Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination

Chao Wang, Franjo Ivančić, Malay Ganai, and Aarti Gupta

NEC Laboratories America,
4 Independence Way, Princeton, NJ 08540, USA

Abstract. Separation logic is a subset of the quantifier-free first order logic. It has been successfully used in the automated verification of systems that have large (or unbounded) integer-valued state variables, such as pipelined processor designs and timed systems. In this paper, we present a fast decision procedure for separation logic, which combines Boolean satisfiability (SAT) with a graph based incremental negative cycle elimination algorithm. Our solver abstracts a separation logic formula into a Boolean formula by replacing each predicate with a Boolean variable. Transitivity constraints over predicates are detected from the constraint graph and added on a need-to basis. Our solver handles Boolean and theory conflicts uniformly at the Boolean level. The graph based algorithm supports not only incremental theory propagation, but also constant time theory backtracking without using a cumbersome history stack. Experimental results on a large set of benchmarks show that our new decision procedure is scalable, and outperforms existing techniques for this logic.

1 Introduction

Separation logic (also called difference logic) is a subset of the quantifier-free first order logic for which efficient decision procedures exist. It has been successfully used in the automated verification of systems that have large (or unbounded) integer-valued state variables, such as pipelined processor designs and timed systems. Since integer variables and arithmetic operators are not flattened into the bit vector format, separation logic can model and verify systems at a higher abstraction level than Boolean logic. The UCLID verifier [4], for instance, relies on the decision procedure for separation logic as its back-end engine.

A separation logic formula contains the standard Boolean connectives as well as separation predicates of the form $(v_i - v_j \leq c)$ where v_i, v_j are integer variables and c is an integer constant. The validity of a separation logic formula can be checked by translating it to an equi-satisfiable Boolean formula, which in turn is checked by a Boolean SAT solver. Many existing techniques took this approach to leverage the recent advances of Boolean SAT algorithms, with differences only in the timing of the transformation and in the Boolean encoding methods. In particular, they can be classified as either *eager* or *lazy* depending on when the transformation happens.

In the eager approaches [4, 18, 16, 19], separation logic formulae are converted to equi-satisfiable Boolean formulae in a single step. The two existing encoding methods used during the transformation are *small domain encoding* and *per constraint encoding*.

© Springer-Verlag Berlin Heidelberg 2005

In small domain encoding, integer variables and arithmetic operators are bit-blasted with a sufficiently large vector size. In per constraint encoding, the formula is abstracted by replacing each predicate with a Boolean variable, and then augmented by adding all possible transitivity constraints over predicates. In addition, a hybrid method can be used to combine the strength of these two encoding schemes. A previous experimental study [16] showed that per constraint encoding based approach is often faster than small domain encoding. However, the complete set of transitivity constraints is added in one shot regardless of whether they are needed or not.

In the lazy approaches [2, 1, 8, 9, 3], transitivity constraints are added only dynamically on a "need-to" basis to augment the Boolean skeleton. Whenever the assignment to the Boolean skeleton is not consistent with the separation predicates, a transitivity constraint is added to eliminate the inconsistency before SAT search is resumed. Lazy approaches exploit the fact that transitivity constraints are often highly redundant and some of them may never be needed in solving the validity problem.

Deciding separation logic is an NP-complete problem [13]. However, experience with Boolean SAT solvers shows that practically efficient search heuristics often exist even for NP-complete problems. For example, the recent advances of DPLL SAT solvers (Davis-Putnam-Logemann-Loveland [7]) have led to their widespread application in industry settings, e.g. in verification of pipelined microprocessors. The two technical breakthroughs responsible for much of the performance improvement are (1) conflict analysis based learning and non-chronological backtracking [17] and (2) watched literal based fast Boolean Constraint Propagation (BCP) [11, 10]. These two parts, however, remain the weak links in separation logic solvers based on the lazy approach.

In this paper, we propose a procedure for lazily deciding separation logic by combining a DPLL Boolean SAT procedure with an efficient graph algorithm in the style of recent SAT Modulo Theory (SMT) solvers. Our emphasis is on the efficient implementation of conflict analysis (for both Boolean and theory conflicts) and on the data structure that supports fast theory backtracking. Our method maintains and incrementally updates a constraint subgraph for all active separation constraints. The theory part only receives assignments from the Boolean part and detects conflicts; it does not perform exhaustive theory propagation nor feed back implications. Theory conflicts are removed by augmenting the Boolean formula with conflicting clauses. Our procedure is both sound and complete; it terminates as soon as a consistent assignment is found or all possible cases are explored.

A major contribution of this paper is our fast theory propagation and backtracking algorithm, which not only prunes theory constraints incrementally, but also performs constant time backtracking. Unlike the existing techniques in [2, 9, 3, 6], we do not need expensive book-keeping on the constraint graph for (non-chronological) backtracking, nor do we need a history stack to store any of its previous states. In fact an analogy exists between our graph-based constraint propagation (GCP) algorithm and the watched literal based Boolean constraint propagation (BCP) in Chaff [11], in that both have constant time backtracking.

