Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Artificial Intelligence 151 (2003) 43–89
Artificial Intelligence
www.elsevier.com/locate/artint

Efficient solution techniques for disjunctive temporal reasoning problems

Ioannis Tsamardinos [a,*], Martha E. Pollack [b]

[a] *Vanderbilt University, Department of Biomedical Informatics, Room 444, Eskind Biomedical Library, 2209 Garland Ave, Nashville, TN 37212-8340, USA*

[b] *Computer Science and Engineering, University of Michigan, USA*

Received 7 September 2001; received in revised form 10 February 2003

Abstract

Over the past few years, a new constraint-based formalism for temporal reasoning has been developed to represent and reason about Disjunctive Temporal Problems (DTPs). The class of DTPs is significantly more expressive than other problems previously studied in constraint-based temporal reasoning. In this paper we present a new algorithm for DTP solving, called Epilitis, which integrates strategies for efficient DTP solving from the previous literature, including conflict-directed backjumping, removal of subsumed variables, and semantic branching, and further adds no-good recording as a central technique. We discuss the theoretical and technical issues that arise in successfully integrating this range of strategies with one another and with no-good recording in the context of DTP solving. Using an implementation of Epilitis, we explore the effectiveness of various combinations of strategies for solving DTPs, and based on this analysis we demonstrate that Epilitis can achieve a nearly two order-of-magnitude speed-up over the previously published algorithms on benchmark problems in the DTP literature.

Keywords: Constraint-based temporal reasoning; Constraint satisfaction; Scheduling; Planning

1. Introduction

Expressive and efficient temporal reasoning is essential to a number of areas in Artificial Intelligence (AI). Over the past few years, a new constraint-based formalism for temporal reasoning has been developed to represent and reason about Disjunctive

* Corresponding author.
E-mail addresses: ioannis.tsamardinos@vanderbilt.edu (I. Tsamardinos), pollackm@eecs.umich.edu (M.E. Pollack).

0004-3702/$ – see front matter
doi:10.1016/S0004-3702(03)00113-9

Temporal Problems (DTPs) [1,20,29]. The class of DTPs is significantly more expressive than other problems already studied in constraint-based temporal reasoning. It extends the well-known Simple Temporal Problem (STP) [13] by allowing disjunctions and the Temporal Constraint Satisfaction Problem (TCSP) *ibid.* by removing restrictions on the form of allowable disjunctions.

Formally, a Disjunctive Temporal Problem (DTP) is a pair $\langle V, C\rangle$ where V is a set of temporal variables and C is a set of constraints among the variables. Every constraint $C_i \in C$ is of the form:

$$c_{i1} \vee \cdots \vee c_{in}$$

where in turn, each c_{ij} is of the form $x - y \leqslant b$; $x, y \in V$ and $b \in \mathbb{R}$.[1] Each c_{ij} is called the jth *disjunct* of the ith constraint. A solution to a DTP is an assignment to each variable in V such that all the constraints in C are satisfied. If a DTP has at least one solution, it is *consistent*. Notice that each constraint C_i may involve more than two temporal variables, in which case it is not a binary constraint. Only DTPs, and not STPs or TCSPs, allow difference constraints of arbitrary arity.

The increased expressivity of DTPs makes them a suitable model for many planning and scheduling problems. Plan generation, plan merging, job-shop scheduling, and even temporal reasoning under certain forms of uncertainty can all be modeled as DTPs. As a motivating example, consider a temporal plan with steps A and B that both represent actions requiring the same unary resource, e.g., they both use the same printer. In this case, the executions of A and B should not overlap. We can encode this fact as a DTP constraint by defining the DTP variables $start(A)$, $start(B)$, $end(A)$, and $end(B)$ associated with the instants of starting and ending A and B. The DTP constraint then is the following:

$$end(A) - start(B) \leqslant 0 \vee end(B) - start(A) \leqslant 0,$$

i.e., either A finishes before B starts or vice versa. It is easy to see how such constraints can encode classical threat resolution in planning. Analysis of the DTP defining this plan can reveal whether it is feasible: the DTP is consistent if and only if there is a way to execute the plan such that the deadlines are all met, and the unary resource (the printer) is never used more than once at the same time.

Threat-resolution constraints in planning, as just described, can also be encoded as binary constraints between intervals. There are situations however when this is not possible and higher arity constraints are required and thus cannot be expressed (in the general case) by any other current formalism but the DTP. For example consider reasoning about the following scenario: "if you can, stop by the post-office for 10–15 minutes, then take route A for 10–15 minutes, or else take route B for 10–15 minutes". If we use PO to denote the fluent 'in the post-office', then this scenario can be represented as the following constraints:

$$\big(10 \leqslant end(PO) - start(PO) \leqslant 15 \wedge 10 \leqslant end(A) - start(B) \leqslant 15 \wedge end(PO) = start(A)\big) \vee 10 \leqslant end(B) - start(A) \leqslant 15.$$

In turn, this can be directly converted to DTP form.

[1] As is standard in the literature, in this paper we will make the assumption, without loss of generality, that the bounding values b are integers.

The principal approach to DTP solving taken in the literature has been to convert the original problem to one of selecting one disjunct, $x_i - x_j \leqslant b_{ji}$ from each constraint $C_i \in C$ and then checking that the set of selected disjuncts forms a consistent STP. Checking the consistency of and finding a solution to an STP can be performed in polynomial time using shortest-path algorithms [13]. The computational complexity in DTP solving derives from fact that there are exponentially many sets of selected disjuncts that may need to be considered; the challenge is to find ways to efficiently explore the space of disjunct combinations. This has been done by casting the disjunct selection problem as a constraint satisfaction processing (CSP) problem [20,30] or a satisfiability (SAT) problem [1].

In this paper, we present a new algorithm and heuristics for DTP solving, embodied in a system we call *Epilitis*,[2] which integrates strategies for efficient DTP solving from the previous literature, including conflict-directed backjumping, removal of subsumed variables, and semantic branching, and further adds no-good recording [27] as a central technique. We discuss the theoretical and technical issues that arise in successfully integrating the previous strategies with one another and with no-good recording. Using an implementation of Epilitis, we explore the effectiveness of various combinations of strategies for solving DTPs, and based on this analysis, we demonstrate that Epilitis achieves a nearly two-order-of-magnitude speed-up over the previously published algorithms on benchmark problems from the DTP literature. This result is based on speed comparisons because we demonstrate that counting the number of forward-checks (also called consistency-checks), as is commonly done in the literature, is not an accurately descriptive measure of performance.

The overall structure of the paper is as follows. In Section 2 we present necessary background information on the DTP and DTP solving. Section 3 explains all previous methods for pruning the search for a DTP solution. Section 4 presents necessary background information on no-good recording. Section 5 uses the background material presented to describe the Epilitis system and the underlying algorithms we used. In Section 6 we present our experiments with Epilitis. Section 7 reviews related work in the field. Section 8 concludes the paper with an overall discussion and presentation of future work.

2. Solving disjunctive temporal problems

2.1. The basic approach

We begin by reviewing a simpler class of temporal problems: the Simple Temporal Problems (STPs). An STP, like a DTP, is a pair $\langle V, C\rangle$, where V is again a set of temporal variables; for an STP, however, C is a single constraint of the form $x - y \leqslant b$, where $x, y \in V$. Because an STP contains only binary constraints, it can be represented with a weighted graph called a Simple Temporal Network (STN), in which an edge (y, x) with weight b_{yx} exists between two nodes *iff* there is a constraint $\{x - y \leqslant b_{yx}\} \in C$. Polynomial-time algorithms can be used to compute the all-pairs shortest path matrix, or

[2] From the Greek word $E\pi\iota\lambda\upsilon\tau\acute{\eta}\varsigma$ (solver).

distance array of the STN. We denote the distance (shortest path) between two nodes x and y as d_{xy}. The concept of the distance is important because in a consistent STP, d_{xy} is the largest number for which the constraint $y - x \leqslant d_{xy}$ holds in every solution. In addition, an STP is consistent if and only if for every node x in its associated STN, $d_{xx} \geqslant 0$, which means that there are no negative cycles [13].

A DTP can be viewed as encoding a collection of alternative STPs. To see this, recall that each constraint in a DTP is a disjunction of (one or more) STP-style inequalities. Let c_{ij} be the jth disjunct of the ith constraint of the DTP. If we select one disjunct c_{ij} from each constraint C_i, then the set of selected disjuncts forms an STP, which we will call a *component STP* of the given DTP. It is easy see that a DTP D is consistent if and only if it contains at least one consistent component STP. Moreover, any solution to a consistent component STP of D is also a solution to D itself. Because only polynomial time is required both to check the consistency of an STP, and, if consistent, extract a solution of it, in the remainder of this paper we will say that *the solution of a given DTP is any consistent component STP of it*. When we need to refer to an actual assignment of numbers to the time-points in the DTP, we will call this an *exact solution*. A consistent component STP represents a number of exact DTP solutions. This is particularly important in planning since it provides execution flexibility. The consistent component STP can then be executed as described in [31].

Definition 1. A time assignment to the time-points of a DTP is called *an exact solution* of the DTP. A consistent component of a DTP is called a *solution* of the DTP.

All existing algorithms for DTP solving, including the one we present in this paper, work by searching for a consistent component STP S from a given DTP D rather than attempting to search directly for a consistent assignment to the nodes of D. The process of finding S can itself be modeled as one of constraint satisfaction processing. Because the original DTP is itself also a CSP problem, we will refer to the problem of extracting a consistent component STP as the *meta-CSP problem.* The meta-CSP contains one variable for each constraint C_i in the DTP. The domain of C_i is the set of disjuncts in the original DTP constraint C_i. The constraints in the meta-CSP are not given explicitly, but must be inferred: an assignment satisfies the meta-CSP constraints *iff* the assignment corresponds to a component STP that is consistent. For instance, if the variable C_i is assigned the value $x - y \leqslant 5$ it would be inconsistent to extend that assignment so that some other variable C_j is assigned the value $y - x \leqslant -6$.

In this paper, we will refer to the variables of the DTP as *time-points* and will reserve the term *variables* for the meta-CSP. We will use the terms *constraint* and *value* interchangeably, to refer to a single, non-disjunctive constraint c_{ij}: such constraints (values) constitute the domains of the meta-CSP variables. Finally, we will reserve term *node* to refer to the nodes of a CSP tree search, which we will typically be performing for the meta-CSP—recall that we do not perform direct CSP processing on the DTP. The relationship between the original CSP (the DTP) and the meta-CSP (which aims to find a consistent component STP) is summarized in Table 1.

A typical forward-checking CSP algorithm, shown in Fig. 1 can be used to solve a DTP—or more precisely, to solve its meta-CSP. The algorithm takes two parameters: A,

Table 1
Correspondence between the DTP and the meta-CSP

	Original CSP (the DTP)	Meta-CSP
"Variables"	$x, y, z, \ldots$ (Time points)	One variable C_i for each constraint C_i of the original DTP (Variables)
Domains	$(-\infty, +\infty)$ for all variables	$D(C_i) = \{c_{i1}, \ldots, c_{im}\}$ (Sets of constraints)
"Constraints"	$x_1 - y_1 \leqslant b_1 \vee \cdots \vee x_n - y_n \leqslant b_n$, i.e., $c_{i1} \vee \cdots \vee c_{in}$ (Disjunctions of constraints)	Implicitly defined by the underlying semantics of the values in each domain

Basic-DTP(A, U)
1. If $U = \emptyset$ stop and report A as a solution.
2. $C \leftarrow$ **select-variable**(U), $U' \leftarrow U$-$\{C\}$
3. For each value c of *d(C)* in some order
4. $A' = A \cup \{C \leftarrow c\}$
5. If **forward-check**(A', U')
6. **Basic-DTP**(A', U')
7. EndIf
8. **un-forward**(U')
9. EndFor
10. Return *failure*

forward-check(A, U)
11. For each variable C in U
12. For each value c in $d(C)$
13. If not **STP-consistency-check**$(A \cup \{C \leftarrow c\})$
14. Remove c from $d(C)$
15. If $d(C) = \emptyset$
16. return *false*
17. EndIf
18. EndIf
19. EndFor
20. Return *true*

Fig. 1. The Basic DTP algorithm.

denoting the set of already assigned variables and their assigned values, and U, the set of as-yet unassigned variables. The initial call to solve a DTP $\langle V, C\rangle$ should be made with $A = \emptyset$ and $U = C$, and the initial *current domains* $d(C)$ should be initialized to the *original domains* $D(C_i)$, i.e., to the set of constraints that constitute the disjuncts in C_i in the DTP (see Table 1).

The function **select-variable** heuristically selects the next variable to which to make an assignment; the decision about how to make that selection is left unspecified in the generic algorithm, but we discuss it further in Section 6.4. The function **forward-check**(A, U) performs forward-checking, i.e., it removes from the domains of the variables still in U all those values that are inconsistent with the current assignment A, returning *false* if, as a result, one or more variables in U has a domain reduced to $\emptyset$. Note that in DTP-

solving, **forward-check**[3] operates by checking the consistency of an STP (specifically, a component STP of the DTP), which, as mentioned above, requires only polynomial time. If forward-checking fails, then the function **un-forward** restores the domains of the variables to those before the last call to **forward-check**.

2.2. Improved forward-checking

A large portion of the computation in algorithm **Basic-DTP** is spent at line 13 in **forward-check,** where each value of each variable is added to the set of constraints of the current assignment A and checked for STP-consistency. STP-consistency checking takes time $O(|V|^3)$ where $|V|$ is the number of time-points; thus forward-check takes time $O(v|V|^3)$ on each node, where v is the number of values to forward-check. Fortunately, there is a computationally less costly way of achieving forward-checking of values, based on the following theorem:[4]

Theorem 1. *A value c_{ij}: $y - x \leqslant b_{xy}$ is inconsistent with a consistent STP S (that is, $S \cup c_{ij}$ is inconsistent) if and only if the following condition holds*:

$$b_{xy} + d_{yx}(S) < 0 \quad (\textit{FC-condition})$$

where $d_{yx}(S)$ is the distance between nodes y and x in S.

Theorem 1 (the proof is in Appendix A) indicates that to forward-check a particular value $y - x \leqslant b_{xy}$ against an assignment A, we just need to check the FC-condition. In turn, this requires calculating the distances d_{yx} in STP S for all nodes x and y. One method for calculating all these distances efficiently is to calculate the distance array; this is equivalent to running full path consistency, which has time complexity time $O(|V|^3)$. Once the distance array has been calculated, the distance between any two nodes y and x can be recovered by matrix lookup in constant time; hence the overall time required for each node is $O(|V|^3 + v)$, where v is the number of values to be forward-checked. An alternative technique is to compute directional path consistency [9] where only part of the shortest path array is cached, in a manner that permits the uncached distances to be recovered in time at most $O(|V|)$.

To modify the main algorithm in Fig. 1 with improved forward-checking, we only need do two things. First, we replace the forward-checking routine with one that uses the FC-condition, as shown in Fig. 2. Second, we add one line to the main program (Basic-DTP); specifically,

$$S' = \textbf{maintain-consistency}(c, S)$$

should be inserted between lines 4 and 5. Each time a new variable is assigned a value $\{C \leftarrow c\}$, the constraint is propagated in S by **maintain-consistency**(c, S), which can

[3] We only describe DTP solvers using forward-check because it has been proven very efficient in DTP literature and it is a standard component of every DTP solver. In addition, not using forward-checking dramatically reduced efficiency in preliminary experiments in our lab.

[4] This theorem was suggested, but not proved, in [20]; see Appendix A for its proof.

```
forward-check-with-FC(S, U)
11.  For each variable C in U
12.    For each value c: x − y ⩽ b_xy in d(C)
13.    If b_xy + distance(y, x, S) < 0
14.      Remove c from d(C)
15.      If d(C) = ∅
16.        return false
17.      EndIf
18.    EndIf
19.  EndFor
20.  Return true
```

Fig. 2. Forward-checking in STPs using the FC condition.

be implemented with either full path consistency or with directional path consistency, as described above.

