.                  remove testConstraint from testCoMSS
10.     For all testCoMSS ∈ CoMSSes
11.         If testCoMSS contains newConstraint
12.             remove testCoMSS from CoMSSes
13. Return MUS
```

Figure 3: Extracting a single MUS from CoMSSes(C)

4.2 Obtaining MUSes(C)

Every MUS of C is an irreducible hitting set of CoMSSes(C). The decision version of the HITTING-SET problem (deciding if there exists a hitting set of cardinality less than or equal to k) is NP-Complete [Karp, 1972], but in this case we are interested in irreducibility, not cardinality. Combined with the property that no CoMSS is a subset of any other, this allows us to extract a single MUS via a greedy approach in polynomial time, without resorting to search. Figure 3 provides pseudocode for the algorithm.

The construction works by sequentially adding constraints to a forming MUS. Intuitively, an MUS is a set of constraints containing at least one constraint from every CoMSS such that every constraint is an essential element of the set. By "essential" we mean that removing a constraint from the MUS will leave at least one CoMSS unrepresented. Thus, every time a constraint is added, the remaining problem is modified to enforce the requirement that the constraint be essential. This is done by ensuring that the new constraint is the only representative of at least one CoMSS.

Lines 3 and 4 of **Musilitis-ExtractMUS** choose a CoMSS from the set and a single constraint, *newConstraint*, from that CoMSS. *newConstraint* is added to the MUS, and the remaining lines ensure that it is essential. Lines 6 through 9 remove the remaining constraints in the chosen CoMSS from all other CoMSSes in the input set. This prevents any of those constraints from being added in later iterations, which could make *newConstraint* non-essential. Then, lines 10-12 remove any CoMSSes containing *newConstraint*, because they are now represented in *MUS*. Following the modifications to

CoMSSes, the algorithm iterates. When *CoMSSes* is empty, the constructed set of constraints is a complete, exact MUS.

Extracting all of the MUSes from CoMSSes(C) requires searching for all irreducible hitting sets of CoMSSes(C). Due to the potential combinatorial explosion, the number of MUSes may be impractically large; however, in many cases the result is tractable.

One method for generating the complete set of MUSes from CoMSSes(C) uses the general form of the **Musilitis-ExtractMUS** algorithm in Figure 3. The particular MUS generated by that algorithm depends on the order in which CoMSSes and constraints are selected from CoMSSes(C); thus, by branching on those two decisions (lines 3 and 4), all possible MUSes can be generated. This can be accomplished with a recursive function that takes as input a) the remaining set of CoMSSes and b) the MUS currently being constructed in each branch of the recursion (these are initialized to CoMSSes(C) and the empty set, respectively). The branching is not ideal, and duplicate branches are both possible and quite common in practice, so pruning heuristics can be employed to reduce the search space.

5 Experimental Results

We benchmarked both phases of the MUS generation process (finding CoMSSes(C) and extracting MUSes(C) from it) using DTPs created by a random DTP generator used in testing previous DTP solvers [Stergiou and Koubarakis, 1998]. We collected performance data and analyzed the sets of CoMSSes and MUSes for DTPs over a range of DTP generator parameters.

The test case generator takes as arguments the parameters $\langle k, N, m, L\rangle$, where k is the number of disjuncts per constraint, N is the number of time points, m is the number of constraints, and L is the constraint width, i.e., a positive integer such that for each disjunct $x - y \leq b$, $b \in [-L, L]$ with uniform probability. In our experiments, we set $k = 2$, $N \in \{5, 6, 7\}$, and $L = 100$. A derived parameter R (the ratio of constraints over variables, m/N) was varied from 6 to 11. For each setting of N and R, 50 random problems were generated. (For example, we generated 50 problems for the case where N is 7 and R is 7; those problems have 7 temporal variables and 49 constraints each). These problems are likely to be highly overconstrained, with 30-77 constraints on only 5-7 time points. The size of these problems is dwarfed somewhat by those which Maxilitis can solve (e.g., N = 20), although one must keep in mind that the objective of maximal constraint satisfaction is much easier to achieve. The domains of the variables are integers instead of reals, which again is standard in DTP literature. Our implementation of Musilitis was developed in Java (finding CoMSSes) and C++ (extracting MUSes), and our experiments were conducted in Linux on a 2.2GHz Opteron processor with 8GB of RAM.