In [3], an incremental and layered procedure for deciding linear arithmetic logic was proposed for the MathSat solver. It includes a separation logic solver based on incremental Bellman-Ford algorithm for detecting theory conflicts, but no further details of the algorithm are available in [3] or related papers. In particular, it is not clear how their theory backtracking is implemented and what the backtracking cost is.

The more recent work by Cotton [6] also has an incremental negative cycle detection algorithm, but is significantly different from ours in backtracking. In the broader area, the work by Ramalingam *et al.* [15] is the first dynamic algorithm for arbitrary edge weighted graphs that has a per edge complexity bound better than that of Bellman-Ford. The cycle detection algorithm in our approach has the same complexity bound as [15]. In addition to incremental cycle elimination, we propose several optimizations for its tighter integration with the Boolean SAT solver and for fast backtracking.

In [9], a DPLL(T) framework was proposed for SAT modulo theories, but including only EUF logic. Recently, the DPLL(T) approach has been extended to separation logic [12]. They perform exhaustive theory propagation, making the algorithm quite different from ours. We have implemented a variant of [12] on top of our own solver; our experiments show that this addition can further improve the performance of our solver on examples where theory conflicts play a larger role.

We also provide in this paper experimental comparisons of our solver with the latest versions of both DPLL(T) and MathSAT, as well as other solvers including ICS [8], UCLID [4], and TSAT++ [1]. The results show that our new algorithm outperforms these existing techniques, particularly on harder test cases.

The rest of the paper is organized as follows. We give technical background in Section 2, describing separation logic, the transformation to SAT, and the constraint subgraph. We then give the overall algorithm in Section 3. Our fast GCP and incremental negative-cycle detection algorithms are described in Section 4. We give experimental results in Section 6, and then conclude in Section 7.

2 Separation Logic

Definition 1. *A separation logic formula consists of the standard propositional connectives and predicates of the form $v_i - v_j \leq c$, where v_i and v_j are integer variables and c is a constant.*

To canonize the individual predicates, we impose an order on the integer variables such that $i \leq j$ for all constraints of the form $v_i - v_j \leq c$. Input formulae that do not meet this above requirement are normalized through rewriting, before they are given to the solver. For example, $(x - y > 5)$ is equivalent to $\neg(x - y \leq 5)$, while $(x - y < 5)$ is equivalent to $(x - y \leq 4)$. For predicates in the form of $x \leq c$, a common integer variable $ZERO$ can be added to encode the predicates into $(x - ZERO \leq c)$. Note that with the implicit order on all integer variables, predicates $(x - y \leq 5)$ and $(y - x \leq -6)$ are mapped to the same Boolean variable (P and $\neg P$) instead of two.

The validity of a separation formula can be checked by a Boolean SAT solver via transformation. The first step is to abstract the original formula ϕ into a Boolean skeleton ϕ_{bool}, by replacing separation predicates with fresh Boolean variables. Since transitivity constraints among predicates are removed, ϕ_{bool} has all the possible satisfying assignments of ϕ, and possibly more. Formula ϕ_{bool} is put into the Conjunctive Normal Form (CNF) before it is given to the SAT solver. A CNF formula is a conjunction of *clauses*, each of which is a disjunction of *literals*. A literal is a Boolean variable or its negation.

An example of a separation logic formula is given as follows,

$$\begin{array}{l}(x - y \leq 2 \vee x - z \leq 6) \wedge (x - y \leq 2 \vee \neg(x - z \leq 6)) \wedge \\ (\neg(x - y \leq 2) \vee y - z \leq 3) \wedge (\neg(x - y \leq 2) \vee \neg(y - z \leq 3) \vee w - y \leq 10) \\ (\neg(x - y \leq 2) \vee w - y \leq 10) \ ,\end{array}$$

where w, x, y and z are all integer variables. Note that this formula is already in the CNF format. After replacing the predicates by Boolean variables as follows,

$$A : (x - y \leq 2), B : (x - z \leq 6), C : (y - z \leq 3), D : (w - y \leq 10)$$

ϕ is abstracted into ϕ_{bool}:

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (\neg A \vee \neg C \vee D) \wedge (\neg A \vee D) \ .$$

Although the Boolean assignment $(A, \neg B, C, D)$ satisfies ϕ_{bool}, the set of corresponding separation constraints do not have a solution. In fact, $(x - y \leq 2 \wedge y - z \leq 3) \rightarrow (x - z \leq 5)$. To make the Boolean formula equi-satisfiable to ϕ, one must augment ϕ_{bool} with transitivity constraints among separation predicates to rule out inconsistent assignments. In the above example, we can derive the constraint $A \wedge C \rightarrow B$ to augment the Boolean skeleton.

A set of separation predicates can be mapped to a weighted directed graph, called the *constraint graph*. Every negative weight cycle in this graph represents a transitivity constraint.