The comparison of overall complexity in each node does not yield an obvious "best" approach. As already noted, (i) the basic **forward-check** procedure requires $O(v|V|^3)$ time for each node in the CSP search tree; (ii) computing full path consistency and checking the FC-condition requires $O(|V|^3 + v)$; and (iii) computing directional path consistency for FC-checking requires $O(|V|^3 + v|V|)$. In addition, since assignment A is built incrementally by adding constraints on each new node, we can use incremental versions of these previous techniques to build S, namely *incremental full path consistency* (IFPC) [19] and *incremental directional path consistency* (IDPC) [9]. The incremental versions drop the exponent in all the above complexities to quadratic and so (i) takes time $O(v|V|^2)$, (ii) takes time $O(|V|^2 + v)$ and (iii) $O(|V|^2 + v|V|)$. Thus, the worst-case comparison favors maintaining full path consistency (i.e., the distance array).[5] The average case comparison cannot easily be resolved theoretically and further experiments are required to determine under which conditions each method is the best. In our experiments, reported below in Section 6 we used method (ii), maintaining the distance array at every node.

3. Previous pruning techniques for DTP solving

Once the DTP problem has been cast as one of solving a meta-CSP, a number of different backtracking search techniques can be used to increase efficiency by early pruning of dead-end branches. The idea in pruning techniques is to utilize the underlying semantics of the values of the meta-CSP, namely the fact that they express constraints on some STP, to make inferences regarding the infeasibility of certain regions of the search. In this section we describe three methods previously considered in the literature: *Conflict-Directed Backjumping* (CDB) (used in [30]), *Semantic Branching* (SB) (used in [20] and [1]), and *Removal of Subsumed Variables* (RSV) (used in [1]). In the next section we will add *No-good Recording* (NR) (also called no-good learning), and hence throughout both

[5] This is because the worst-case bounds are tight.

these sections we will be particularly concerned with the theoretical and technical issues that arise in successfully integrating these pruning strategies with one another and with no-good recording.

3.1. Conflict-directed backjumping for DTPs

The simplest algorithms for solving CSP problems rely on chronological backtracking, in which the failure of a partial assignment of values to variables results in backtracking to the point in the search just before the most recent assignment of a value to a variable was made. Previous work has shown that backtracking can be made more efficient by instead restarting the search at a more carefully selected point: techniques developed for this include Dynamic Backtracking [15], and Conflict Directed Backjumping (CDB) [6, 23]. In these approaches, when a dead end is encountered, the search backtracks to the most recently assigned variable that is *related* to the failure. The variables that are unrelated to the failure are backjumped over, since trying to assign different values for them will result in the same dead end.

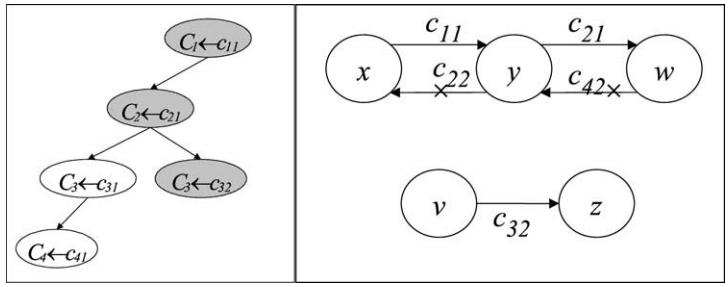
It is obvious that to implement CDB, it is necessary to be able to identify the culprit of the failures, i.e., the variables that participate in the constraints that lead to failure. Stergiou and Koubarakis [30] present a method for culprit identification in DTP solving, which they call the *dependency pointers scheme*. This scheme is based on the fact that, during DTP solving, whenever an assignment A is extended to $A' = A \cup \{C \leftarrow c\}$, forward-checking is performed. If a value c' is removed from some domain, then the most recent value assignment, $\{C \leftarrow c\}$, must directly contribute to its removal. In the approach of Stergiou and Koubarakis, a dependency pointer from c' to c is stored. If the algorithm subsequently needs to backtrack because the domain of some variable has been reduced to $\emptyset$, the algorithm checks the dependency pointers for values that were removed from that variable's domain, and follows the one that points to the most recently assigned variable, thereby backjumping over any irrelevant variables.

Although the dependency pointer scheme achieves CDB, it does not integrate well with semantic branching and no-good recording. We thus developed an alternative scheme for calculating the culprit of a failure. Our technique returns the variables of the current assignment that are involved in the failure; these can be used in a manner similar to dependency pointers, to backjump to the most recent relevant variable. Additionally, however, the returned set of variables can be used as justifications in no-good recording, as described in Section 4.

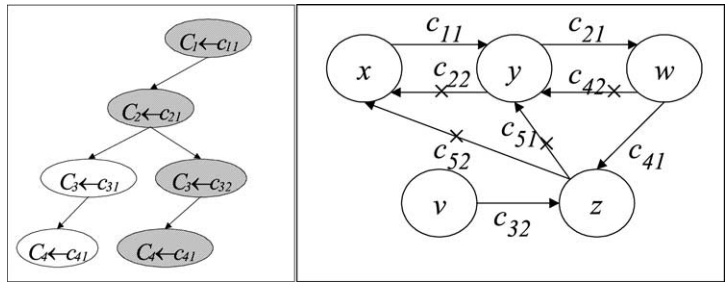
Our approach is straightforward, and builds directly on the fact that backtracking is required only when **forward-check** has reduced the domain of some variable to the empty set by removing every value c_j of that domain. This in turn implies that every c_i that was in the domain is part of a negative cycle p_i formed by constraints $c_1, \ldots, c_k$. We introduce the technique with an example.

Example 1. Fig. 3 illustrates the processing of the following DTP:

$$C_1\colon \{c_{11}\colon\ y - x \leqslant 5\},$$
$$C_2\colon \{c_{21}\colon\ w - y \leqslant 5\} \vee \{c_{22}\colon\ x - y \leqslant -10\} \vee \{c_{23}\colon\ z - y \leqslant 5\},$$



(a)



(b)

Fig. 3. The chronological backtracking algorithm on a DTP.

C_3: $\{c_{31}\colon v - x \leqslant 5\} \vee \{c_{32}\colon z - v \leqslant 10\}$,

C_4: $\{c_{41}\colon z - w \leqslant 5\} \vee \{c_{42}\colon y - w \leqslant -10\}$,

C_5: $\{c_{51}\colon y - z \leqslant -20\} \vee \{c_{52}\colon x - z \leqslant -20\}$.

In the figure, the top two boxes (a) represent a "snapshot" of the DTP solving process: the left-hand side shows the meta-CSP search tree, and the right-hand side shows the STP entailed by the current assignment. The bottom two boxes (b) show a snapshot later in the process. At the time of Fig. 3(a), assignments have been made to C_1, C_2, and C_3. The assignments chosen are indicated by the gray ovals while the white ovals indicate already explored nodes. Note that values that have been ruled out by forward-checking are crossed out in the STP diagrams. For instance, the assignment of c_{11}: $(y - x \leqslant 5)$ to C_1 rules out the possibility of assigning c_{22}: $(x - y \leqslant -10)$ to C_2, and so this value is crossed out in the right-hand part of Fig. 3(a).

Fig. 3(b) shows a later point in the processing, by which an assignment has also been made to C_4 (specifically $C_4 \leftarrow c_{41}$). At this point, forward-checking will eliminate both possible values for C_5, because they participate in negative cycles. These cycles are independent of the assignment made to C_3, which should thus be backjumped over. That is, c_{51} and c_{52} are removed from $d(C_5)$ because they form the negative cycles (in the STP): $p_1 = (c_{21}, c_{41}, c_{51})$ and $p_2 = (c_{11}, c_{21}, c_{41}, c_{52})$. The variables that participate in the failure then are $\mathbf{vars}(p_1) \cup \mathbf{vars}(p_2) = \{C_2, C_4, C_5\} \cup \{C_1, C_2, C_4, C_5\} = \{C_1, C_2, C_4, C_5\}$, where $\mathbf{vars}(p)$ are the variables whose value assignments are the constraints in p.

justification-value(c: $y - x \leqslant b$, S)
1. $p =$ **shortest-path**(y, x, S)
2. Return **vars**$(p \cup c)$

Fig. 4. Function justification-value.

It is apparent that our technique requires the identification of a negative cycle for each removed value by forward-check. This can be implemented by maintaining a predecessor array[6] [10] when calculating the shortest path array. Entry $\langle i, j \rangle$ of the predecessor array contains nil when $i = j$; otherwise it is a predecessor of j on the shortest path from i. It should be updated by the function **maintain-consistency**, which can be done without changing the time complexity of the function. When a value c: $y - x \leqslant b$ completes a negative cycle (i.e., the FC-condition holds), we follow the predecessor array to retrieve the shortest path p from y to x and return **vars**$(p \cup (x, y))$, where (x, y) is the edge from x to y. The pseudo-code for implementing this approach is given in Fig. 4: the function **justification-value** returns the justification (i.e., the culprit set of variables) for the removal of value c: $y - x \leqslant b$ from the domain of its variable, given an STP S that corresponds to the current assignment.

3.2. Removal of subsumed variables

The main idea of the heuristic that we will call *Removal of Subsumed Variables* (RSV) is that if a disjunct c_{ij} of a variable C_i is already satisfied by the current assignment A, there is no reason to try other values in the variable's domain under assignment A because either (i) the current assignment leads to a solution, and since c_{ij} is already satisfied under A, C_i is satisfied in the solution, or (ii) there are no solutions under A and trying other values for C_i will only restrict A even further, with no possibility of discovering a solution. We now proceed by formalizing this idea.[7]

Definition 2. *A value c_{ij} is subsumed by an STP S* (equivalently by an assignment A that implies S) if and only if the constraint c_{ij} always holds in any exact solution of S. (Recall that an exact solution to a DTP D is an assignment of numbers to the time-points in D.) A *variable C_i is subsumed by an STP S* if and only if there is a value c_{ij} in the domain of C_i that is subsumed by S.

Theorem 2. *A value c_{ij}: $y - x \leqslant b$ is subsumed by an STP S if and only if $d_{xy}(S) \leqslant b$* (Subsumption-Condition)*, where $d_{xy}(S)$ is the distance between x and y in S.*

Theorem 3. *Let $D = \langle V, C \rangle$ be a DTP, let A be an assignment on D (i.e., a component STP), and let C_i be a variable subsumed by A. Then A is a solution of D if and only if it is a solution of $D' = \langle V, C - C_i \rangle$.*

[6] Recall that the predecessor array stores a predecessor of j on the shortest path from i in all entries $\langle i, j \rangle$ that are not on the main diagonal.

[7] RSV was first used by [20] but without providing a proof.

Corollary 1. *Let A be a partial assignment during a DTP search, U be the unassigned variables, and Sub be the set of subsumed variables in U. If A can be extended to a solution over variables in $U - Sub$, it can be extended to a solution over variables in U. In other words, we can remove the subsumed variables from the unassigned variables during search. The solution to the reduced problem is a solution to the original one.*

The proofs for the above theorems and corollaries are in Appendix A.

Example 2. The ramifications of the above corollary are shown pictorially Fig. 6, which depicts a search *without* the use of RSV for a solution to the DTP in Fig. 5. The variables are considered in order (1–6). Initially, $A_1 = \{C_1 \leftarrow c_{11}\}$. This is then extended to $A_2 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}\}$. Without removing the subsumed variables, the next assignment would be $A_3 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}\}$. Notice though that value c_{31}, and thus variable C_3 is subsumed by A_2, because together c_{11}: $y - x \leqslant 5$ and c_{21}: $x - z \leqslant 5$ imply that $y - x \leqslant 10$, which subsumes the constraint c_{31}: $y - z \leqslant 15$. Thus, by Corollary 1, C_3 can safely be removed from the search underneath the subtree of A_2. Suppose, however, that it is not removed. Then the search will proceed as in Fig. 6. When the search of subtree T_1 in figure fails, as it will in this particular example, the search continues by trying the other value of C_3 and so $A_4 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{32}\}$. By Corollary 1, since A_3 has failed, A_4 will fail too. By removing the subsumed variable C_3 in this particular

C_1: $\{c_{11}$: $y - x \leqslant 5\} \vee \{c_{12}$: $w - y \leqslant -10\}$
C_2: $\{c_{21}$: $x - z \leqslant 5\}$
C_3: $\{c_{31}$: $y - z \leqslant 15\} \vee \{c_{32}$: $z - v \leqslant 10\}$
C_4: $\{c_{41}$: $z - v \leqslant 5\} \vee \{c_{42}$: $y - w \leqslant -10\}$
C_5: $\{c_{51}$: $v - y \leqslant -20\} \vee \{c_{52}$: $z - x \leqslant -10\}$
C_6: $\{c_{61}$: $z - v \leqslant 2\} \vee \{c_{62}$: $x - y \leqslant -10\}$

Fig. 5. Example DTP for removal of subsumed variables and semantic branching.

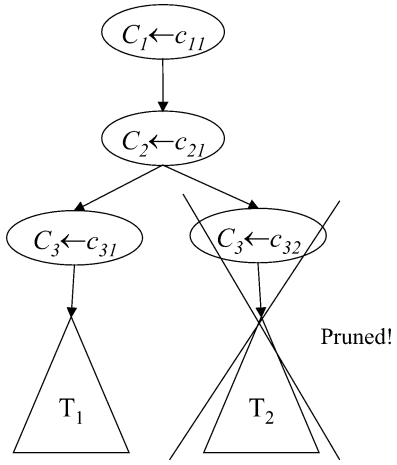


Fig. 6. The search tree showing the effects of the removal of subsumed variables.

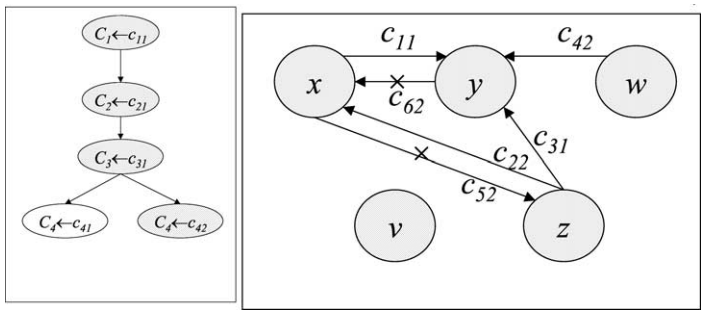


Fig. 7. Semantic branching example (c).

example, subtree T_2 and the node corresponding to A_4 in the figure would have been safely pruned.[8]

3.3. Semantic branching

A third pruning method used in solving DTPs is semantic branching (SB), which has been shown to be very effective [1]. Like *RSV*, *SB* relies on the semantics of the constraints in the DTP, i.e., on the fact that they encode numeric inequalities. The basic idea of semantic branching is the following. Suppose that during search the assignment $A \cup \{C_i \leftarrow c_{ij}\}$ is expanded in every possible way but it leads to no solution. That means that in any solution that is an extension of A, if there is any, the constraint c_{ij} does not hold. Thus, the negation of this constraint has to hold in any such solution. In other words, if c_{ij} is the constraint $x - y \leqslant b$ and we know c_{ij} does not hold, then in any solution that is an extension of assignment A, $\neg c_{ij}$ has to be true, i.e., it must be the case that $y - x < -b$. Thus, when search "branches" after failing to extend $A \cup \{C_i \leftarrow c_{ij}\}$ to a solution, and tries a different value for C_i for the rest of the search under A, we can assume $\neg c_{ij}$ holds.[9] The constraint $\neg c_{ij}$ often tightens the STP that corresponds to the current assignment A; explicitly adding this constraint can lead to values in other variable domains being removed earlier than they otherwise would have been.

Notice that with SB the current assignment A at any point in processing no longer stands in a one-to-one correspondence with an STP S. Instead S is the union of the values assigned to variables in A *and* all the current semantic branching constraints.

Example 3. To see how SB prunes the search space, we compare the search space for the DTP of Fig. 5 without and then with semantic branching. Suppose the algorithm has already assigned $A_1 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}\}$ as shown in Fig. 8. (On the left is the search of the meta-CSP and on the right is the implied STP.) The x-crossed edges are the ones removed by forward-checking while the filled nodes are the ones that belong to the current assignment. In Fig. 9 the assignment is extended to

[8] The node corresponding to A_3 would also have been pruned.