First, we compared the runtime of the two versions of the **Musilitis-CoMSS** algorithm described in Section 4.1—one which uses a single backtracking search to find all CoMSSes, and another which does so incrementally. The former requires an additional postprocessing step to remove all redundant CoMSSes, a procedure whose runtime was found to be negligible. Figure 4 shows the median runtimes for both algorithms on all problem parameters. Not surprisingly, runtime increases both with size (N) and density (R). However, notice that the times for the single backtracking algorithm (labeled 'Backtracking') tend to be roughly an order of magnitude faster than those for the incremental method (labeled 'Incremental'). This contrasts with the results reported in [Moffitt and Pollack, 2005], where the opposite effect was observed. The difference is that Maxilitis can stop at any iteration where a solution is found, as only a single solution of maximal cardinality is needed. Even though multiple searches are performed in the incremental mode, substantially more is pruned than in a single search that has no upper bound. In our case, iteration must continue until *all* CoMSSes of any size have been found. The combined runtime of these separate searches can easily exceed that of a single backtracking search.

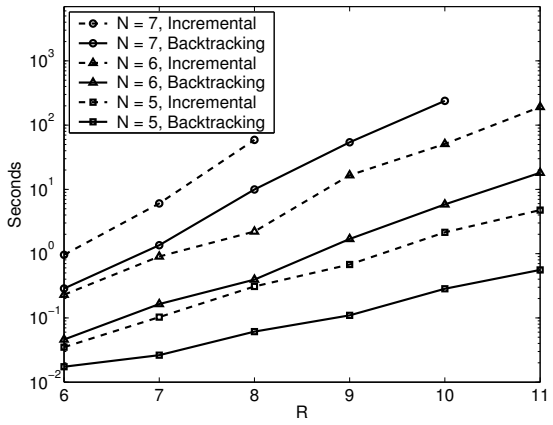

Figure 4: Median runtimes for finding CoMSSes(C)

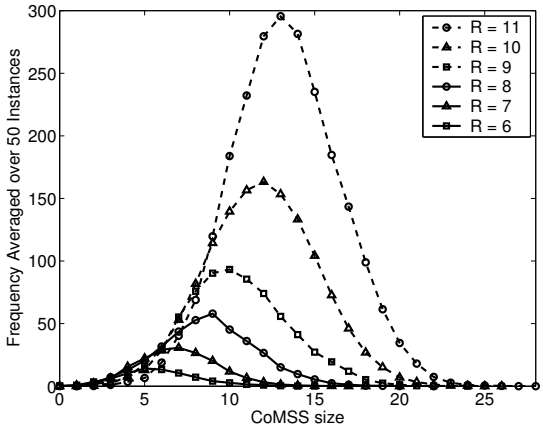

Figure 5: Distribution of CoMSS sizes ($N = 6$)

Figure 5 shows the distribution of sizes of CoMSSes found by the **Musilitis-CoMSS** algorithm for the cases where $N = 6$ and R ranges from 6 to 11. Because our DTPs were randomly generated, these results are not particularly useful alone; however, they do provide valuable information when one considers the runtime of the MUS extraction phase of

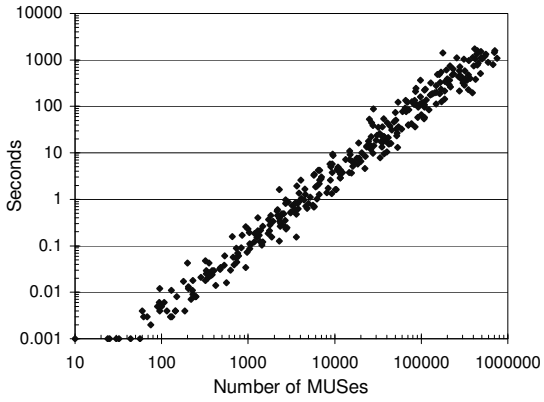

Figure 6: Runtimes of extracting MUSes(C) vs $|\,\mathrm{MUSes}(C)|$

the Musilitis algorithm, since the set of CoMSSes is the input to this procedure. In the graph, we have taken the total frequency of each size of CoMSS found in all instances and divided this count by 50 (the number of problem instances). As one would expect, both the number and size of CoMSSes increases with the constraint density R.

The runtime of extracting all MUSes depends mainly on the number of MUSes present. In the extremely overconstrained benchmarks used here, this number is often very high, reaching into the hundreds of thousands. Figure 6 shows the runtime of the MUS extraction algorithm versus the number of MUSes found for instances with $N = 5$. Runtimes for any given number of MUSes are clustered within an order of magnitude of each other.