Definition 2. *The constraint graph G of a set of separation predicates is a weighted directed graph whose vertices correspond to integer variables and whose edges correspond to predicates and their negations. In particular,* $(v_i - v_j \leq c)$ *corresponds to the edge* (v_j, v_i) *with weight c, and* $\neg(v_i - v_j \leq c)$ *corresponds to* (v_i, v_j) *with weight* $(-c - 1)$.

A *constraint subgraph* contains all the vertices but a subset of the edges of a constraint graph. A full or partial assignment to ϕ_{bool} induces a constraint subgraph, which has only those edges corresponding to the active constraints.

Theorem 1. *Let* G_s *be the constraint subgraph induced by a (partial) assignment to* ϕ_{bool}. *The assignment is consistent with the set of separation predicates if and only if* G_s *does not have a negative weight cycle.*

As an example, the constraint graph for the set of predicates $\{A, B, C, D\}$ is given in Figure 1. The positive and negative phases of each predicate are mapped to two different edges. Such a graph implicitly encodes all the possible transitivity constraints. The constraint subgraph corresponding to the assignment $(A, \neg B, C, D)$ is given in Figure 2, which has a negative weight cycle $(x \rightarrow z \rightarrow y \rightarrow x)$.

In the lazy approaches, transitivity constraints in G_s are added dynamically whenever they are needed. However, this requires a call to the negative cycle detection algorithm every time a full or partial assignment is found. A standard graph-based approach

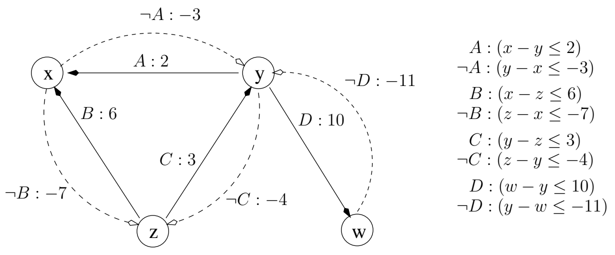

Fig. 1. Constraint graph for the predicate set $\{A, B, C, D\}$

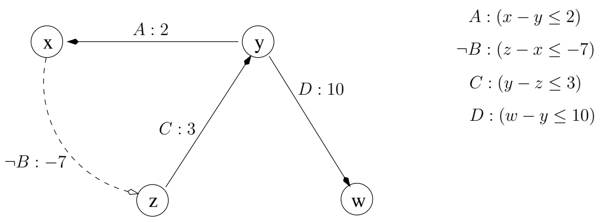

Fig. 2. Constraint subgraph induced by the assignment $(A, \neg B, C, D)$

for detecting negative cycles is the Bellman-Ford shortest path algorithm, which gives negative cycles as a by-product. In practice, the number of calls to a negative cycle detection procedure can be extremely large, therefore making it a potential bottleneck for lazy separation logic solvers.

3 SLICE: The New Solver

We present a new solver called SLICE (for **S**eparation **L**ogic solver with **I**ncremental **C**ycle **E**limination), which tightly integrates a DPLL style SAT procedure with a fast graph-based constraint propagation algorithm. The theory part in SLICE is kept quite passive. It only reports conflicts, but does not propagate implications back as in [12]. However, it is equipped with new data structures that support efficient propagation and constant time backtracking.

3.1 The Overall Algorithm

The overall algorithm of SLICE can be viewed as a modification of the DPLL procedure (Figure 3). It takes the Boolean skeleton ϕ_{bool} as input, and initializes the constraint subgraph G_s with all the vertices – one for each integer variable – but no edges. Procedure *decide()* picks one Boolean variable at a time and assigns it either true or false. When all Boolean variables are assigned and there is no conflict, it returns with SAT; if a conflict appears before any decision is made (i.e. the decision level is 0), we declare the formula UNSAT.

```
slice_sat() {
  while (1) {
    if (decide()) {
      while (slice_cp()==CONFLICT) {
        level = conflict_analysis();
        if (level < 0)
          return UNSAT;
        else
          back_track(level);
      }
    }else
      return SAT;
}

slice_cp() {
  if (bcp()==CONFLICT)
    return CONFLICT;
  else if (gcp()==CONFLICT)) { //propagate constraints on graph
    add_conflicting_clause();  //add new constraints on Boolean formula
    return CONFLICT;
  }else
    return NO_CONFLICT;
}
```

Fig. 3. SLICE: The new decision procedure for separation logic

SLICE only makes Boolean decisions. Implications of these decisions are propagated first by *bcp()* among Boolean clauses, then by *gcp()* in the constraint subgraph. Note that the passing of implications from Boolean to theory part is one-way; there is no feedback from *gcp()* to *bcp()*. BCP is based on unit implication, i.e. when all the other literals are set to false in a clause, the only remaining one must evaluate to true. GCP is based on the incremental negative cycle detection algorithm (details in Section 4). If either of them detects a conflict, we perform conflict analysis to locate the decision level at which the conflict is triggered. After adding a conflict clause to rule out the same assignment in the future, the procedure backtracks non-chronologically to the appropriate decision level and resumes the search. Procedure *slice_sat()* terminates as soon as a valid assignment is found or all possible cases have been explored.