[9] The implementation of semantic branching is discussed in Section 6.6.

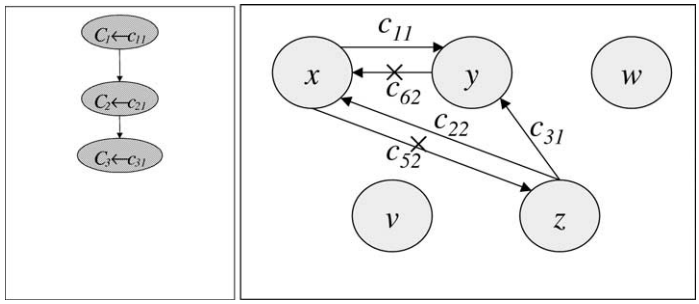


Fig. 8. Semantic branching example (a).

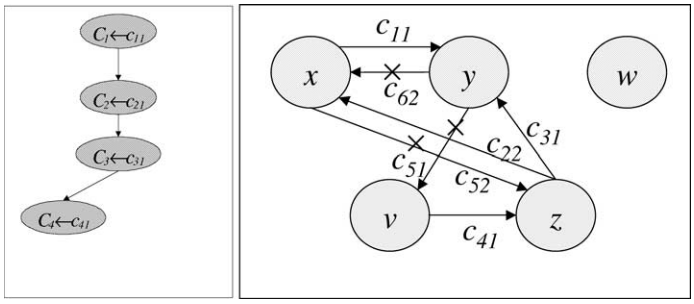


Fig. 9. Semantic branching example (b).

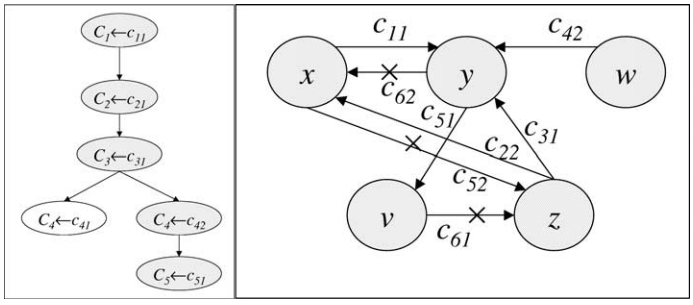


Fig. 10. Semantic branching example (d).

$A_2 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}, C_4 \leftarrow c_{41}\}$. This assignment fails because both values in $D(C_5)$ are removed.

The search then continues by trying a different value for C_4 (Fig. 7). Finally, the search reaches a dead end again because both values in $D(C_6)$ are removed (Fig. 10), after which it will backtracks back to node C_3 and continue the search.

Had we used semantic branching however, when we branched to try the second value of C_4 we would have explicitly added the constraint $\neg c_{41}$, as shown in boldface in Fig. 11. The constraint $\neg c_{41}$ allows forward-checking to eliminate value c_{61} immediately, thus reaching a dead end. In this simple example, SB prunes only one node, the one that assigns

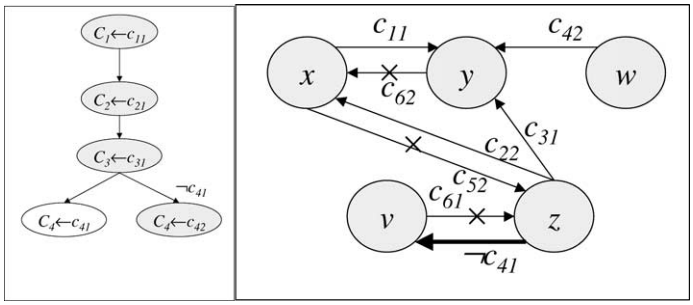


Fig. 11. Semantic branching example (e).

$C_5 \leftarrow c_{51}$ (last node in left picture of Fig. 10), but in general SB can prune an arbitrarily large number of nodes.

As is noted in [20], Semantic Branching is only useful when the disjuncts in each constraint are not mutually exclusive. For example, in scheduling applications where the constraints are typically of the form $\{A < B \text{ or } B < A\}$, when the first disjunct fails, SB will add its negation $A > B$, having no pruning effect, since the next disjunct $B < A$ is the same constraint.

4. No-good recording

No-good recording (also called no-good learning) is a powerful pruning technique for solving general CSPs [11,12,14,16,26,27,34] and SAT problems [25]. In this section, we adapt this technique to DTP solving. Intuitively, a no-good is an assignment of the variables that cannot lead to a solution, and is thus either an induced or explicit constraint of the CSP. It is important not to confuse no-goods with semantic branching constraints. No-goods are constraints of the meta-CSP, while SB constraints are constraints of the component STP associated with one particular (possibly partial) assignment to the variables of the meta-CSP.

In our Epilitis algorithm we use no-goods for two purposes: (i) for pruning the search space and (ii) as heuristic information to estimate which variables constrain the remaining search space the most. This section deals with the former, while Section 6.4 with the later.

We begin by defining no-goods in general, for an arbitrary CSP $\langle V, C\rangle$. In our definitions, we will use $C_X \in C$ to denote the constraints in C that involve only the variables in X, where $X \subseteq V$.

Definition 3. A *no-good* of CSP $\langle V, C\rangle$ is a pair $\langle A, J\rangle$, where A is a set of forbidden value assignments to a subset of V, and J, called the no-good justification or culprit, is a subset of V such that no solution of the CSP $\langle V, C_J\rangle$, given the specified domains for the variables in V, contains the assignments in A.

Example 4. Consider a CSP where $V = \{a, b, c\}$, $D(a) = D(b) = D(c) = \{1, 2\}$, with the following constraints: $C = \{C_1 = \{\neg(a \leftarrow 1 \wedge b \leftarrow 2)\}, C_2 = \{\neg(a \leftarrow 2 \wedge c \leftarrow 2)\}, C_3 = \{\neg(b \leftarrow 2 \wedge c \leftarrow 2)\}, C_4 = \{\neg(a \leftarrow 1 \wedge \text{c} \leftarrow 1)\}, C_5 = \{\neg(a \leftarrow 1 \wedge c \leftarrow 2)\}\}$. Each constraint C_i trivially induces a no-good. For example, C_1 implies that $\langle\{a \leftarrow 1, b \leftarrow 2\}, \{a, b\}\rangle$ is a no-good. Now notice that if an assignment were to include $a \leftarrow 1$, constraint C_4 would preclude c from taking value 1 and C_5 would preclude c from taking value 2. Since these are the only values in the original domain of c, we can infer the new constraint $\{\neg(a \leftarrow 1)\}$. Thus, the pair $\langle A, J\rangle$, where $A = \{a \leftarrow 1\}$ is also a no-good, for some justification J. What is the justification J? The constraints that imply the no-good are C_4 and C_5: it is as a result of these two constraints that we cannot assign a the value 1. Thus, the variables that "justify" the no-good are the variables of these two constraints and so $J = \{a, c\}$. Then $C_J = \{C_2, C_4, C_5\}$ and, as the definition requires, $A = \{a \leftarrow 1\}$ cannot be part of any solution to the CSP $\langle V, C_J\rangle$.[10] Notice that a no-good $\langle A, J\rangle$ does not only depend on the constraints C of the CSP, but also on the domains of the variables. If the domain of c in this example contained more values than 1 and 2 we could not have inferred that $A = \{a \leftarrow 1\}$ is an induced constraint.

The above example illustrates a particular point: knowing a set of no-goods, we may be able to infer other no-goods. The following two theorems present two methods for such inferences.

Theorem 4. *Let $\langle A, J\rangle$ be a no-good. Then $\langle A \downarrow J, J\rangle$ is also a no-good, where $A \downarrow V$ denotes the assignment that results from projecting assignment A on the variables of V* (*Theorem* 3.2 *in* [27]).

Intuitively, the theorem states that we can reduce the assignment of a no-good, by only considering the variables in the justification. For example, if $\langle\{a \leftarrow 1, c \leftarrow 2\}, \{a, b\}\rangle$ is a no-good, then $\langle\{a \leftarrow 1, c \leftarrow 2\} \downarrow \{a, b\}, \{a, b\}\rangle = \langle\{a \leftarrow 1\}, \{a, b\}\rangle$ is also a no-good.

Theorem 5. *Let A be a* (*partial*) *assignment of the variables in V, v_s be an unassigned variable in V, and $\{A_1, \ldots, A_m\}$ be all the possible extensions of A along v_s, using every possible value of $D(v_s)$. If $\langle A_1, J_1\rangle, \ldots, \langle A_m, J_m\rangle$ are no-goods, then $\langle A, \cup_i J_i\rangle$ is a no-good* (*Corollary* 3.1 *in* [27]).

Example 5. Theorem 5 is exactly what we used intuitively in Example 4 to infer that $\langle\{a \leftarrow 1\}, \{a, c\}\rangle$ is a no-good. Let us illustrate now the same CSP and the same derivation

[10] The reader might wonder why we define a no-good as the pair $\langle A, J\rangle$, where the justification J is the set of the *variables* involved in the constraints that imply A, instead of having J to be the set of the actual constraints. Indeed, Schiex and Verfaille [27] record the involved constraints as the no-good justifications. For our current example, the no-good would be $\langle\{a \leftarrow 1\}, \{C_4, C_5\}\rangle$ instead of $\langle\{a \leftarrow 1\}, \{a, c\}\rangle$. Notice that $C_J = C_{\{a,b\}}$ is a superset of $\{C_4, C_5\}$. In general, Definition 3 leads to less specific justifications than those discovered using the Schiex and Verfaillie method. However, when employing no-goods for solving the Disjunctive Temporal Problem, it is more convenient to encode and use as justifications sets of variables than sets of constraints, especially since in the DTP the constraints are implicit. The two definitions of no-goods are equivalent for all purposes of this paper. For a more thorough discussion on the subject see [24].

again in light of Theorem 5. If we let $A = \{a \leftarrow 1\}$, we see that $A_1 = \{a \leftarrow 1, c \leftarrow 1\}$ and $A_2 = \{a \leftarrow 1, c \leftarrow 2\}$ are all the possible extensions of A along variable c. Trivially (see the discussion in Example 4), $\langle\{a \leftarrow 1, c \leftarrow 1\}, \{a, c\}\rangle$ and $\langle\{a \leftarrow 1, c \leftarrow 2\}, \{a, c\}\rangle$ are no-goods, or equivalently $\langle A_1, \{a, c\}\rangle$ and $\langle A_2, \{a, c\}\rangle$ are no-goods. By the theorem we infer that $\langle A, \{a, c\}\rangle$ is a no-good too.

4.1. Building, recording, and using no-goods during search

Suppose we design our search algorithm so that, given a partial assignment A, it explores all extensions of A, and always returns one of two results: a solution, or a justification J for the failure of all the extensions of A. In other words, we assume that invoking a search on the successor $A \cup \{v \leftarrow u_1\}$ returns either a solution or the no-good $\langle A \cup \{v \leftarrow u_1\}, J_1\rangle$. By Theorem 5, if all successors of A fail returning $\langle A \cup \{v \leftarrow u_k\}, J_k\rangle$, then we can infer the new no-good $\langle A, \cup J_k\rangle$, which can be further reduced to the no-good $\langle A \downarrow \cup J_k, \cup J_k\rangle$ by Theorem 4. This no-good has a smaller forbidding assignment than all the no-goods of the successors and it can be returned recursively to the parent of the current node to explain why A failed to be extended to a solution. Thus, if the leaves of the search return a no-good with a justification for the failure, the internal nodes can infer and build smaller no-goods using the method just described.

The preceding discussion shows how to propagate constraints from the leaf nodes through internal nodes of the CSP. The remaining question is how to generate the no-goods at the leaves. Building no-goods at the leaves is easy for standard CSPs: if the current assignment at the leaf violates a constraint C, then the no-good $\langle A, V_C\rangle$ is returned, where V_C contains the variables in V that appear in the constraints in C. If more than one constraint C is violated, we can arbitrarily select one to return.[11]

In the meta-CSP of a DTP, the constraints among the CSP variables are implicit and have to be inferred, and so it is not as straightforward to determine what justification to return when a constraint is violated. We distinguish two cases for when assignment A violates a constraint and correspondingly two ways to form a justification for the failure:

(1) *A is a superset of A' for some already recorded no-good $\langle A', J\rangle$.* In this case J is returned as the justification.
(2) *A corresponds to an STP that is inconsistent.* Suppose that p is the negative cycle in the inconsistent STP. If there are no semantic branching constraints added, then this negative cycle is formed entirely from value-variable assignments in A. If $vars(p)$ are the variables whose value assignments are the STP constraints in p, the set $vars(p)$ is the justification that should be returned. For example, if assignment $A = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}\}$ and $p = \{c_{11}, c_{21}\}$, then the justification $J = \{C_1, C_2\}$ should be returned. However, if semantic branching constraints are added, then they might also participate in the negative cycle p, e.g., if assignment

[11] In [27] the idea of returning more than one justification per failure is explored, but this is outside the scope of this paper.

$A = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}\}$ and $p = \{c_{11}, c_{21}, v\}$, where v is a semantic branching constraint. In this case, the set of variables that constitutes the culprit of the failure is $vars(p)$ and all the variables that justify the addition of v. Assuming that we have a way of obtaining the justification of the semantic branching constraints, denoted by the function $just(v)$ for a constraint v, the justification to be returned should be is $vars(p) \cup_i just(v_i)$, where v_i are all the semantic branching constraints that participate in the negative cycle p. In order to implement function $just(v)$ we need to store the pairs $\langle v, J\rangle$ where v is a semantic branching constraint that holds in the current assignment and J the justification of the most recent failure prior to the addition of v (i.e., the failure that led to the addition of v).

Notice that case (1) requires that during search the current assignment A is checked against all recorded no-goods to determine whether $A \supseteq A'$ for some no-good $\langle A', J\rangle$. This lookup operation imposes a significant overhead for using the recorded no-goods (see [32] for an efficient implementation of no-good lookup scheme). Recording more no-goods provides better chances for pruning the search space; however, it increases the time for the lookup operation. Thus, one needs to determine which no-goods to keep among all possible no-goods discovered during search. The easiest scheme is to limit the size of the no-goods recorded by a fixed constant k: a no-good assignment is recorded only if it contains less than k value-variable assignments (independent of the size of the justification set). In the experiments we conducted we determined the best value for k for the range of problems we tested, as described in Section 6.4.

5. Integrating all pruning methods: the Epilitis algorithm

We are now ready to describe our algorithm for DTP solving. Called Epilitis, the algorithm combines all the pruning methods used in the previous literature on DTP solving—namely Conflict Directed Backjumping, Removal of Subsumed Variables, and Semantic Branching—and it adds in the no-good recording scheme of discussed in Section 4. The main difficulty in designing the algorithm is that no-good recording, CDB and SB interact and special attention is required to combine them. Here we present a high-level description of the algorithm shown in Fig. 12; complete details, sufficient for implementation, are provided in the Appendix A.

As in the previous approaches, Epilitis attacks a DTP by attempting to solve the associated meta-CSP, searching for a consistent component STP. It takes three arguments:

- A, the current assignment of values (of the form $x - y \leqslant b$) to variables,
- U, the yet-to-be-assigned variables,
- S, the current induced STP, which is represented by a distance array, a precedence array, and a set of pairs $\langle v, J\rangle$, such that v are the semantic branching constraints justified by the meta-level constraints involving the variables in J.

```
Epilitis(A, U, S)
1. /* (Removal of Subsumed Variables) */
2. For all variables x in U,
3.    Remove x from U if for any value v in d(x) the Subsumption Condition holds in the
4.    current STP S.
5. EndFor
6. If U = ∅ then
7.    Stop and report A as a solution
8. Else
9.    Select a variable x in U
10.   For all values v in the current domain of x, d(x)
11.      forward-check v
12.      If forward-check fails with justification Just,
13.         record ⟨A ∪ {x ← v} ↓ Just, Just⟩ (No-good recording)
14.      Else,
15.         Try extending A by {x ← v} (Recursively call Epilitis).
16.         If the call returns with justification J_i that does not involve x,
17.            backjump and return J_i (Conflict-Directed Backjumping)
18.         EndIf
19.      EndIf
20.      If value v fails, add reverse(v) to S, EndIf (Semantic-Branching)
21.   EndFor
22.   If all values v (in the original domain of x, D(x)) have failed or been removed from D(x)
23.   with justifications J_i
24.      record ⟨A ↓ ⋃_i J_i, ⋃_i J_i⟩, return ⋃_i J_i (No-good recording)
25.   EndIf
26. EndIf
```

Fig. 12. High-level description of Epilitis algorithm.