In some benchmarks, the MUS extraction reached a 30-minute timeout without finishing. In that time, however, it found as many as one million MUSes. The algorithm for extracting MUSes is inherently an anytime algorithm; MUSes are output as they are found, and if not all MUSes are required, it can be stopped at any time. Our experiments showed that the algorithm has reasonably good anytime properties: in one typical highly-constrained instance with $N = 6$ and $R = 9$, it generated more than 70,000 MUSes in the first minute, and the rate then gradually slowed. This property could be exploited in a system that interleaves MUS identification with constraint relaxation. Generally, if the constraint system is highly overconstrained, resolving one MUS is likely to resolve many others, since the same constraints are likely to appear in multiple MUSes.

6 Conclusions and Future Work

We have presented algorithms for deriving minimally unsatisfiable subsets of constraints in any infeasible constraint set, building on a strong relationship between maximal satisfiability and minimal unsatisfiability. The techniques are general, in that they are applicable to any type of constraint, and we have implemented them for Disjunctive Temporal Problems. The flexibility permitted by the constraints involved in these problems presents the opportunity for relaxation to be more fine-grained than simple constraint elimination, and MUSes provide the information needed to support this type of relaxation.

The connection between maximal satisfiability and minimal unsatisfiability we describe has been independently noted by [Bailey and Stuckey, 2005], who applied it to the problem of type-error diagnosis in software verification. Their implementation, however, differs from ours, in that they employ different algorithms in an interleaved approach as compared to our serial, two-phase algorithm. In [Liffiton and Sakallah, 2005], we performed an experimental comparison of their approach, adapted to Boolean satisfiability, with our own. We found that ours was consistently faster, and we identified several ways to combine ideas from both techniques for further improvements.

Further work also includes enhancing the efficiency of our algorithms. One possible direction would be to investigate tradeoffs between runtime and completeness or correctness. The requirements of finding all MUSes or of finding exact MUSes could be relaxed to decrease the runtime of the algorithms. Techniques exploiting domain knowledge or features of particular types of constraints could help improve the efficiency as well. More generally, it will be important to conduct studies that examine the performance of Musilitis on DTPs representing real-world problems, as opposed to the random problems studied in this paper. We currently have experiments underway to evaluate Musilitis on job-shop scheduling problems, and we will also be studying problems taken from the domains of schedule management and medical protocols.

The relationship between maximal satisfiability and minimal unsatisfiability is deserving of further attention. Our algorithms for extracting all MUSes are just one way of exploiting this relationship. Further understanding of infeasibility in constraint satisfaction problems, such as rich links between conflicts in basic constraints and in the high-level problem they represent, could result from additional research.

Finally, an important future goal is to employ the algorithms developed in this paper in a mixed-initiative constraint relaxation system that provides a user with the information needed to make principled decisions about how to weaken constraints so as to achieve feasibility.

Acknowledgments

This material is based upon work supported by the National Science Foundation under ITR Grant No. 0205288, the Defense Advanced Research Projects Agency (DARPA) under Contract No. NBCHD030010, and the Air Force Office of Scientific Research under Contract No. FA9550-04-1-0043. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation (NSF), DARPA, the Department of Interior-National Business Center, or the United States Air Force.

References

[Armando *et al.*, 1999] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *Proceedings of the 5th European Conference on Planning*, pages 97–108, 1999.

[Bailey and Stuckey, 2005] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of

constraints using hitting set dualization. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[Bruni and Sassano, 2001] Renato Bruni and Antonio Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, Electronic Notes in Discrete Mathematics volume 9. Elsevier Science Pub., 2001.

[Chinneck, 1996] John W. Chinneck. An effective polynomial-time heuristic for the minimum-cardinality IIS set-covering problem. *Annals of Mathematics and Artificial Intelligence*, 17:127–144, 1996.

[Dechter *et al.*, 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[Garey and Johnson, 1990] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[Karp, 1972] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Liffiton and Sakallah, 2005] Mark H. Liffiton and Karem A. Sakallah. On finding all minimally unsatisfiable subformulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[Moffitt and Pollack, 2005] Michael D. Moffitt and Martha E. Pollack. Partial constraint satisfaction of disjunctive temporal problems. In *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2005)*, 2005.

[Oddi and Cesta, 2000] Angelo Oddi and Amedeo Cesta. Incremental forward checking for the disjunctive temporal problem. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 108–112, 2000.

[Oh *et al.*, 2004] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Proceedings of the 41st annual conference on Design automation (DAC'04)*, pages 518–523. ACM Press, 2004.

[Stergiou and Koubarakis, 1998] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 248–253, 1998.

[Tsamardinos and Pollack, 2003] Ioannis Tsamardinos and Martha E. Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1-2):43–90, 2003.

[Zhang and Malik, 2003] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formula. Presented at the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), 2003.