3.2 Handling Conflicts

Conflicts from BCP and GCP are both handled at the Boolean level, by the same conflict analysis procedure. Our Boolean SAT solver is based on Chaff [11], which maintains an *implication graph* by recording the clause responsible for each implication (called the *antecedent*) and associating it with the implied variable. BCP detects a conflict when it finds a conflicting clause. During conflict analysis, we start from the conflicting clause and trace backward in the implication graph, to locate a proper cut-set (e.g. the 1st UIP in Chaff) between the decision nodes and the conflict. A *conflict clause* is then derived and added to the clause database, after which the procedure backtracks non-chronologically to the decision level where the conflict is triggered.

In GCP, we maintain a constraint subgraph to store all the active predicates, but do not maintain any data structure to store the implication relation. Every time a predicate is assigned at the Boolean level, its corresponding edge is scheduled to be added to the constraint subgraph. GCP starts adding and propagating edges only after BCP finishes, in order to amortize the cost of GCP (other heuristically driven schemes are also possible to change the ratio of calls to BCP and GCP). For each negative cycle detected during the propagation, it adds a conflicting clause whose literals are the negation of the edges on the negative cycle. Note that this particular call sequence guarantees that the added conflicting clause is always irredundant—otherwise, BCP would have detected the conflict. When we jump back to the Boolean level, the added conflicting clause enables us to perform conflict analysis and non-chronological backtracking using the same procedure, as if the conflict is detected during BCP.

We use the example in Section 2 to illustrate how conflicts are handled. Here we use $\neg D@L1$ to denote that Variable D is set to false at decision level 1.

– Assume that the SAT procedure makes the following decisions/implications,

$$\begin{array}{lll}
\neg D@L1; & & \text{decision} \\
 & \neg A@L1; & \text{due to } (\neg A \vee D) \\
 & \neg B@L1; & \text{due to } (A \vee \neg B) \\
 & (A \vee B) = false; & \text{conflict!}
\end{array}$$

Note that the first line is decision and the rest are implications. By tracing back from $(A \vee B)$, we find the 1st UIP ($\neg A@L1$), add the conflict clause (A), and backtrack to decision level 0. Backtracking restores all the assignments made to D, A and B.

– The added clause (A) forces the SAT procedure to flip the value of A,

$$\begin{array}{lll}
 & A@L0; & \text{due to } (A) \\
 & D@L0; & \text{due to } (\neg A \vee D) \\
 & C@L0; & \text{due to } (\neg A \vee C) \\
\neg B@L1; & & \text{satisfiable assignment!}
\end{array}$$

At this point, BCP finishes without detecting any conflict. This Boolean assignment induces the constraint subgraph in Figure 2. However, GCP finds a negative weight cycle due to $\{A, \neg B, C\}$, and adds a conflicting clause $(\neg A \vee B \vee \neg C)$. The added clause itself represents a conflict in the Boolean part, therefore triggers the 1st UIP conflict analysis. After adding a conflict clause (B), we backtrack again to decision level 0.

– The added clause (B) forces the SAT procedure to flip the value of B.

$$\begin{array}{ll}
A@L0; & \text{due to } (\neg A); \\
D@L0; & \text{due to } (\neg A \vee D); \\
C@L0; & \text{due to } (\neg A \vee C); \\
B@L0; & \text{due to } (B);
\end{array}$$

Another call to GCP confirms that this is a consistent assignment; therefore, the separation logic formula is satisfiable.

We should note that both the conflicting clauses added for negative cycles and the conflict clauses learned from conflict analysis can be made *volatile*; that is, they are allowed to be deleted. In many modern SAT solvers, periodically deleting redundant clauses has been helpful in solving hard SAT problems. The removal of conflict clauses does not affect the completeness of the SAT algorithm (for proof, please refer to [20]).

In practice, however, we choose to make the conflicting clauses added for negative cycles *non-volatile*, since they represent the constraints not yet contained in the original Boolean formula ϕ_{bool}. On the other hand, we make conflict clauses *volatile* since they are always redundant (though their existence may help prune the search space).

4 Negative Cycle Elimination

Let the constraint subgraph $G_s = (V, E)$ be a weighted directed graph, $w[u, v]$ be the weight of edge (u, v), and $d[v]$ be the cost of node v. The following statements are equivalent: (1) The set of separation constraints has a valid solution $\{d[v]\}$; and (2) there is no negative weight cycle in the corresponding constraint subgraph.

Bellman-Ford solves the single-source shortest-paths problem in graphs where edge weights can be negative; as a by-product, it also detects negative-weight cycles that are reachable from the source (cf. [5]). Although several separation logic solvers use Bellman-Ford to detect theory conflicts, it is not very suitable for a tight on-line integration with the Boolean SAT solver. This is especially true when the cycle detection algorithm must be called every time a predicate is assigned. In such a case, even making Bellman-Ford incremental is not very effective. However, studying Bellman-Ford does shed some light on how an efficient theory solver can be implemented.