When Epilitis is initially invoked to solve a DTP $\langle V, C\rangle$, $A = \emptyset$, $U = C$, the variable domains are initialized as in the basic DTP algorithm of Fig. 1, and the distance and predecessor arrays are empty, as are the SB constraints.[12]

On any (recursive) call, if $U = \emptyset$ then A represents a solution to the DTP. If any variable in U is subsumed by S, then it is removed from U. Next, a variable x in U is selected and an attempt is made to extend A by making an assignment to x. Each value v_k in $d(x)$ is considered in turn and A is extended to $A' = A \cup \{x \leftarrow v_k\}$, while constraint v_k is propagated in S. If forward-checking a value v_k reduces the domain of some variable to the empty set, then a dead-end has been reached. At this point, Epilitis records the no-good $\langle A \cup \{x \leftarrow v\} \downarrow \mathit{Just}, \mathit{Just}\rangle$ where *Just* is the justification for the failure as discussed in Section 4.1. Otherwise, if forward-checking does not lead to a dead-end, then Epilitis is recursively invoked.

If a dead-end has been reached for every possible extension of x among all v_k then we build and record another no-good $\langle A \downarrow \bigcup_k J_k, \bigcup_k J_k\rangle$ where J_k are the justifications for each $A \cup \{x \leftarrow v_k\}$ failing.

[12] Recall that the distance array is the all-pairs shortest path matrix, and the predecessor array stores a predecessor of j on the shortest path from i in all entries $\langle i, j\rangle$ that are not on the main diagonal.

CDB is implemented with the following scheme: If while recursively calling Epilitis with assignment $A' = A \cup \{x \leftarrow v_k\}$ a failure occurs with justification J, then there is no need to try another value v_p if x does not appear in J: if x is not in the culprit of the failure, the same dead-end will be encountered again for $A \cup \{x \leftarrow v_p\}$. Thus, we can stop trying any remaining values in the domain of x, and backjump over x.

Finally, SB is implemented by propagating the **reverse** $w = \neg v_k$ in the current STP S, when $A \cup \{x \leftarrow v_k\}$ leads to failure. However, recall that in order to create the correct justifications for building no-goods and performing CDB the pairs $\langle w, J \rangle$ of the current set of semantic branching constraints need to be maintained, where J is the justification for adding w.

6. Experimental results

6.1. Experimental setup

We next describe the results from a series of experiments that we ran on Epilitis and the solver of Armando, Castellini, and Giunchiglia called TSAT, publicly available at http://www.mrg.dist.unige.it/~drwho/Tsat. (As described further below, TSAT has been shown to be the most efficient DTP solver previously developed [1].) The goal of the experiments was to assess the effectiveness of various combinations of the pruning strategies described in the previous sections. As is customary in the DTP literature [1,20,29,30], experimental sets were produced using the random DTP generator implemented by Stergiou, in which DTPs are instantiated according to the parameters $\langle k, N, m, L \rangle$, where k is the number of disjuncts per constraint, N the number of DTP variables, m the number of DTP constraints, and L a positive integer such that for all the disjuncts $x - y \leqslant b$, $b \in [-L, L]$ with uniform probability. In the random DTP problems we used, we used the typical settings in the literature where $k = 2$, $L = 100$, and $N \in \{10, 15, 20, 25, 30, 35\}$. Parameter k is chosen to be 2 because this is the case for constraints that typically appear in many planning and scheduling (e.g., $A < B$ or $B < A$). We also employ a derived parameter R, the ratio of constraints over variables, m/N. For each setting of N, we varied R from 2 to 14, and we generated 50 random problems for each setting of N and R. (For example, we generated 50 problems for the case where N is 30 and R is 10; those problems have 30 variables and 300 constraints.) The total number of experiment problems was $50 \times 13 \times 6 = 3900$ (13 values for R, 6 values for N). The domains of the variables are integers instead of reals so that semantic branching can easily be implemented: the negation of the constraint $x - y \leqslant b$ is $y - x \leqslant -b - 1$.[13] This is again standard with the rest of the literature.

[13] There are specific reasons why we chose to implement $\neg(x - y \leqslant b)$ as $y - x \leqslant -b - 1$. First, if the variables are integer-valued *and* all the bounds are integer valued, then obviously $y - x < -b$ is equivalent to $y - x \leqslant -b - 1$ which is stricter than $y - x \leqslant -b - \varepsilon$. In addition, both in TSAT and in the Oddi and Cesta's work, this is the method that semantic branching has been implemented. Thus, it would be unfair to compare Epilitis with TSAT using any other method. If the assumption of integer valued variables and bounds does not hold, semantic branching can be implemented as $y - x \leqslant -b - \varepsilon$ or even $y - x \leqslant -b$ (which is not as strict as possible, but sound).

The output of Epilitis provides the following statistics for each DTP solved:

- The *Time* it took to solve the problem.
- The number of constraint-checks *CCs* (i.e., the number times the algorithm checked the FC-condition or the Subsumption-Condition).
- The number of search *Nodes* generated.
- The number of constraint propagations *CProps* (i.e., number of calls to **maintain-consistency**).
- The number of no-goods checks *NCs* (i.e., the number of times a no-good is checked for retrieval).
  The number of no-goods recorded *NGs*.

In the graphs and tables showing the results below, except where otherwise noted we present the median of the above statistics over the series of the 50 experiments with the same parameters N and R. Again, this is consistent with the literature on DTP solving.

The Epilitis algorithm was implemented in Allegro Common Lisp 5.0. Both Epilitis and the ACG solver were ran on the same Intel Pentium III machine running Windows 2000, having 384 MB memory and a clock speed of 1 GHz. There is no time-out for the experiments run using Epilitis, but we used the time-out of 1000 seconds provided as a default with the ACG solver. This time-out limit is reasonable because it is an order of magnitude larger than the maximum amount of time taken by Epilitis to solve any of the test problems. Note, moreover, that by imposing a time-out limit on ACG but not on Epilitis, we are, if anything, providing an advantage to ACG in the experimental comparison.

All of our experiments confirm the existence of a critical region for values $R = 5, 6, 7, 8$, where the percentage of solvable problems is less than 10% and the median time to find a solution or prove there is no solution to a DTP problem substantially exceeds the median time taken when $R < 5$ or $R > 8$.

6.2. Pruning power of techniques

In the first set of experiments we investigated the pruning power of all the pruning methods and combinations thereof. This set of experiments answers the following questions: (i) can all pruning methods be integrated efficiently? (ii) how do the pruning methods and their combinations compare quantitatively?

The pruning methods we tried are:

- Removal of Subsumed Variables (RSV).
- Conflict-Directed Backjumping (CDB).
- Semantic Branching (SB).
- No-good Recording (NG).

In Epilitis all of the above methods can be individually turned on and off, providing us with the opportunity to try any combination we desire. The only limitation is that whenever *NG* is on, *CDB* must also be turned on. We name our graphs and tables using the following convention: we list the options that were turned on separated by spaces or dashes. When we

bound the size of the no-goods, as explained in Section 4.1, we follow the name with the numerical bound. For example, CDB-RSV-SB-NG_10 is Epilitis with CDB, RSV, SB, NG on and a maximum size of no-goods set to 10, and CDB-RSV-SB-NG the same algorithm with no bound on the size of no-goods. We use the term "Nothing" to identify "bare" Epilitis, with no pruning techniques turned on.

For this set of experiments we used the following *dynamic* variable and value heuristics:

- Select the variable according to the MRV (*Minimum Remaining Values*). Break the ties by selecting the variable that contains the value that *maximizes* the number of pairwise inconsistencies with the values in the domains of the unassigned variables.
- Select the value that *minimizes* the number of pairwise inconsistencies with the values in the domains of the unassigned variables.

This heuristic is typical in the CSP literature. The idea is that by choosing the variable with the value that maximizes the pairwise inconsistencies, the branching factor of the search is reduced, since this is the variable that most constrains the search. On the other hand, when we select a value we prefer the one that least constrains the search so that we increase the probability of finding a solution that contains this value.

In Tables 2–4 we show the results for $N = 20$, $N = 25$, $N = 30$ for various pruning methods and their combinations. The results for $N < 20$, not reported here, are similar. The columns are listed in increased order of efficiency for $R = 6$; this is the peak of the critical region. The tables are not complete, i.e., some pruning method combinations are missing, because they caused the algorithm to be too slow for the experiments to complete (e.g., "nothing", i.e., the no pruning methods version is not reported in Table 3). All times are reported in seconds, as in all experiments in this paper. We selected size 10 for bounding the no-good size because in other experiments (described subsequently), size 10

Table 2
The ordering of the pruning methods according to Median Time when $R = 6$, and $N = 20$

Ratio	Nothing	RSV	CDB	CDB-RSV	NG_10	NG	SB	SB-RSV	CDB-SB	CDB-SB-RSV	CDB-SB-RSV-NG	CDB-RSV-SB-NG_10
2	0.02	0.02	0.02	0.02	0.03	0.03	0.02	0.02	0.02	0.02	0.03	0.021
3	0.05	0.05	0.05	0.05	0.06	0.06	0.05	0.05	0.05	0.05	0.06	0.06
4	0.14	0.13	0.13	0.11	0.14	0.15	0.14	0.12	0.14	0.11	0.13	0.13
5	1.91	1.39	1.05	0.811	0.49	0.551	0.531	0.51	0.501	0.451	0.421	0.431
6	4.1	3.33	2.8	2.39	1.53	1.46	1.43	1.34	1.25	1.24	1.04	0.941
7	1.93	1.74	1.87	1.5	1.07	1.31	1.05	1.01	0.981	0.971	1.02	0.851
8	1.15	1.11	1.05	0.892	0.781	0.982	0.671	0.651	0.661	0.621	0.751	0.701
9	0.711	0.661	0.671	0.611	0.621	0.681	0.551	0.54	0.521	0.53	0.62	0.571
10	0.671	0.611	0.631	0.571	0.6	0.66	0.55	0.551	0.541	0.511	0.571	0.531
11	0.56	0.551	0.521	0.521	0.541	0.61	0.461	0.441	0.451	0.441	0.531	0.511
12	0.491	0.501	0.481	0.461	0.491	0.551	0.451	0.431	0.43	0.431	0.521	0.48
13	0.461	0.48	0.461	0.461	0.461	0.521	0.45	0.44	0.42	0.411	0.51	0.48
14	0.441	0.42	0.441	0.43	0.471	0.541	0.42	0.41	0.421	0.411	0.551	0.491

Table 3
The ordering of the pruning methods according to Median Time when $R = 6$, and $N = 25$

Ratio	RSV	CDB	CDB-RSV	NG_10	SB	NG	CDB-SB	SB-RSV	CDB-SB-RSV	CDB-RSV-SB-NG_10	CDB-SB-RSV-NG
2	0.04	0.04	0.05	0.04	0.04	0.05	0.04	0.04	0.04	0.05	0.05
3	0.1	0.11	0.111	0.12	0.11	0.12	0.11	0.1	0.1	0.11	0.11
4	0.37	0.291	0.31	0.32	0.361	0.311	0.311	0.32	0.251	0.29	0.281
5	9.22	4.3	2.41	1.06	2.08	0.892	1.5	1.69	1.04	0.811	0.681
6	40.8	27.5	24.8	10.1	8.77	8.72	7.98	7.7	7.43	7.05	5.38
7	18.5	16.3	14.1	8.56	7.72	7.66	7.94	7.47	6.94	6.78	7.09
8	6.8	5.81	5.02	4.1	3.37	3.83	3.36	3.2	3.14	2.74	2.71
9	4.99	4.46	4.45	3.51	3.3	3.56	3.14	3.1	2.89	2.83	2.36
10	3.46	2.69	2.45	2.08	2.01	2.36	1.94	1.97	1.74	1.92	1.9
11	2.48	2.21	1.86	1.9	1.52	2.17	1.57	1.42	1.36	1.61	1.52
12	2.44	2.3	1.98	1.71	1.44	1.86	1.58	1.36	1.36	1.53	1.44
13	1.84	1.63	1.44	1.31	1.26	1.48	1.2	1.13	1.11	1.18	1.16
14	1.68	1.3	1.25	1.38	1.18	1.51	1.07	1.12	1	1.3	1.22

Table 4
The ordering of the pruning methods according to Median Time when $R = 6$, and $N = 30$

Ratio	SB	CDB-SB-RSV	CDB-SB	SB-RSV	CDB-SB-RSV-NG_10
2	0.07	0.07	0.07	0.07	0.08
3	0.17	0.18	0.17	0.17	0.18
4	0.4	0.39	0.371	0.36	0.39
5	10.3	7.42	7.12	8.25	4
6	149	142	140	138	79.8
7	74.6	70.5	69.7	78.2	48.8
8	29.4	26.6	25.9	31.5	21.1
9	13.9	12.6	12.4	14.1	11.1
10	9.73	9.15	8.92	10.3	7.72
11	6.73	6.28	6.09	6.69	4.93
12	4.47	4.64	4.42	4.61	4.12
13	4.32	4.31	3.77	4.38	3.97
14	3.96	4.02	3.91	4.17	3.2

was determined to be optimal size for the Epilitis with all pruning methods on. Although it would be desirable to have an analytic technique for predicting the optimal size of no-goods, we do not know of a suitable such account, and to date, all results on optimal no-good size have been determined experimentally.

As expected in Tables 2–4, “nothing” performs the worst, then RSV, then CDB, then SB, NG, and NG_10 following closely together. It makes sense to compare the time of each algorithm, since their underlying implementation is the same.

Table 5
The statistic Median Nodes divided by Median Nodes of "Nothing" for $N = 20$. The pruning methods are sorted according to this statistic for $R = 6$

Ratio	RSV	CDB	CDB-RSV	SB	SB-RSV	CDB-SB	CDB-SB-RSV	NG_10	NG	CDB-SB-RSV-NG_10	CDB-SB-RSV-NG
	%	%	%	%	%	%	%	%	%	%	%
2	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
3	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
4	100.00	100.00	98.91	97.83	97.83	97.83	97.83	100.00	100.00	97.83	97.83
5	76.00	42.00	42.00	30.40	30.40	27.73	27.73	20.33	17.53	15.67	15.13
6	88.84	67.36	63.64	35.29	35.29	32.77	32.52	31.12	27.31	19.75	19.75
7	89.71	79.22	73.69	50.49	50.49	45.24	44.66	39.13	39.13	31.17	32.82
8	100.00	84.01	77.07	54.72	54.72	51.25	50.67	52.99	53.95	44.51	44.12
9	91.20	84.00	84.00	78.00	78.00	72.00	72.00	70.80	70.00	59.60	59.60
10	95.24	81.43	78.10	77.62	77.62	69.05	69.05	68.10	68.10	56.19	56.19
11	98.57	81.43	81.43	77.14	77.14	67.86	67.86	71.43	71.43	66.43	66.43
12	100.00	96.12	96.12	95.15	95.15	86.41	86.41	84.47	84.47	78.64	78.64
13	100.00	88.64	88.64	92.05	92.05	81.82	81.82	84.09	84.09	79.55	79.55
14	100.00	97.56	97.56	95.12	95.12	87.81	87.81	82.93	82.93	82.93	82.93

Since the search space increases exponentially with the number of variables, the results become more significant as N grows. Thus the differences among pruning methods is most obvious in Table 4, which shows the results for $N = 30$. The worst combination is the CDB-RSV: we were not even able to complete this experiment for $N = 30$. The best performance is the CDB-SB-RSV-NG_10. When we did not bound the size of the no-goods, the performance of the algorithm was seriously degraded for $N = 30$ and this is why it is not included in the table.

We now compare the different pruning methods strictly according to their pruning power, i.e., not including the computational overhead to implement them. Table 5 shows the statistic $Nodes_c/Nodes_{\text{Nothing}}$ where $Nodes_c$ is the median number of search nodes explored by the algorithm in each column c, and $Nodes_{\text{Nothing}}$ the search nodes explored by Epilitis with no pruning methods. As we would expect, the more methods we add, the more we prune the search space. In this case NG is the best single[14] method, exploring only 27.31% of the whole search space (i.e., when no pruning method is on) for $R = 6$. NG is even better than the combination of all three other methods CDB-SB-RSV, which explores 32.52% of the space. This result encourages us to look for even more efficient implementations of recording and retrieving no-goods to reduce the overhead of the technique. Table 6 supports the same argument, showing the effect of pruning methods on the search space explored for $N = 30$, where the search space is significantly larger than for $N = 20$ (the value in Table 5). The statistic displayed is $Nodes_c/Nodes_{SB}$, where $Nodes_c$ is the median number of search nodes explored by the algorithm in each column c, and $Nodes_{SB}$ the search nodes explored by Epilitis with SB on. For example, in the last column, for $R = 6$, we see that the ratio is 38.99% meaning that the algorithm CDB-RSV-

[14] Recall, however, that when NG is on, CDB is also on, so the comparison is not entirely fair.

Table 6
The statistic Median Nodes divided by Median Nodes of SB for $N = 30$. The pruning methods are sorted according to this statistic for $R = 6$

Ratio	SB-RSV %	CDB-SB %	CDB-SB-RSV %	CDB-SB-RSV-NG_10 %
2	100.00	100.00	100.00	100.00
3	100.00	100.00	100.00	100.00
4	89.31	89.94	89.31	89.31
5	92.96	71.37	69.25	32.16
6	100.00	94.93	94.93	38.99
7	100.00	97.13	93.03	47.13
8	100.00	89.59	89.59	55.90
9	100.00	86.26	86.26	66.79
10	100.00	93.37	93.37	65.06
11	100.00	85.98	85.98	60.31
12	100.00	92.95	92.95	63.57
13	100.00	95.23	95.23	72.57
14	100.00	96.97	96.97	66.67

SB-NG_10 explored only 38.99% of the space the algorithm SB explored on problems for $N = 30$ and $R = 6$. The results show in an impressive way the pruning power of no-goods: the last column, corresponding to the algorithm with the no-goods on, display a significant reduction of the space searched.

Summarizing the results of this section:

- A rough partial ordering of the pruning methods is RSV $<$ CDB $<$ CDB-RSV $<$ NG_10 $<$ SB $<$ {CDB-SB, SB-RSV, CDB-SB-RSV} $<$ CDB-SB-RSV-NG_10.
- No-good learning needs to limit the size of the no-goods recorded because asymptotically the overhead of recording and looking-up all the possible no-goods greatly outweighs the benefits.
- SB is the best single pruning method in terms of performance, i.e., displaying a good trade-off between pruning power and implementation overhead.
- NG is the best single pruning method in terms of pruning, even better than all the other methods combined CDB-SB-RSV.
- The Epilitis with options CDB-SB-RSV-NG_10 considerably improves performance over all previous other methods combined CDB-SB-RSV.

6.3. The number of forward-checks is the wrong measure of performance

Our first set of experiments were designed to compare the effectiveness of alternative combinations of pruning strategies and it was straightforward to present the results of those experiments since what we are concerned with is precisely the number of search nodes in the meta-CSP that are pruned. In the third major experiment, which we present below in Section 6.5, we compare Epilitis running with the most effective combination of pruning heuristics, to TSAT, the previous most effective DTP solver. It is less obvious

what metrics to use in this comparison. It is customary in the literature to compare DTP solvers and report their performance using the number of forward-checks—more precisely the total number of values that the algorithm forward-checked during search, also called consistency-checks *CCs*[15] [1,20,29,30]. Such comparisons make the implicit hypothesis that the number of consistency-checks is a machine and implementation independent measure of performance. In this section we present both theoretical arguments and empirical evidence that this hypothesis is false and *CCs* is the wrong measure of performance.

There are at least three reasons for rejecting *CC* counts as a performance metric. First, the use of no-good recording in Epilitis gives an unfair advantage for a comparison based on *CC* counts. This is because no-good recording requires significant overhead to record and retrieve no-goods, and this overhead is not represented in the number of forward-checks.

Second, as discussed in Section 2.2, the time required for each consistency-check depends on the method used for maintaining consistency. For example, when the distance array of each current STP is available, checking the FC-condition takes constant time, but when it is not, more time might be required.

The third argument against the use of CC-counts as a metric is that there are techniques that have been employed in previous DTP solvers that result in fewer values being forward-checked even though more time is spent exploring. For example, a technique used by Stergiou and Koubarakis [29,30] and ACG [1] is what we will call *Forward-Checking Switch-off* (FC-Off). In Appendix C, we provide an example that illustrates that FC-Off can reduce the number of forward-checks by increasing the number of search nodes explored and thus it may even decrease the performance of a DTP solving algorithm.

As a result of the three problems with using *CC* counts as a measure of performance, we report actual computation times used in our comparison of TSAT and Epilitis. One drawback of such a comparison is that we cannot as easily draw conclusions about the pruning efficiency of Epilitis' additional pruning methods, such as RSV, CDB, and NG, since TSAT uses a very inefficient method for consistency-checks. Thus, the better performance of Epilitis might be attributed only to the better consistency-checking techniques it employs. We cannot totally dismiss this hypothesis until a version of TSAT is re-implemented using better forward-checks methods. However, our first set of experiments, which analyzed the pruning methods and showed their effectiveness, make it unlikely that efficient consistency-checking is solely responsible for Epilitis' performance advantage.

6.4. Heuristics and optimal no-good size bound

Before presenting the actual comparison of Epilitis and TSAT, we need to pin down one more detail, namely, the search heuristic used for selecting which variable/value

[15] In our implementation a consistency-check is essentially checking the FC-condition. We also felt that we should count as a consistency-check determining whether the Subsumption-Condition holds, because both are similar operations having similar functions and take the same time. Not counting the Subsumption-Condition checks in *CCs* would favor all algorithms with RSV on since those are the ones that perform this operation.

combination to select during each stage of the search. Interestingly, the use of no-good recording increases the range of search heuristics available, because the heuristic itself can take into account the no-good information. In turn, however, this means that the performance of the search heuristic is intertwined with the size of the no-good recorded. Thus, we designed a factorial experiment aimed at discovering the best combination of search heuristic (from amongst a set of plausible heuristics) and bound on no-good size.

As heuristic information we considered four functions that estimate how much a value x constrains the remaining search space. These are:

- E0: the number of remaining values that are pairwise inconsistent with x (i.e., calculated dynamically during search using the current domains).
- E1: the number of values that are pairwise inconsistent with x determined *statically* before search begins.
- E2: the number of the remaining values that are pairwise inconsistent with x plus the number of no-goods x appears in.
- E3: the number of the remaining values that are pairwise inconsistent with x. Ties are broken by the number of no-goods x appears in.

As estimators of how much a variable v constrains the remaining search space we used the maximum of the value estimator used over all remaining values in v's domain.

In each of our heuristics we follow the principle of selecting the variable that most constrains the remaining search space, in an attempt to minimize the branching factor. Thus, a variable with minimal current domain is chosen first (*Minimum Remaining Values* heuristic) by default. However, since all domains have maximum size two, it is often the case that there are many ties, and these are then broken by using one of the above estimators E0–E3, giving rise to the four heuristics H0–H3, respectively.

In contrast, as value selection principle we select the value that least constrains the search space, in an effort to hit a solution faster. We again use the estimators above. For example, in H2 we would first select the variable v with the value that maximizes the number of pairwise inconsistencies and participates in the most recorded no-goods, among all the variables with least domain size. Then, we would select the value in v's domain that minimizes this estimator (E2).

Fig. 13 presents the results. The x-axis shows the R, ratio of constraints to variables; this is the critical parameter for the DTP-solving problems. The y-axis shows computation time taken, in seconds; note that the scale is logarithmic. We name the curves as "NG xy" to denote Epilitis with *all the pruning options on*, where x is the limit on the size of no-goods, and y is the search heuristic used. We show only the graph for $N = 30$ since this is the largest size we ran. For each no-good size the best heuristic is selected (e.g., for size 6 we determined the best heuristic to be H2). Overall, the combination of H2 with size bound of 10 works best, although H3 with bound of 14 come very close. However, we suggest using H2 with size bound of 10 because it exhibits a better average case behavior.

To summarize the results of this section:

- The recorded no-goods do not only prune the search space but can also be effectively used as heuristic information.

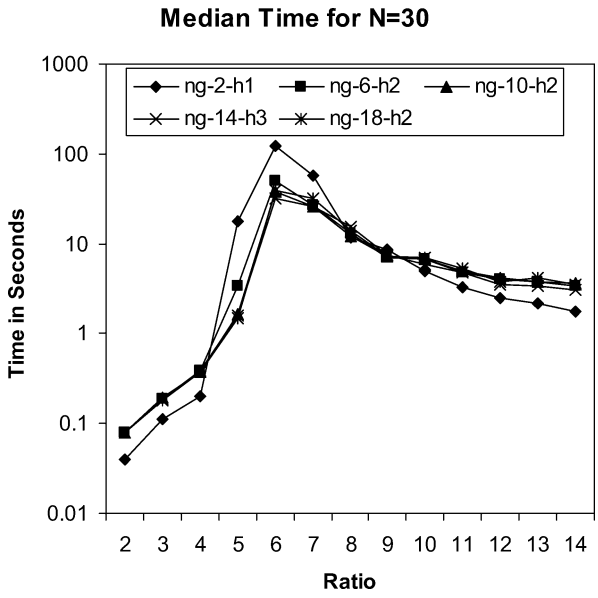


Fig. 13. Comparison for the best overall algorithm for $N = 30$.

- Epilitis with CDB, SB, RSV, NG on, maximum no-good size 10, and heuristic H2 is the best algorithm in the set of experiments we ran.

6.5. *Comparing Epilitis to the previous state-of-the-art DTP solver*

In Section 7 below we provide details of Epilitis' three predecessors: one developed by Stergiou and Koubarakis [30], one by Oddi and Cesta [20], and one by Armando, Castellini, and Giunchiglia (TSAT) [1]. Oddi and Cesta present an experimental comparison, noting that while their algorithm consistently outperforms that of Stergiou and Koubarakis, it is at best competitive with TSAT algorithm. Moreover, they show that TSAT is particularly good at the hardest problems, i.e., those in the critical region. As Oddi and Cesta note, "further work [on their system] will be needed to clearly outperform TSAT". These are the problems that are most significant, since problems outside of this range can already be solved very quickly. Given these results, we view TSAT as the state-of-the-art predecessor to Epilitis, and conduct head-to-head experiments with it. It is worth noting, however, that there is one class of problems for which Oddi and Cesta's approach outperforms that of TSAT: problems with small R values ($R \leqslant 5$). On these problems, Oddi and Cesta's system is about one order of magnitude faster than TSAT. Although we have not conducted a head-to-head comparison of Epilitis with Oddi and Cesta's system on such problems, Epilitis is also about an order of magnitude faster than TSAT, and thus it is reasonable to conclude that it is competitive with Oddi and Cesta's system for these (relatively easy) problems.

We now turn to the details of the final experiment, in which we compare CDB-SB-RSV-NG_10-H2 with TSAT. Fig. 14 shows the results for $N = 25$ and $N = 30$. As explained above, we report overall computation time; as in Fig. 13, the x-axis shows R, and the

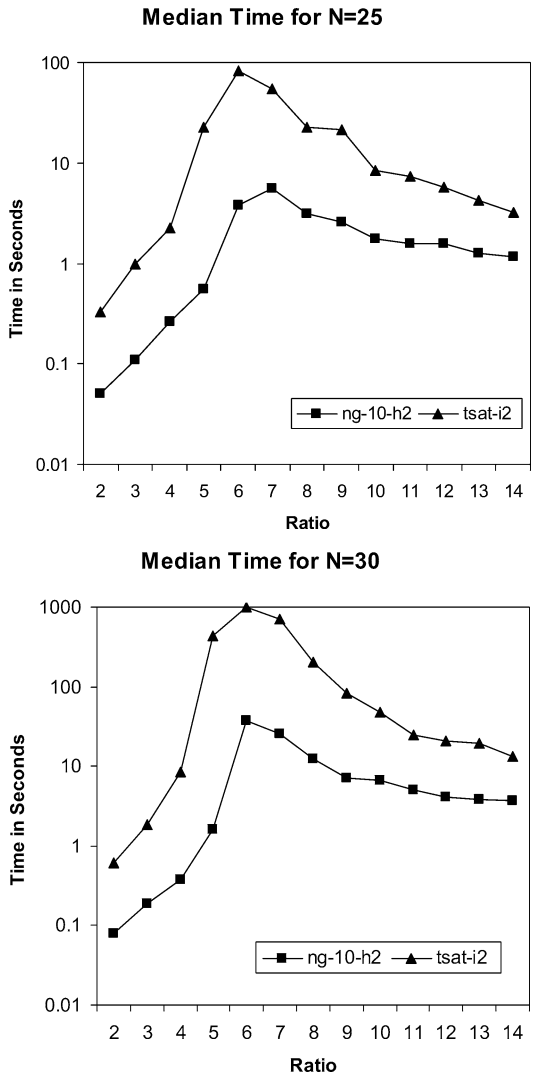


Fig. 14. Comparison of Epilitis and ACG's solver for $N = 25$ and $N = 30$.

y-axis, which is logarithmic, shows median computation time in seconds. Epilitis is faster by about two orders of magnitude for the larger ($N = 30$) problems. Also recall that TSAT has a time-out of 1000 seconds imposed, and so the real median time taken by their program might be significantly more than is indicated here. For example, for $N = 30$ and $R = 6$ the median time is exactly 1000 seconds implying that the TSAT solver timed-out on more than half the problems.

We also ran the best version of Epilitis on problems of up to size $N = 35$ (the largest size of random DTP problems reported as so far) and we observed that the algorithm scales relatively well. Fig. 15 shows the performance of Epilitis on problems of different sizes.

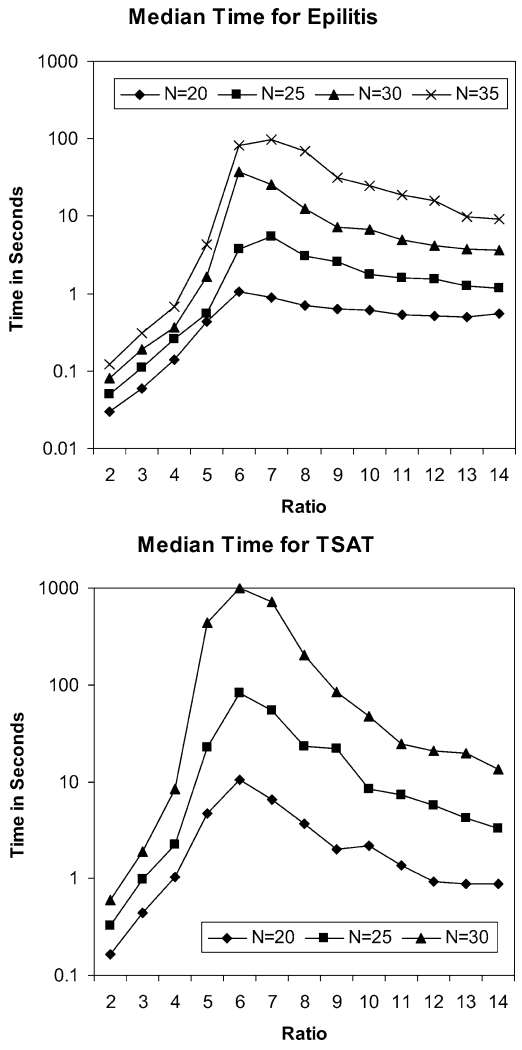


Fig. 15. The median time performance of Epilitis and TSAT.

For the larger problems where $N = 35$ Epilitis has a median time performance of about 100 seconds, while the corresponding TSAT performance is more than 1000 seconds for problems of size $N = 30$. The overall performance of TSAT is also shown in the same figure. As we can see TSAT requires about one order of magnitude more time every time N is increased by 5. In contrast Epilitis' performance does not degrade as fast.