The basic operation in searching for a solution is **relax**, which operates on edges as shown below. Here $pi[v]$ represents the edge responsible for the last change to $d[v]$; it can be used to retrieve the negative weight cycles.

```
relax (u,v) {
   if (d[v] > d[u] + w[u,v]) {
     d[v]  = d[u] + w[u,v];
     pi[v] = (u,v);
   }
}
```

An edge is *stable* if relax does not change the cost of its sink node. A solution is found when all edges are stable. Each solution $\{d[v]\}$ represents a class of solutions $\{d[v] + c\}$, since $(d[v] \leq d[u] + w[u, v])$ implies $(d[v] + c \leq d[u] + c + w[u, v])$. If a solution exists, all edges will become stable after a bounded number of relaxing operations. When there is no solution (i.e. some negative cycles exist), some edges can never become stable. This is the basis of many existing negative cycle detection algorithms, including Bellman-Ford.

However, the original Bellman-Ford algorithm runs $n \times m$ relax operations (where n and m are the number of nodes and edges, respectively) before checking whether all edges are stable. The first optimization is to stop relaxing as soon as all edges are stable, or to stop as soon as possible in the presence of negative cycles.

Bellman-Ford returns more information than needed for negative cycle detection or finding an arbitrary solution. Assume that $Ax \leq b$ is a system of m separation constraints in n integer variables, Bellman-Ford algorithm gives a solution that maximizes $\sum_{i=1}^{n} x_i$ subject to $Ax \leq b$ and $x_i \leq 0$ for all x_i ([5]). We recognize and exploit the fact that if the purpose is to search for an arbitrary solution or simply to detect negative cycles, we can use an arbitrary set of initial node values as the starting point. Note that the proof follows Pratt's theorem in [14] (and also in [5]).

Proposition 1. *For the purpose of detecting negative weight cycles, Bellman-Ford is sound and complete by starting with an arbitrary set of initial node values (instead of initializing $d[v]$ to ∞).*

Although the initial node values do not affect the correctness of the algorithm, they do affect the run-time in practice. Typically, the closer $\{d[v]\}$ is to a solution, the less effort is needed for the relaxing phase to converge. For example, if the current $\{d[v]\}$ is already a solution, then no edge needs to be relaxed. Our new GCP algorithm exploits this fact by updating the subgraph incrementally.

Let the set $\{d_i[v]\}$ be the stable node values after adding the i-th edge. The key invariant to our negative cycle detection algorithm is given as follows:

Theorem 2. *If no conflict is detected by the previous call to negative cycle detection, all edges in the subgraph must have been stable. Therefore, the set $\{d_i[v]\}$ of node values is always a valid solution to the current set of separation constraints.*

Since there is no negative cycle in the subgraph, if adding a new edge creates one, the cycle must go through the new edge. In the relaxing phase, if the new edge is relaxed more than once, we declare it as a conflict.

The algorithm is given in Figure 4. Initially, the constraint subgraph contains all the nodes but no edge. Each time a separation predicate is assigned a value, the corresponding edge is scheduled to be added. After each SAT decision (and after BCP finishes), we search for negative weight cycles in the subgraph. Starting from the newly added edge (u, v), we propagate the value of the separation predicate. If all edges eventually

```
gcp()
{
  for each predicate assigned at current level {
    added edge (u,v);
    if (detect_negative_cycle(u,v))
        return CONFLICT;
  }
  return NO_CONFLICT;
}

detect_negative_cycle(u,v)
{
  if (d[v]>d[u]+w[u,v]) {
    relax (u,v);
    enqueue(v);
  }
  while ((x=dequeue())!=NULL) {
    for each edge (x,y) {        // sequenced with priority queue
      if (d[y]>d[x]+w[x,y]) {
        if (u==x && v==y)
          return TRUE;
        else {
          relax (x,y);
          enqueue(y);
        }
      }
    }
  }
  return FALSE;
}
```

Fig. 4. Incremental negative cycle detection algorithm

become stable, the FIFO queue becomes empty, meaning that there is no negative cycle. If there exists a negative cycle, the cycle must go through edge (u, v); therefore we can detect it when node v is visited again during the constraint propagation. The cycle can be retrieved by following $pi[v]$ all the way back to edge (u, v).

Given a constraint subgraph with n nodes and k edges, the detection algorithm can run in $O(n\ log\ n + k)$ time per added separation predicate. Since all edges are stable before adding (u, v), we can sequence our relaxation operations with a Fibonacci heap based priority queue ordering nodes according to their maximal node value changes [15] and [6]. If there is no negative weight cycle even after adding (u, v), relaxing will converge after going through those nodes exactly once. However, it is worth pointing out that this worst-case complexity bound seldom reflects the performance of the algorithm in practice.