Summarizing the results of this section:

- Epilitis is almost two orders of magnitude faster than the previous state-of-the-art DTP solver TSAT on standard benchmark problems.
- Epilitis' performance scales comparatively well as the size of the problems increase.

6.6. Application experience

The previous sections have described controlled experiments we performed using synthetic, abstract test problems, to analyze the performance of the various pruning strategies developed for DTP solving and to compare Epilitis with the previous state-of-the-art system. We also have some experience using Epilitis in a large application, which we briefly describe here.

Autominder [21] is an intelligent cognitive orthotic system: a system designed to help older adults with memory decline by providing them with adaptive, personalized reminders about their daily activities. Autominder has three main components: a Plan Manager, which stores a plan of it's clients daily activities, and is responsible for updating it and identifying potential conflicts in it; a Client Modeler, which uses information about the client's observable activities[16] to track the execution of the plan, inferring what activities the client has already performed and what activities are pending; and a Personal Cognitive Orthotic, which reasons about any disparities between what the client is supposed to do and what she is doing, and makes decisions about when to issue reminders.

The relevant component for the current paper is the Plan Manager. It maintains a model of the client's daily plan encoded as a DTP. It then invokes Epilitis to update the plan in response to four types of events:

(1) The addition of a new activity to the plan.
(2) The modification or deletion of an activity in the plan.
(3) The execution of an activity in the plan, as reported by the Client Modeler.
(4) The passage of a time boundary in the plan.

In each of these cases, the Plan Manager formulates a DTP: for instance, when a new activity is added, the DTP consists of the constraints in the original client plan, the constraints in the new activity, and a set of constraints generated to represent the resolution of any conflicts between the two. Epilitis then attempts to solve the DTP, indicating whether or not it succeeded, and returning any newly required constraints. For instance, the addition of an activity may result in added constraints on the time of performance of a previously added activity.

In general, the size of the plans managed by Autominder are small by the standards of the planning community. (On the other hand, our focus is not on plan generation, but on a range of other tasks, such as plan monitoring, update, and dispatch.) Generally the client plan has on the order of 30 actions, meaning that there are 60 events (start and end points), and, with typically temporal constraints, the representation requires about 4 or 5 constraints per action, and about 2 or 3 disjuncts per constraint. For problems of this scale, Epilitis nearly always produces solutions (or determines inconsistency) in less than a second, an amount of time that is well within the bounds we require.

[16] The current version of Autominder is deployed on a mobile robot, and uses on-board sensors to observe a client's movement about her home.

Table 7
Comparison of all DTP solvers

Technique	SK	ACG	OC	Epilitis
		Temporal reasoning		
Constraint Propagation	Maintain IDPC, $O(\|V\|^2)$	N/A	Maintain IFPC, $O(\|V\|^2)$	Maintain IFPC, $O(\|V\|^2)$
Forward-Checking (time complexity is for each value)[a]	Find distance, check FC-Condition. Actual implementation $O(\|V\|^2)$; could be done in $O(\|V\|)$.	Run an STP consistency-checking algorithm, Actual implementation: $O(2^{\|V\|})$. Could be done in $O(\|V\|^3)$.	Lookup distance, check FC-Condition, O(1)	Lookup distance, check FC-Condition, O(1)
Value Subsumption (time complexity is for each value)[b]	Not in the original implementation. Could be done by finding distance, then checking Subsumption-Condition, $O(\|V\|)$	Not in the original implementation. Could be done by running an STP consistency algorithm $O(\|V\|^3)$.	Lookup distance, check Subsumption-Condition, O(1)	Lookup distance, check Subsumption-Condition, O(1)
		Searching methods		
CDB	Yes	No	No	Yes
RSV	No	No	Yes	Yes
SB	No	Yes	Yes	Yes
FC-off	Yes	Yes	No	Yes
NG	No	No	No	Yes
IFC[19]	No	No	Yes	No
		Heuristics		
Variable	MRV	Max-Inc	MRV	See Section 6.3
Value	The value with the time-points with the most appearances in other values	The value with the most pairwise inconsistencies (but it might be negated)	No specific heuristic	See Section 6.3

[a] The time complexity shown is for implementing forward-checking with the best known algorithm: Given that in the SK approach STP, the current STP is in directional path consistency, we can find the distance between a pair of nodes in time $O(|V|)$ and check the FC-Condition. As already mentioned, in the actual implementation, SK used a less efficient scheme that takes quadratic time. In the ACG approach, we have to run an STP consistency-checking algorithm, while in the actual implementation ACG use Simplex.

[b] Neither SK nor ACG use RSV, so these two cells do not refer to their actual implementation. Instead, this is the most efficient way they could have implemented RSV had they desired to, given the way they perform temporal propagation.

[c] IFC refers the to Incremental Forward-Checking technique of [20] in which a value v: $x - y \leqslant b_{yx}$ is forward-checked only if the distance d_{xy} has changed since last forward-checking took place. If it has not, then v satisfies the FC-Condition for sure. We mention this technique for completeness. IFC does not reduce the search space and it can be used in conjunction with any pruning technique.

7. Related work

7.1. Previous DTP solvers

There are two previous DTP-solvers that treat component-STP selection as CSP problems and perform a search in the same meta-CSP as Epilitis: that of Stergiou and Koubarakis [29,30] (hereafter SK) and of Oddi and Cesta [20] (hereafter OC). We can compare these approaches in terms of the implementation of (1) **maintain-consistency**, (2) **forward-check**, (3) the variable ordering heuristic, (4) the value ordering heuristic, (5) and the techniques employed for the search and for pruning the search space. An additional approach, which casts DTP-solving in terms of SAT, is discussed in Section 7.1.2.

7.1.1. The CSP approaches

In the SK approach, function **forward-check** is implemented by adding a value to the current STP S and propagating using again the IDPC algorithm, identifying the inconsistency if there is one, and then retracting the constraint using again IDPC so as to be ready to forward-check the next value. This requires two calls to IDPC with $O(|V|^2)$ in the worst-case for each value to be forward-checked. There is of course a much faster algorithm: checking if the FC-Condition holds. The FC-condition requires the distance between two time-points, which given that the SK approach maintains the current STP in directed path consistency form, can be found in $O(|V|)$ time in the worst case with the algorithm described at [9].

The variable ordering heuristic in SK is the *Minimum Remaining Values* (MRV) in which the variable with the fewest remaining values in its domain is selected first. Ties among variables are broken by choosing the variable that contains the time-points that appear the most in the rest of the variables. Each variable may contain many disjuncts and each disjunct contains two time-points, so we can choose the variable that contains the time-point with the maximum appearance in other variables/disjunctions or the variable with the largest sum of appearances of its time-points in other variables/disjunctions. The SK paper does not discuss exactly how the selection is performed. The value ordering heuristic is the same as the tiebreaker heuristic above. The disjunct whose time-points appear in the most in other variables is selected first. SK experimented with the anti-heuristic but with disappointing results.

For the approach of OC, the table summarizes well the design-choices made. The OC variable ordering heuristic is the MRV with no other tie-breaking heuristics. There is no particular value ordering heuristic.

7.1.2. The SAT approach

The Armando, Castellini, and Giunchiglia (ACG) approach differs from the previous ones in that it treats the component-STP selection not as a meta-CSP problem, but as a SAT problem instead. However, we can still use the above classification scheme to compare the approach: The ACG algorithm does not use **maintain-consistency**. Every time the algorithm requires checking the consistency of a set of STP-like constraints they use a

version of the Simplex algorithm for linear programming. Simplex has exponential worst-case performance.

During a preprocessing step, ACG enhances the SAT formula with clauses (equivalently variables in a CSP-based approach) that correspond to inconsistent pairs of literals (equivalently values in a CSP-based approach). This provides additional guidance to an MRV-like criterion for variable selection: they choose the clause that contains the literal with the greatest number of occurrences in the clauses of minimal-length (which implies they will choose the clause with the literal that participates in the most pairwise inconsistencies with other literals). We will call this heuristic *Max-Inc* because essentially it picks the clause with the literal that participates in most pairwise inconsistencies.

For variable ordering, they choose the literal that maximizes the number of pairwise inconsistencies in the previous step. Notice here however, that a SAT-based procedure can either choose to branch on the literal c_{ij} or the literal $\neg c_{ij}$. Instead, a CSP-based approach can only branch on c_{ij}, i.e., assign a value to a variable, and never the negation of the value to a variable. From the ACG paper it is unclear which branch (i.e., the positive or the negative) is taken first and how this choice is made.

Table 7 summarizes the above discussion and comparison between the different approaches. The DPT solving approaches are ordered chronologically according to the date of appearance in the literature.

7.2. Other temporal reasoning formalisms

The focus of this paper has been on developing more efficient techniques for solving DTPs. One question we must address is why we want to use DTPs, when there are other formalisms for temporal reasoning that are computationally more tractable. For example, as we noted at the beginning of the paper, DTPs subsume both Simple Temporal Problems (STPs) and Temporal Constraint Satisfaction Problems (TCSPs). STPs allow only non-disjunctive constraints, while TCSPs allow constraints of the form $c_{i1} \vee \cdots \vee c_{in}$ where each c_{ij} is of the form $x - y \leqslant b$ with the restriction that x and y are the same in all c_{ij}. STPs can be solved in polynomial time. Although the same is not true for TCSPs, they are still computationally more tractable than DTPs, because all their constraints are binary, involving only two variables, while the DTP constraints may be non-binary, and it is significantly easier to calculate path-consistency in networks of binary constraints than in networks where the constraints are non-binary [3–5].

It turns out, however, that the limitations of TCSPs do result in weak expressive power: in particular, we consider the most serious disadvantage of TCSPs to be their inability to express the fact that two intervals should not overlap. This kind of constraint is essential in scheduling and planning applications where it is often the case that some actions should not overlap, for example if they utilize the same unary resource. As was illustrated in the example in the Introduction, it is straightforward to encode such a restriction with a DTP constraint: if denote with A_S (A_E) and B_S (B_E) the start (end) times of actions A and B, then the fact that they cannot overlap can be written as the DTP constraint:

$$A_E - B_S \leqslant 0 \vee B_E - A_S \leqslant 0.$$

This constraint involves four time-points A_S, B_S, A_E, and B_E and so it cannot be represented by a TCSP, but it is perfectly acceptable in a DTP. There are other, *binary*, representations however, that allow such constraints to be represented. These include the *Point-Interval-Algebra* (PIA) described in [18]. In PIA the variables can be either time-points or intervals; and all relations are binary: interval-interval, point-point, interval-point. The constraints between time-points can be metric TCSP constraints, while the rest of constraints are qualitative. Having two more interval variables A_I and B_I representing the intervals associated with the actions can then encode the above situation by imposing the disjunctive constraint:

$$A_I \ \{before, after\} \ B_I.$$

Although PIA can thus model prohibited overlaps, it cannot readily handle requirements of temporal separation between actions. For example, suppose that A and B are two medical treatment procedures applied to the same patient with the constraint that if A is applied first, B can only be applied 3 days later, while if B is performed first then A can be performed 2 days later. The constraint cannot be represented in PIA but written as the DTP constraint

$$A_E - B_S \leqslant -3 \vee B_E - A_S \leqslant -2.$$

Nevertheless, extensions of the PIA have appeared that allow constraints of this sort to be represented while remaining within the realm of binary constraints [7,8].

The above argument may suggest that we can avoid non-binary constraints if we employ both intervals and time-points as our representational elements. However, there are other types of constraints that are inherently non-binary such as conditional constraints of the form "if *constraint1* then *constraint2*", e.g., "if treatment A does not last enough, then perform treatment B for at least e days". The constraint can be written as:

$$\neg(d \leqslant A_E - A_S) \Rightarrow (e \leqslant B_E - B_S), \quad \text{or equivalently}$$
$$(d > A_E - A_S) \vee (e \leqslant B_E - B_S)$$

and these are only expressible with DTPs.

There are two other formalisms that are as expressive as DTPs, namely the Generalized Temporal Network (GNC) described in [28] and the Temporal Constraint Networks (TCN) of Barber [2]. The former is essentially a DTP-like formalism that allows conjunctions of STP-like constraints in each disjunct. Because it is straightforward to convert a constraint of this form into a standard DTP-constraint, the advantages of GNCs is not obvious. Barber's TNCs are also as expressive as DTP. A TNC is a TCSP with the addition of what Barber calls I-L-Sets. I-L-Sets (Inconsistent-Label-Sets) are essentially no-goods: an I-L-Set looks has the form $\neg(c_{ij} \wedge \cdots \wedge c_{mn})$ denoting that the conjunction does not hold in the TCSP. A constraint $a \vee b$, where a and b are STP-like constraints of the form $x - y \leqslant b_1$ and $w - z \leqslant b_2$ (involving two pairs of different variables) cannot be represented as a TCSP constraint, but it can be encoded as the I-L-Set $\neg(\neg a \wedge \neg b)$ (using De'Morgan's rule for Boolean Algebra). However, TCNs require that the disjuncts in an I-L-Set participate in some other TCSP constraint. Thus, it is not enough to just add the above I-L-Set; we also have to add TCSP constraints so that a and b appear in the underlying TCSP. For example,

we can add the TCSP constraints $(a \vee \neg a)$ and $(b \vee \neg b)$. By using this scheme, TNCs reach the expressiveness of the DTP, albeit in a peculiar way. To solve TCNs, Barber in [2] provides a path-consistency algorithm that in essence calculates the full set of all no-goods (whose number is exponential to the number of disjuncts), but no experimental results are provided.

7.3. Scheduling algorithms

Another related area of work is that of Automated Scheduling. In particular, the Precedence Constraint Posting (PCP) technique of Cheng and Smith [7,8] bears certain similarities to DTP solving. Cheng and Smith apply PCP to typical scheduling problems such as the Job-Shop Scheduling (JSSP) and the Hoist Scheduling Problem with very encouraging results. Cheng and Smith used a formalism based on the PIA in [18] and employed domain-specific heuristics. What makes DTP and PCP solving similar, and distinguishes them from most other automated scheduling techniques, is their use of the meta-CSP approach. This contrasts with scheduling algorithms that formulate the problem as a CSP with variables that are the start times of the events, and directly solve that CSP.

Stergiou and Koubarakis [30] applied their DTP solver on JSSP with somewhat disappointing results. However, it bears remembering that they were comparing a fairly general-purpose temporal reasoning module (their DTP solver) against many highly optimized algorithms that had been tuned specifically for job-shop scheduling problems. It is also worth noting that, JSSP problems are typically optimization problems: it is often relatively easy to find a solution, but very hard to find an optimal solution. In contrast, at least on the random DTP problems we have tested, just finding one solution is inherently hard.

8. Discussion, contributions, and future work

In this paper, we have focused on the development of efficient techniques for solving Disjunctive Temporal Problems (DTPs). DTPs are a class of constraint-based temporal reasoning problems that appeared in the literature for the first time only in 1998 [29]. Although DTPs are potentially very useful for a range of planning and scheduling problems, solving them can be computationally quite costly. We therefore examined the strategies that had been proposed in the previous literature for improving the efficiency of DTP-solving, considered how to integrate these strategies with one another and with no-good learning, and conducted systematic experiments to determine what combination of strategies is most effective.

Our experiments were conducted using a DTP-solving system that we implemented, Epilitis. Epilitis is instrumented so that the user can "turn on" various pruning strategies. It is publicly available,[17] and may be used as a testbed for further exploration of DTP

[17] Contact the first author at ioannis.tsamardinos@vanderbilt.edu. Epilitis will also be available on the first author's web site in the future.

solving. In our own experiments, we were able to achieve a speed-up of almost two orders-of-magnitude over the previous fastest algorithm, by combining a set of pruning strategies, adding no-good learning with an experimentally determined size bound, and using a carefully analyzed search heuristic.

One important result of our experiments was the discovery that no-good learning is particularly powerful in improving the efficiency of DTP-solving. We can explain this by noting that, in a DTP solver, forward-checking a value requires the propagation of the corresponding STP-constraint in the current STP, which is a relatively costly (albeit polynomial) operation. Thus, even though there is computational overhead associated with retrieving no-goods, this overhead may be outweighed by the savings in forward-checking. In an ordinary (non-temporal) CSP, forward-checking may be less expensive, and the benefits of no-good recording might not be as great.