Unlike Bellman-Ford which recomputes node values each time from scratch, our new algorithm propagates the constraints incrementally. Since all existing edges are already stable before the addition of the new edge, the number of edges that need to be relaxed is often significantly reduced. For example, if the new edge is already stable under the previous $\{d_j[v]\}$ (i.e. node values at the j-th decision level), then no propagation is needed; if the new edge is not stable but $\{d_j[v]\}$ is already very close to a solution, then not many edges need to be relaxed. Data in Section 6 show that the reduction in the number of relax operations can be several orders of magnitude.

5 Efficient Backtracking

Efficient implementation of backtracking on the theory part is important since in practice the number of backtracks is often very large. This imposes two constraints on designing a backtracking algorithm: First, it should have low runtime overhead; second, it should be scalable in terms of memory usage. For instance, the approach of storing the theory solver's states at all previous decision levels in a history stack does not scale well in practice. In SLICE, we do not need such a history stack, and we do not need to restore the theory solver's state either, even during non-chronological backtracking.

Indeed, the invariant maintained by our algorithm makes a constant-time backtracking possible. Note that in Chaff's two-literal watch list based BCP, backtracking in the Boolean part has already been made a constant time operation – Chaff does not update during backtracking any of the affected clauses and their watched literals. Similarly, in SLICE we do not need to update (or restore) any of the node values; the procedure remains sound and complete as long as all existing edges are stable before every call to negative cycle detection. We shall show in the following that this invariant is maintained throughout the solving process.

First, the invariant always holds when we add edges to the subgraph and there is no conflict in either BCP or GCP. Let $\{d_j[v]\}$ be the node values at the j-th decision level. If no conflict is detected, $\{d_j[v]\}$ is a solution to the set of separation constraints after the call to negative cycle detection. Furthermore,

Theorem 3. $\{d_j[v]\}$ *is also a solution for the set of separation constraints at any previous decision level* i *such that* $i \leq j$.

This is because constraint subgraphs at previous levels contain subsets of these edges—a solution remains valid when some constraints are dropped. We should note that multiple edges can be added at each decision level, and a conflict detected in GCP is guaranteed to involve at least one assignment at the current decision level.

Second, if backtracking from decision level j to i is triggered by a conflict in BCP, the node values right before backtracking are $\{d_{j-1}[v]\}$ (since GCP has not been performed yet). The only thing we need to do is to delete edges added after decision level i. However, we *do not have to* restore the node values from $\{d_{j-1}[v]\}$ back to $\{d_i[v]\}$. More often than not, $\{d_{j-1}[v]\}$ is a better solution than $\{d_i[v]\}$ since it satisfies more separation constraints. In practice, relaxation of edges will be avoided later if some of the deleted edges are added back.

Third, if backtracking from decision level j to i is triggered by a conflict in GCP, by the time we detect the negative cycle (i.e. edge (u, v) is revisited), $\{d_j[v]\}$ may no longer be a valid solution (because some edges may still need to be relaxed). We have two choices in restoring the invariant. If we keep relaxing the edges other than (u, v) until convergence, we will get a set $\{d_j[v]\}$ that is a solution at the previous level. However, if we want to stop the propagation as soon as the first conflict is detected, backtracking is no longer constant-time since we need to restore a valid solution. We can record the node value changes during the current cycle detection call and restore them as soon as we detect the first negative cycle. Note that only local changes in the current call need to be recorded (as opposed to all the solver states between level i and level j), even when the backtracking is non-chronological. Finally, none of these two choices affects the worst-case complexity of negative cycle detection.

The Working Example. Figure 5 shows the constraint subgraphs at different stages of applying our GCP algorithm. We use the same separation logic formula (from Section 2) as an example. The initial subgraph is given at the left top, in which all node are initialized to 0. The subgraph at the right top is after the partial assignment $(\neg D, \neg A, B)$; note that no constraint propagation is needed when the edges (z, x) and (x, y) are added, because they are already stable under the existing node values after (w, y) is added. When backtracking from this partial assignment, we only delete the three edges while leaving the node values unchanged. The right bottom subgraph is under the assignment $(A, \neg B, C, D)$, which has a negative weight cycle. After backtracking and setting B true, the subgraph is shown at the left bottom. At this point, all Boolean variables are assigned and there is no conflict, the separation formula is proved to be satisfiable. Note that the set of $\{d[v]\}$ values is a solution to the current set of separation constraints.

6 Experiments

We have implemented our new decision procedure on top of the zChaff SAT solver, by integrating the incremental negative cycle elimination algorithm with the DPLL based SAT search. During the implementation of our graph algorithm, effort has been made to make sure that both adding and deleting an edge take constant time.

We have conducted experiments with a set of 385 public benchmark formulae generated from verification problems and scheduling problems. It includes 159 formulae of the MathSAT suite, 99 of the SAL suite, 31 of the DLSAT suite, 60 DTP formulae, and 36 *diamonds* formulae. All the experiments were run on a workstation with 3.0 GHz Intel Pentium 4 processor and 2 GB of RAM running Red Hat Linux 7.2. We set the time limit to 3600 seconds and the memory limit to 1 GB.