We have already demonstrated the practical usefulness of DTP-solving in general, and Epilitis in particular, in two of our other research projects. In one of these, the Plan Management Agent (PMA) [22], we are designing an intelligent calendar that manages a user's plan. In the other, the Autominder [33], we are designing a cognitive orthotic system intended to manage and monitor the daily activities of an elderly user, providing him or her with appropriate, timely reminders. Epilitis plays a central role in both systems, serving as the main engine for updating and modifying the modeled plans.

There are many avenues for future work on this topic, and here we mention just a few.

- Explore dynamic DTPs. Dynamic DTPs are sequences of DTPs that differ from successive elements by a few constraints. Such sequences arise naturally in certain planning problems in which new goals are periodically added to an existing set of goals. No-good recording may be especially useful for dynamic DTPs, as it is possible that some of the no-goods identified and recorded for the previous DTP will still hold for the next DTP in the sequence, and thus can prune the search space. In one extreme case, if the no-good $\langle \emptyset, J \rangle$ still holds in the next DTP, then the DTP is inconsistent and this is proven without any search performed! In [27] it is shown that the method greatly improves the performance of CSP solvers on dynamic (non-temporal) CSPs.
- Consider replacing forward-checking as the underlying algorithm in Epilitis with a full looking ahead approach, investigating the interactions between such an approach and the various speed-up techniques discussed in this paper.
- Investigate the use of random restart techniques [17] to supplement the efficiency-increasing strategies already described in this paper. It seems plausible that there will be a synergistic influence between no-good recording and random restarts. We also expect that certain scheduling techniques, such as profiling, could be applied to DTP solving.
- Use DTPs to model conditional plans. As mentioned in Section 7.1, we noted that the n-ary constraints of DTPs can be used to model plans with conditional (if-then) branches. In work already underway, we have analyzed the notion of consistency in conditional temporal networks, and have shown that consistency-checking in these networks can be reduced to DTP-solving [33].
- Introduce temporal uncertainty. Even though DTPs are very expressive, subsuming a large number of other temporal and scheduling problems, they have one key limitation:

they do not readily permit one to model events whose time of occurrence is not known and is not under the control of a planning agent. The recently developed STPU formalism does support modeling of such events, but only as extensions to STPs. It would be very useful to develop techniques for combining STPU and DTPs.

Appendix A. Proofs of theorems

Let us denote by $d_{xy}(S)$ the distance between time-points x and y in STP S and by $d_{xy}(A)$ the distance between time-points x and y in the STP induced by assignment A.

Theorem 1. *A value c_{ij}: $y - x \leqslant b_{xy}$ is inconsistent with a consistent STP S (that is, $S \cup c_{ij}$ is inconsistent) if and only if the following condition holds*:

$$b_{xy} + d_{yx}(S) < 0 \quad (\textit{FC-condition}).$$

Proof. Let p_{yx} be the shortest-path from y to x in S and thus the length of p_{yx} is d_{yx}. If the FC-condition holds, then the path $p_{yx} \cup (x, y)$ has length $b_{xy} + d_{yx}(S)$ and so it forms a negative cycle making S inconsistent.[18]

Conversely, let us suppose that the FC-condition is false and prove that $S' = S \cup c_{ij}$ will be consistent. We will prove this claim by contradiction, i.e., we will assume FC condition is false and S' is inconsistent, and will derive a contradiction. If S' is inconsistent then there is a negative cycle, and because S was consistent before we added c_{ij}, that means that the negative cycle involves the new constraint c_{ij}. Let us assume the negative cycle is $c_{ij} \cup p'_{yx}$, for some path p'_{yx}. So $b_{xy} + d''_{yx} \leqslant b_{xy} + \textit{length}(p'_{yx}) < 0$ (1), where d''_{yx} is the distance between y and x in S' for some path p'', which is the shortest path from y to x (Fig. A.1). Since the negative cycle is a simple cycle (no loops allowed), then edge (x, y) is not a member of p''. Therefore, the shortest path p'' from y to x in S' is does not contain the new constraint c_{ij} and thus distance from y to x in S' and S is the same: $d_{yx} = d''_{yx}$. By (1) above we get that $b_{xy} + d''_{yx} = b_{xy} + d_{yx} < 0$, i.e., FC-condition holds, contrary to what we assumed. □

Theorem 2. *A value c_{ij}: $y - x \leqslant b$ is subsumed by an STP S if and only if $d_{xy}(S) \leqslant b$* (Subsumption-Condition), *where $d_{xy}(S)$ is the distance between x and y in S.*

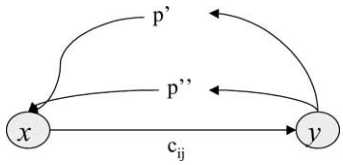


Fig. A.1. Proof of Theorem 1.

[18] Let p_{ab} be a path $(a, n_1, \ldots, n_k, b)$ in a graph G, and let c be a node in G. Then $p_{ab} \cup c$ is the path $(a, n_1, \ldots, n_k, b, c)$ in G.

Proof. ($\Leftarrow$) Suppose that the Subsumption-Condition holds for value c_{ij}. By definition of the distance $y - x \leqslant d_{xy}(S)$ holds in all exact STP solutions, so $y - x \leqslant d_{xy}(S) \leqslant b$ holds in all exact solutions, and c_{ij} holds in all exact solutions of S.

($\Rightarrow$) Conversely, suppose the Subsumption-Condition does not hold, i.e., $b < d_{xy}(S)$ holds. Then there is some exact solution s of the STP for which $y - x = d_{xy}(S)$ (as shown in [13]). Thus, in s, $b < y - x$, i.e., c_{ij} does not hold in all of S's exact solutions. □

Lemma A.1. *Adding more constraints to an STP can only result in monotonic decrease in the distances between nodes.*

Proof. Immediate, by the fact that adding a constraint to an STP can only reduce the solution set, and thus the distances have to be smaller or equal to the original STP. □

Lemma A.2. *Adding a value c: $\{y - x \leqslant b\}$ to an STP S when S subsumes c, does not change the distance array of S (i.e., the resulting STP is equivalent to the one before the addition).*

Proof. Since c is subsumed, by Theorem 2, $d_{xy}(S) \leqslant d$. Let p be a shortest path in the distance array from x to y, and so its weight is $d_{xy}(S)$. Let us suppose now that the distance array of S changes when c is added. That means that the new constraint c participates in at least one shortest path, let us say on the path from w to z that before the addition had distance $d_{wz}(S)$. From shortest path properties we get:

$$d_{wz}(S) \leqslant d_{wx}(S) + d_{xy}(S) + d_{yz}(S) \leqslant d_{wx}(S) + b + d_{yz}(S). \tag{A.1}$$

After the addition, in the new STP $S' = S \cup c$, the new distance $d_{wz}(S')$ is:

$$d_{wz}(S') = d_{wx}(S') + b + d_{yz}(S') = d_{wx}(S) + b + d_{yz}(S). \tag{A.2}$$

Notice that $d_{wx}(S') = d_{wx}(S)$ and $d_{yz}(S') = d_{yz}(S)$ because b is already participating in the shortest path from w to z and therefore cannot participate on the shortest paths from w to y or from x to z or a cycle would be present on the path from w to z (i.e., the shortest paths would not be simple).

Since the distance array changed, the new shortest path has to be strictly smaller the one than before c was added (Lemma A.1), i.e.,

$$d_{wz}(S') < d_{wz}(S). \tag{A.3}$$

(A.1) and (A.2) imply that $d_{wz}(S) \leqslant d_{wz}(S')$ which contradicts (A.3). Therefore our initial assumption that the distance array will change is false. □

Theorem 3. *Let $D = \langle V, C\rangle$ be a DTP, let A be an assignment on D (i.e., a component STP), and let C_i be a variable subsumed by A. Then A is a solution of D if and only if it is a solution of $D' = \langle V, C - C_i\rangle$.*

Proof. Since we assume that C_i is a subsumed variable, then, there must be a value c: $y - x \leqslant b$ in the domain of C that is subsumed by A. Suppose that A is a solution

of D. Then obviously, it is a solution of $D' = \langle V, C - C_i \rangle$ since D' has one less variable (DTP constraint). Conversely, suppose A is a solution of D'. Since c is subsumed by A, it holds in all of A's exact solutions, and thus C_i which is a disjunction involving c, holds in all of A's exact solutions. Thus, if A is a solution of the DTP constraint $C - C_i$, it also solves the DTP constraints C. □

Corollary A.1. *Let A be a partial assignment during a DTP search, U be the unassigned variables, and Sub be the set of subsumed variables in U. If A can be extended to a solution over variables in $U - Sub$, it can be extended to a solution over variables in U. In other words, we can remove the subsumed variables from the unassigned variables during search. The solution to the reduced problem is a solution to the original.*

Proof. By Theorem 3 if A' is an extension of A over the variables at $U - Sub$, and A' is consistent, then A' is also a solution to the original DTP. □

Appendix B. The Epilitis algorithm in detail

Epilitis, shown in Fig. B.1, is a generalization of the no-good recording algorithm in [26] (see Fig. 4 *ibid*.), and it just adds code to this algorithm. *The lines common to both algorithms are annotated with an asterisk* following the line number.[19] Epilitis would still correctly solve DTP problems if only these lines are included. The only difference from the plain no-good recording algorithm would be in forward-checking. Epilitis' forward-checking mechanism is similar to the one in Fig. 2, which takes into consideration the fact that the values of the meta-CSP express STP-like constraints. In other words, the algorithm in [26] (Fig. 4), with a modified forward-checking function, is a no-good recording DTP solver.

Having said that, two points must be explained regarding the workings of Epilitis: (i) the additional ("un-starred") lines in the algorithm and (2) the exact way forward-checking is performed. The former is the topic of the rest of this section, and the latter the topic of the next one.

For the rest of the discussion let us assume that there is available a function **forward-check**(A, U, S) that given the assignment A, removes from the variables U all the values in their current domains that are inconsistent with A. To check the FC-Condition efficiently we provide to the function the distance array S that corresponds to A. If a domain of a variable is reduced to the empty set, then **forward-check** should return a justification K (also called the *value killers* of the domain values; see also [32] (Section 3.5) and [26]), which is a minimal set of variables in A such that the constraints among them cause all the variables of the domain to be removed.

[19] The additional lines 45–47 are not in the original paper [26]. In direct communication with the first author of the paper it was established that lines 45–47 are indeed required for the algorithm to be complete. The experimental results in that paper are not invalidated however because lines 45–47 were included in the implementation and were only missing from the pseudo-code description of the algorithm. Careful implementation of the Epilitis algorithm also revealed a typo in the original publication of the algorithm. Line 33 appears originally as record(project(A, K), K) while it should be record(project(A', K), K).

```
1. Epilitis(A, U, S) /* A is the set of assigned CTP variables, U the set of unassigned variables, and S
a distance and predecessor array representing the current STP */
2*      If U = ∅ Then
3*        A is a solution, Stop
4*      Else
5*        Let x be a variable in U, J = ∅, BJ = false
6. SB     SBJ = ∅              /* Set the semantic branching justification to empty */
7. RSV    If there is value v ∈ d(x) subsumed by S Then
8. RSV      Return Epilitis(A, U − {x}, S)
9. RSV    EndIf
10*       For each v ∈ d(x) until BJ      /* loop for all values in the current domain or until the BJ flag is true */
11*         A' = A ∪ {x ← v} /* add value to a (new) assignment */
12.         S' = maintain-consistency(v, S)
13.         If S' is inconsistent Then
14. SB        SBJ = justification-value(v, S')
15.           J = J ∪ SBJ
16.           GoTo 36
17.         EndIf
18. FC-off  If d(x) is singleton /* when FC-off omit forward-checking when d(x) is singleton */
19. FC-off    K = ∅
20. FC-off  Else
21*           K be forward-check(A', U, S')
22.         EndIf
23*         If K = ∅ /* If we have not reach a dead end ... */
24*           Let J-sons be Epilitis(A', U − {x}, S') /* then recursively call Epilitis */
25* CDB       If x ∈ J-sons Then
26*             J ← J ∪ J-sons
27* CDB       Else /* If the current variable does not participate in the failure justification */
28* CDB         J ← J-sons, BJ = true          /* then exit the loop */
29* CDB       Endif
30. SB        SBJ = J-sons
31*         Else
32*           J ← J ∪ K
33* NG        record(project(A', K), K)
34. SB        SBJ = K
35*         Endif
36. SB      If v is not the last value in d(x) /* remember SB-constraints is the only global variable */
37. SB        S = maintain-consistency(S, reverse(v)); SB-Constraints = SB-constraints ∪ ⟨reverse(v), SBJ⟩
38. SB        If S is inconsistent
39. SB          un-forward(x), FinishLoop
40. SB        EndIf
41. SB      EndIf
42*         un-forward(x)
43*       EndFor
44*       If BJ = false Then
45*         For each v ∈ D(v) − d(v) /* add the justifications that removed the values */
                  /* from the current domain */
46*           J ← J ∪ killers(v)
47*         EndFor
48* NG      record(project(A, J), J)
49*       Endif
50. SB    Remove all semantic branching constraints added in this invocation of Epilitis from SB-Constraints
51*       Return J
52*     Endif
```

Fig. B.1. The Epilitis algorithm.

The annotations SB, FC-off, CDB, NG and RSV shown next to a line in the algorithm indicate which of the corresponding techniques (semantic branching, FC-off, conflict directed backjumping, no-good recording, and removal of subsumed variables, respectively) the line serves. For example, to remove semantic branching from the algorithm, we could just remove the lines annotated with SB.

The *Removal of Subsumed Variables* (RSV) in lines 7–9 is achieved by testing if, in the next variable to assign, there is a value that is subsumed by the current STP S. The test can be achieved by checking the Subsumption-Condition of Theorem 2. If the variable is subsumed, then it is removed from the unassigned variables and Epilitis is recursively called.

Line 12 propagates the value/constraint of assignment $\{x \leftarrow v\}$ in the current STP S' so that the distances of the STP corresponding to the current assignment A' are available with a simple table lookup. Recall that the distances are required to calculate both the FC-Condition and the Subsumption-Condition. This technique of maintaining the distance array was described in full in Section 2.2 and presented in the algorithm in Fig. 2.

When only the basic forward-checking DTP solving algorithm is used, no assignment $\{x \leftarrow v\}$ will ever cause an inconsistency to the current STP because, if it did it would have been removed by forward-checking. However, when semantic branching is used, it becomes necessary to check that the constraint $\{x \leftarrow v\}$ added by semantic branching does not cause an inconsistency. This is the reason for the check at line 13. If we have indeed hit an inconsistency the reason for it is accumulated in variable J (line 15), which is the justification to return in case of a failure (line 50). Line 14, annotated with SB, is explained in the discussion of semantic branching below.

Next the algorithm performs FC-off (lines 18–20). Very simply, if there is only one value in the current domain of the current variable, we omit forward-checking and assume that it succeeded by setting $K = \emptyset$. The ramifications of the omission were the subject of Section 6.3 above and Appendix C. Otherwise, we perform forward-checking and store in K the *value killers* of the domain that was reduced to the empty set or we store $\emptyset$ to K if there is no such domain.

If we have not hit a dead-end, i.e., $K = \emptyset$ (line 23), then we recursively call Epilitis. If it returns, then we have failed to extend the current assignment A' to a solution and the return value is a justification of the failure, stored in the variable *J-sons*. If the current variable participates in this justification (line 25) then we accumulate the justifications in variable J and proceed (after line 36) trying new values for the current variable. If on the other hand, the current variable has nothing to do with the failure, we jump to line 28, where *BJ* (from backjumping) is set to *true* so that we exit the loop and avoid trying any other values of the current variable, and finally return the same reason *J-sons* that caused the failure in the recursive call. On the other hand, if forward-check fails, it returns the value killers K that are accumulated in the overall justification J (line 32). Line 33 records the no-good implied by the dead-end. Lines 23–35 (apart from the addition of lines 30 and 34) are exactly the same as in the non-temporal no-good recording algorithm.