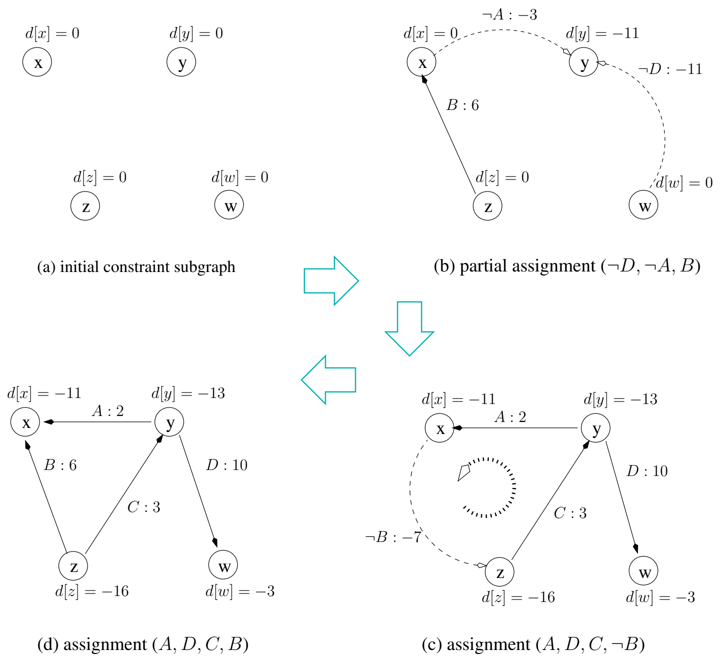

Fig. 5. Applying the graph based constraint propagation

Table 1 compares SLICE's Incremental Negative Cycle Detection with Bellman-Ford. Columns 1-3 show for each set of formulae the suite name, the category and the number of formulae. Column 4 gives the average percentage of non-Boolean variables (or separation predicates). Columns 5-8 are from SLICE runs with incremental cycle detection, which include the average percentage of GCP generated conflicts, the ratio of BCP calls to GCP calls, the percentage of CPU time spent in GCP, and the average number of relaxed nodes per negative cycle detection call. Columns 9-10 are from solver runs with Bellman-Ford, which include the information on CPU time and the number of relaxed nodes per call to Bellman-Ford. Note that only two columns are presented for Bellman-Ford, because the percentage of GCP conflicts and the BCP/GCP ratio stay roughly unchanged with both cycle detection algorithms.

The data show that our incremental graph algorithm significantly reduces the overhead of GCP. Compared to Bellman-Ford, the reduction in the number of relax operations can be several orders of magnitude. In fact, except for *diamonds*, the number of nodes relaxed per call have been reduced to single digit or less. The hand-made *diamonds* formulae [18] are known to have exponential number of negative cycles, each of which contains half of the separation constraints in the formulae.

Table 1. Comparison of Incremental Negative Cycle Detection and Bellman-Ford

Benchmarks				Data from SLICE runs					
suite name				Incremental cycle detection				Bellman-Ford	
		num. of formulae	non-Boolean vars (%)	conflicts in GCP (%)	num. of BCP/GCP	time in GCP (%)	num. of relax	time in GCP (%)	num. of relax
mathsat	FISCHER	119	30	1	20	8	2	46	17
	PO2	7	40	2	16	0	1	14	13
	PO3	9	30	1	14	16	0.4	25	9
	PO4	11	20	1	13	9	0.3	46	5
	PO5	13	13	1	13	4	0.2	57	4
sal	lpsat	20	13	1	10	10	2	62	49
	inf-bak	20	50	32	7	30	3	70	294
	fischer	59	60	12	21	18	7	80	1186
DLSAT	abz5	12	100	32	12	55	7	49	1152
	ba-max	19	13	22	8	25	4	84	233
DTP		60	100	62	8	47	0.4	89	205
diamonds		36	100	3	2	66	79	89	1101

We have also conducted experimental comparison of our new algorithm with other state-of-the-art tools, including UCLID, MathSAT, ICS, TSAT++, and DPLL(T). For all tools, their latest public available released versions were used. For DPLL(T), it includes their latest development as described in [12]. For UCLID we used the default "hybrid method" which combines the strengths of per constraint and small-domain encoding. The overall result is given in scatter plots in Figure 6. Here the x-axis is the CPU time of SLICE, while the y-axis is the CPU time for other solvers. For DPLL(T), which is the closest competitor on this set of benchmarks, we also give the scatter plot in linear scale.

The result shows that SLICE performs significantly better than UCLID, MathSAT, and ICS on the majority of the benchmarks. The only cases on which UCLID runs faster are some smaller *diamonds* formulae. However, SLICE finishes all the 36 *diamonds* formulae within 1 hour, but UCLID times out on 8 larger ones. ICS 2.0 runs faster than SLICE on several formulae from the MathSAT suite, although overall ICS 2.0 is much less robust. The comparison with TSAT++ shows that SLICE performs significantly better on most cases. DPLL(T) is the closest competitor to SLICE on this set of benchmarks. However, as is shown by the last scatter plot, SLICE tends to do better on harder cases, therefore seems to be more robust and scalable.