Perhaps the most complicated addition is the lines that achieve semantic branching. Integrating SB with the rest of the pruning techniques has implications that also affect the details of forward-checking but for the moment we restrict the discussion only to the code that appears in Fig. B.1. When the current assignment $A \cup \{x \leftarrow v\}$ fails to extend

to a solution, the code reaches line 36. As already described in detail in Section 3.3, for the rest of the search under assignment A, we can assume that $\neg v$ does not hold. Thus, line 36 propagates the **reverse** of v in the current STP S (i.e., the STP that corresponds to assignment A). The propagation might cause an inconsistency which would be identified at line 38 in which case there is no reason to try a different value for variable x and we can exit the loop.

For reasons that we explained in Section 5, it is necessary to store the semantic branching constraints that we propagate along with the justification for their addition. The store occurs at line 37 where pairs $\langle v, SBJ \rangle$ are stored in the global variable *SB-Constraints*, where v is an STP-like constraint and *SBJ* a justification (from semantic branching justification). There are three different reasons why the current value v causes an inconsistency, and correspondingly, three different lines where *SBJ* is assigned a value. Value v might directly cause an inconsistency in the current assignment A in which case *SBJ* is the justification discovered by function **justification-value** (Fig. 4)[20] and the assignment takes place at line 14. Alternatively, if after assigning value v forward-check failed, then *SBJ* should be the value killers K that forward-check returned (line 34). Finally, if after assigning value v forward-check succeeded but the recursive call to Epilitis failed, then *SBJ* is assigned the value *J-sons* (line 30). In all three cases, we fail to extend $A \cup \{x \leftarrow v\}$ to a solution and we store in *SBJ* the reason for the failure (i.e., the culprit of the variables participating in the constraints that cause the inconsistency).

The rest of the Epilitis algorithm, as already mentioned, is exactly the same as the non-temporal no-good recording algorithm in [26].

B.1. Forward-checking and justifications in Epilitis

Fig. B.1 presented the Epilitis algorithm, however, important details for its implementation were hidden in the **forward-check** and **justification-value** functions. We now proceed to the discussion of these two functions.

Recall that we assumed that **forward-check**(A, U, S) is a function that, given the assignment A, removes from the variables U all the values in their current domains that are inconsistent with A. The STP S containing the distance array that corresponds to A is passed to efficiently check the FC-Condition. A very important feature of **forward-check** is that if a domain of a variable is reduced to the empty set, it should return a justification K (also called the *value killers*), which is a minimal set of variables in A whose constraints cause the variables of the domain to be removed.

Forward-check should check if each remaining value v in some current domain of a variable should be removed or not. A value v should be removed, if, as before, the FC-Condition holds; is should also be removed if $A \cup \{x \leftarrow v\}$ is a superset of some recorded no-good $\langle A', J \rangle$, i.e., if $A' \subseteq A \cup \{x \leftarrow v\}$. That achieves forward-checking, but it does not solve the problem of assembling and returning a justification in the case where a variable domain is reduced to the empty set. Let us suppose that **justification-value**(v, S)

[20] Function **justification-value** has to be slightly modified from Fig. 4 to work for Epilitis as we will see in the next section.

is responsible for returning the justification of the removal of a single value v given the current STP S. Then, the overall justification for a variable domain being empty is the union of the justifications for removing each value originally in that domain. Function **forward-check** for Epilitis is shown in Fig. B.2.

Now we can turn our attention to the implementation of the function **justification-value**(v, A, S) which should return a set of variables from A, i.e., a justification that explains why $A \cup \{C \leftarrow v\}$ cannot be extended to a solution and thus why v has to be removed from the current domain of C. Trivially, all the variables in A plus the variable C constitute a justification for the removal of v. However, we can find smaller justifications that provide more opportunities for conflict directed backjumping and search pruning.

There are two reasons why a value v might be removed. The first one is if $A \cup \{C \leftarrow v\}$ is a superset of A', where $\langle A', J' \rangle$ is a recorded no-good, and then the justification for removing v is J'.[21] The second reason is if v is removed because $A \cup \{C \leftarrow v\}$ corresponds to an inconsistent STP S. Then, as explained in detail in Section 4, the variables that cause the inconsistency are the ones that have values assigned to them that participate in a negative cycle in S.

forward-check(A, S, U)
1. For each variable C in U
2. For each value c: $x - y \leqslant b_{xy}$ in $d(C)$
3. If $b_{xy} +$ **distance**$(y, x, S) < 0$ (FC-Condition) or
4. $A \cup \{C \leftarrow c\}$ is a superset of A', where $\langle A', \textit{no-good-}J \rangle$ is a recorded no-good
5. Remove c from $d(C)$
6. If $d(C) = \emptyset$
7. $K = \emptyset$
8. For each value v in $D(C)$
9. $K = K \cup$ **justification-value**(v, A, S)
10. EndFor
11. return K
12. EndIf
13. EndIf
14. EndFor
15. Return $\emptyset$

Fig. B.2. Forward-checking for Epilitis.

justification-value$(c$: $y - x \leqslant b, A, S\}$
1. If $A \cup \{C \leftarrow c\}$ is a superset of A', where $\langle A', J' \rangle$ is a recorded no-good
2. Return J'
3. Else
4. $p =$ **shortest-path**(y, x, S)
5. Return **vars**$(p \cup c\} \cup \{J$, where $\langle v, J \rangle \in$ *SB-Constraints* and $v \in p\}$
6. EndIf

Fig. B.3. The function justification-value for Epilitis.

[21] See [32] for a complete description of the algorithm for storing and retrieving no-goods used in Epilitis.

As already mentioned in Section 5, when semantic branching is present, the current STP S does not directly correspond to the current assignment A, but instead is formed by all the constraints in A plus the semantic branching constraints added. Thus, in Epilitis, when the negative cycles are identified, the corresponding justification is not just the variables with values that participate in the cycle, but also the variables (i.e., the justifications) that were responsible for the addition of the semantic branching constraints that participate in the cycle. These justifications can be found in the *SB-Constraints* structure: whenever a semantic branching constraint is propagated the pair $\langle \neg v, SBJ \rangle$ is stored at *SB-Constraints* (line 37, Fig. B.1). Function **justification-value** modified for Epilitis is shown in Fig. B.3.

Appendix C. FC-off reduces the number of forward-checks but increases solving time

With FC-off, when the current domain $d(C_i)$ of a variable C_i has been reduced to a singleton set $\{c_{ij}\}$, forward-checking is suspended and the constraint c_{ij} is assigned to C_i without forward-checking. FC-off is illustrated in the following example:

Example C.1. Consider the following DTP:

$$C_1\colon \{c_{11}\colon\ y - x \leqslant 5\},$$
$$C_2\colon \{c_{21}\colon\ x - z \leqslant 5\},$$
$$C_3\colon \{c_{31}\colon\ v - x \leqslant 5\} \vee \{c_{32}\colon\ z - v \leqslant 10\},$$
$$C_4\colon \{c_{41}\colon\ y - z \leqslant -10\} \vee \{c_{42}\colon\ x - y \leqslant -10\}.$$

Without FC-off, when the current assignment becomes $A_1 = \{C_1 \leftarrow c_{11}\}$, forward-checking will remove variable c_{42} from $d(C_4)$. In the next step, when the current assignment is $A_2 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}\}$, c_{41} is also removed and thus $d(C_4)$ becomes empty and the search returns failure. In contrast, when FC-off is used, when the current assignment is $A_1 = \{C_1 \leftarrow c_{11}\}$ forward-checking is suspended and nothing is removed from any variable's domain. Similarly, when the current assignment becomes $A_2 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}\}$, forward-checking is still turned off, and so still nothing is removed from any variable's domain. Only when the non-singleton domain $d(C_3)$ is encountered, and the current assignment becomes $A_3 = \{C_1 \leftarrow c_{11}, C_2 \leftarrow c_{21}, C_3 \leftarrow c_{31}\}$ is forward-check called; at this point, it will recognize the failure. Notice that when FC-off is not used, the algorithm forward-checks a total of $5 + 3 = 8$ values (i.e., it performs five checks for A_1—one for each of the other values—removing one of those values; and three checks for A_2). However, it only expands two nodes. In contrast, with *FC-off*, the algorithm checks 2 values (performed only when the current assignment is A_3) and it expanded three nodes.

The above example shows that a technique such as FC-off may or may not increase the performance of a DTP solving algorithm. With FC-off there is less forward-checking but more nodes are expanded (Theorem 17 in [30]). Therefore, overall effect of the FC-off technique will depend on the relative time required to expand nodes and to perform forward-checking. There has not been a theoretical analysis of the conditions under which

Table C.1
The ordering of performance for $N = 30$, $R = 6$, from worst (top) to best performance

Median Time	Median *CCs*
SB-FC	SB
CDB-SB-RSV-FC	CDB-SB
SB	SB-RSV
CDB-SB-RSV	CDB-SB-RSV
CDB-SB	SB-FC
SB-RSV	CDB-SB-RSV-FC

Table C.2
The ordering of performance for $N = 20$, $R = 6$, from worst (top) to best performance

Median Time	Median *CCs*
Nothing	Nothing
CDB-RSV-FC	RSV
CDB-FC	CDB
RSV	CDB-RSV
CDB	SB
CDB-RSV	SB-RSV
CDB-SB-RSV-FC	CDB-SB
SB-FC	CDB-SB-RSV
SB	CDB-FC
SB-RSV	CDB-RSV-FC
CDB-SB	SB-FC
CDB-SB-RSV	CDB-SB-RSV-FC

FC-off improves performance, but our experiments, shown below, suggest that *FC-off* frequently degrades performance even though it reduces the number of forward-checks.[22]

The arguments just given about the flaws in using *CC* counts as a metric of performance are supported by our experiments, as illustrated in Tables C.1 and C.2, which show how well different combinations of pruning strategies worked in Epilitis. First consider Table C.1, which shows the results for $N = 30$. The first column orders the algorithms in decreasing order according to the median time taken in the critical region where $R = 6$. The second column orders the algorithms by using the median number of *CCs*, from highest to lowest. (So, in both columns, combinations that are "better" are listed at the bottom on the column.) It is easy to see that the median *CCs* favors the algorithms that use FC-off (denoted with an FC in their name) and ranks them the best while in fact they are the worst in terms of median time performance.

[22] In contrast, Stergiou and Koubarakis note "We have also measured the CPU times used by the algorithms we studied. As expected, the CPU times are proportional to the number of consistency-checks" [30, Section 6]. We hypothesize that this is because in their implementation each consistency-check was not a simple array lookup. Instead, it involved one constraint propagation and one constraint retraction. Thus, the total time spent in forward-check greatly dominates the time spent in maintain-consistency.

We repeated the same procedure as above for $N = 20$, for which a greater number of experiments of pruning combinations were available. The results are displayed in Table C.2. Note again that there is a large disparity between the ranking by *CC* count and the ranking by time. For instance, using the *CC* metric, CDB-SB-RSV-FC is ranked the best, five positions higher than it really is when we consider execution time.

References

[1] A. Armando, C. Castellini, E. Giunchiglia, SAT-based procedures for temporal reasoning, in: Proc. 5th European Conference on Planning (ECP-99), Durham, in: Lecture Notes in Computer Science, Vol. 1809, Springer, Berlin, 1999, pp. 97–108.
[2] F. Barber, Reasoning on interval and point-based disjunctive metric constraints in temporal contexts, J. Artificial Intelligence Res. 12 (2000) 35–86.
[3] C. Bessière, Non-binary constraints, in: Principles and Practice of Constraint Programming (CP'99), Alexandria, VA, Springer, Berlin, 1999.
[4] C. Bessière, P. Mesequer, J. Larrosa, E. Freuder, On forward-checking for non-binary constraint satisfaction, in: Proc. CP'99, Alexandria, VA, 1999.
[5] C. Bessière, J.C. Regin, Arc consistency for general constraint networks: Preliminary results, in: Proc. IJCAI-97, Nagoya, Japan, 1997.
[6] X. Chen, P. van Beek, Conflict-directed backjumping revisited, J. Artificial Intelligence Res. 14 (2001) 53–81.
[7] C. Cheng, S.F. Smith, Applying constraint satisfaction techniques to job-shop scheduling, Technical Report CMU-RI-TR-95-03, Carnegie Mellon University, Pittsburgh, PA, 1995.
[8] C. Cheng, S.F. Smith, A constraint posting framework for scheduling under complex constraints, in: Proc. Joint IEEE/INRIA Conference on Emerging Technologies for Factory Automation, Paris, France, 1995.
[9] N. Chleq, Efficient algorithms for networks of quantitative temporal constraints, in: Constraints'95, 1995.
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990.
[11] R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition, Artificial Intelligence 41 (1990) 273–312.
[12] R. Dechter, D. Frost, Backtracking algorithms for constraint satisfaction problems, Technical Report, University of California at Irvine, 1999.
[13] R. Dechter, I. Meiri, J. Pearl, Temporal constraint networks, Artificial Intelligence 49 (1991) 61–95.
[14] D. Frost, R. Dechter, Dead-end driven learning, in: Proc. AAAI-94, Seattle, WA, 1994.
[15] M. Ginsberg, Dynamic backtracking, J. Artificial Intelligence Res. 1 (1993) 25–46.
[16] M. Ginsberg, D. McAllester, GSAT and dynamic backtracking, in: Proc. KR-94, Bonn, 1994, pp. 226–237.
[17] C.P. Gomez, B. Selman, H. Kautz, Boosting combinatorial search through randomization, in: Proc. AAAI-98, Madison, WI, 1998.
[18] I. Meiri, Combining qualitative and quantitative constraints in temporal reasoning, in: Proc. AAAI-91, Anaheim, CA, 1991.
[19] R. Mohr, T.C. Henderson, Arc-consistency and path-consistency revisited, Artificial Intelligence 28 (1986) 225–233.
[20] A. Oddi, A. Cesta, Incremental forward checking for the disjunctive temporal problem, in: Proc. European Conference on Artificial Intelligence, Berlin, 2000.
[21] M.E. Pollack, Planning technology for intelligent cognitive orthotics, in: Proc. 6th International Conference on AI Planning and Scheduling, 2002.
[22] M.E. Pollack, J.F. Horty, There's more to life than making plans: Plan management in dynamic environments, AI Magazine 20 (4) (1999) 71–83.
[23] P. Prosser, Hybrid algorithms for the constraint satisfaction problem, Comput. Intelligence 9 (1993) 268–299.
[24] E.T. Richards, Non-systematic search and no-good learning, Ph.D. Thesis, IC Parc. London, Imperial College, 1998.

[25] R.J. Bayardo, R.C. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: Proc. AAAI-97, Providence, RI, 1977.
[26] T. Schiex, G. Verfaillie, Nogood recording for static and dynamic constraint satisfaction problems, Internat. J. Artificial Intelligence Tools 3 (2) (1994) 187–200.
[27] T. Schiex, G. Verfaillie, Stubbornness: A possible enhancement for backjumping and no-good recording, in: Proc. European Conference in Artificial Intelligence (ECAI-94), Amsterdam, 1994.
[28] S. Staab, On non-binary temporal relations, in: Proc. European Conference on Artificial Intelligence, Brighton, 1998.
[29] K. Stergiou, M. Koubarakis, Backtracking algorithms for disjunctions of temporal constraints, in: Proc. AAAI-98, Madison, WI, 1998.
[30] K. Stergiou, M. Koubarakis, Backtracking algorithms for disjunctions of temporal constraints, Artificial Intelligence 120 (1) (2000) 81–117.
[31] I. Tsamardinos, Reformulating temporal plans for efficient execution, Masters Thesis, University of Pittsburgh, Pittsburgh, PA, 1998.
[32] I. Tsamardinos, Constraint-based temporal reasoning algorithms with applications to planning, Ph.D. Thesis, Computer Science Department, University of Pittsburgh, Pittsburgh, PA, 2001.
[33] I. Tsamardinos, T. Vidal, M.E. Pollack, CTP: A new constraint-based formalism for conditional, temporal planning, Constraints 8 (2003), to appear.
[34] M. Yokoo, Weak-commitment search for solving constraint satisfaction problems, in: Proc. AAAI-94, Seattle, WA, 1994.