Note that in most of these benchmark examples the percentage of GCP conflicts is very low, which indicates that computing all theory consequences as in [12] will not pay off. We have also implemented in our solver a variant of the exhaustive theory propagation technique of [12], which spends a limited (but not exhaustive) amount of effort in deriving theory implications. We then conducted controlled experiments on a set of randomly generated DTP formulae; in these formulae, the number of integer variables and separation constraints can be carefully controlled. (Due to space limit, we omit the result table.) Our experiments show that on examples in which GCP conflicts play a larger role, spending a limited amount of effort in deriving theory implications can significantly improve the performance of SLICE.

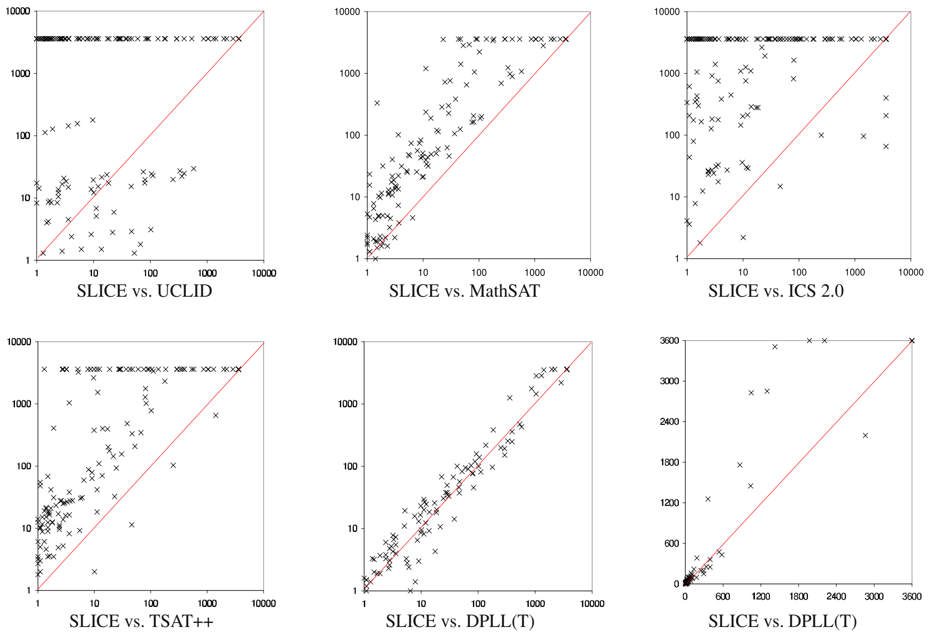

Fig. 6. Performance comparison in scatter plots: The CPU time is in seconds. The x-axis is for SLICE. Comparison with DPLL(T) is also shown in the linear scale.

7 Conclusions

We have presented a fast decision procedure for separation logic, which has an efficient theory engine for incremental conflict detection and constant time backtracking. The graph based theory solver allows fast backtracking without any additional bookkeeping. Controlled experiments indicate that the incremental algorithm is superior to the naive approach of Bellman-Ford; it significantly reduces the overhead of graph based constraint propagation. Performance evaluation on a set of public benchmarks shows that our new solver significantly outperforms leading separation logic solvers. For future work, we want to investigate more efficient ways of handling equality and inequality relations than translating them into separation predicates.

References

[1] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: an open platform for satisfiability modulo theories. In *Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[2] C. Barrett, D. L. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer Aided Design*, November 1996. LNCS 1166.

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. An incremental and layered procedure for the satisfiability of linear arithmetic logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 317–333, 2005. LNCS 3440.

[4] R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *Computer Aided Verification*, July 2002. LNCS 2404.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[6] S. Cotton. Satisfiability checking with difference constraints. Msc thesis, IMPRS Computer Science, Saarbrucken, 2005.

[7] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[8] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In *Computer Aided Verification*, pages 246–249, July 2001. LNCS 2102.

[9] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *Computer Aided Verification*, pages 175–188, July 2004. LNCS 3114.

[10] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe (DATE'03)*, pages 142–149, March 2002.

[11] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, June 2001.

[12] R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Computer Aided Verification*, pages 321–334, July 2005. LNCS 3576.

[13] A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):275–293, October 2002.

[14] V.R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

[15] G. Ramalingam, J. Song, L. Joscovicz, and R.E. Miller. Solving difference constraints incrementally. *Algorithmica*, 23(3):261–275, 1999.

[16] S. A. Seshia, S. K. Lahiri, and R. E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proceedings of the Design Automation Conference*, pages 425–430, June 2003.

[17] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227, November 1996.

[18] O. Strichman, S. A. Seshia, and R. E. Bryant. Deciding separation formulas with SAT. In *Computer Aided Verification*, pages 209–222, July 2002. LNCS 2404.

[19] M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range allocation for separation logic. In *Computer Aided Verification*, pages 148–161, July 2004. LNCS 3114.

[20] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, March 2003.