Exploiting Sparsity in Difference-Bound Matrices

Graeme Gange[1(✉)], Jorge A. Navas[2], Peter Schachte[1], Harald Søndergaard[1], and Peter J. Stuckey[1]

[1] Department of Computing and Information Systems, The University of Melbourne, Melbourne, VIC 3010, Australia
gkgange@unimelb.edu.au
[2] NASA Ames Research Center, Moffett Field, CA 94035, USA

Abstract. Relational numeric abstract domains are very important in program analysis. Common domains, such as Zones and Octagons, are usually conceptualised with weighted digraphs and implemented using difference-bound matrices (DBMs). Unfortunately, though conceptually simple, direct implementations of graph-based domains tend to perform poorly in practice, and are impractical for analyzing large code-bases. We propose new DBM algorithms that exploit sparsity and closed operands. In particular, a new representation which we call split normal form reduces graph density on typical abstract states. We compare the resulting implementation with several existing DBM-based abstract domains, and show that we can substantially reduce the time to perform full DBM analysis, without sacrificing precision.

1 Introduction

Relational numeric abstract domains are an important and large class, ranging from the highly precise polyhedral domain [9] to cheaper but less expressive variants, such as Octagons [18], Zones (or difference-bound matrices, DBMs) [17], and others. They share the advantage over "non-relational" domains that they support the extraction of important runtime relationships between variables. However, for large code bases, even the cheaper relational domains tend to be too expensive to use [10,21], so the usual compromise is to use less expressive (and cheaper) non-relational domains or if possible weakly relational domains tailored to specific applications (e.g., pentagons [16]).

During program analysis there are, however, characteristics of typical program states that we would hope to take advantage of. For example, the relations among variables are often quite sparse, variables tend to settle into disjoint clusters, and many operations in the analysis change only a small subset of the overall set of relations. Moreover, knowledge about typical analysis workflow, which analysis operations are more frequent, which normally succeed which, and so on, can be exploited. Indeed, there have been previous attempts to capitalize on such observations. We discuss related work in Sect. 7.

X. Rival (Ed.): SAS 2016, LNCS 9837, pp. 189–211, 2016.
DOI: 10.1007/978-3-662-53413-7_10

We propose a better *Zones* implementation. With *zones*, program state descriptions take the form of conjunctions of constraints $y - x \leq k$ where x and y are variables and k is some constant. Our implementation is optimized for the commonly occurring case of sparse systems of constraints. We follow a well-established tradition of representing the constraints as weighted directed graphs, and so the analysis operations are reduced to various graph operations.

We assume the reader is familiar with abstract interpretation [6,7] and with graph concepts and algorithms, including the classical shortest-path algorithms. This paper offers the following novel contributions:

- We design new data structures and algorithms for Zones. These are based on refined shortest-path graph algorithms, including a specialised incremental variant of Dijkstra's algorithm, which we call Chromatic Dijkstra (Sect. 3).
- We identify an important source of waste inherent in the standard graph representations of difference constraints (namely, non-relational properties are treated on a par with relational properties, contributing unnecessary density). To fix this we introduce a *split normal form* for weighted digraphs, which preserves many of the properties of transitive closure while avoiding unwanted "densification". We show how to modify the previous algorithms to operate on this form (Sect. 4).
- We propose a graph representation that uses "sparse sets" [2], tailored for efficient implementation across all the abstract operations (Sect. 5).
- We present an experimental evaluation of our implementation of Zones (Sect. 6) and conclude that scalable relational analysis is achievable.

2 Zones, DBMs and Difference Logic

Much of what follows deals with weighted directed graphs. We use $x \xrightarrow{k} y$ to denote a directed edge from x to y with weight k, and $E(x)$ the set of outgoing edges from x in E. $wt_E(x, y)$ denotes the weight of the edge $x \to y$ in E (or ∞ if absent). This is generalized for a directed path $x_1 \xrightarrow{k_1} \ldots \xrightarrow{k_{n-1}} x_k$ as $wt_E(x_1, \ldots, x_k)$ denoting its path length (i.e., $\sum_{i=0}^{n-1} k_i$). We may write a graph as its set of edges.[1] When we take the union of two sets of edges E_1 and E_2, we take only the minimum-weight edge for each pair of end-points.

In many cases it will be useful to operate on a transformed *view* of a graph. $rev(E)$ denotes the graph obtained by reversing the direction of each edge in E (so $x \xrightarrow{k} y$ becomes $y \xrightarrow{k} x$). $E \setminus \{v\}$ is the graph obtained by removing from E all edges incident to v. Note that these graphs are never explicitly constructed; they merely define different interpretations of an existing graph.

Difference-bound matrices or *Zones* [17] approximate concrete states by predicates of the forms $x \leq k$, $x \geq k$, and $y - x \leq k$, where x and y are variables,

[1] For presentation purposes, we assume all program states share a fixed set V of variables. In practice, this is unnecessarily expensive—we instead maintain vertices for only the variables that are in scope, and add or remove vertices as needed.

and k some constant. Constraints $y - x \geq k$ can be translated to the $\leq$ form, and, assuming variables range over the integers[2], so can strict inequality.

The abstract states (systems of constraints) are typically represented as a weighted graph, where $v \in V \cup \{v_0\}$ is associated with a vertex, and a constraint $y - x \leq k$ is encoded as an edge $x \xrightarrow{k} y$. The added vertex v_0 represents the constant 0. Relations are composed by adding lengths along directed paths; transitive closure is obtained by computing all pairs of shortest paths in the graph. In the rest of the paper, the terms *closure* and *closed* will refer to transitive closure. G^* denotes the closure of some graph G.

Example 1. Consider the system of constraints $\{x \in [0,1], y \in [1,2], y - z \leq -3\}$. The corresponding constraint graph is shown in Fig. 1(a). Note that interval constraints $x \in [lo, hi]$ can be encoded as edges $v_0 \xrightarrow{hi} x$ and $x \xrightarrow{-lo} v_0$. □

We shall use $E_1 \oplus E_2$ and $E_1 \otimes E_2$ to denote the pointwise maximum and minimum over a pair of graphs. That is:

$$E_1 \oplus E_2 = \{x \xrightarrow{k} y \mid x \xrightarrow{k_1} y \in E_1 \wedge x \xrightarrow{k_2} y \in E_2 \wedge k = \mathbf{max}(k_1, k_2)\}$$

$$E_1 \otimes E_2 = \left\{ x \xrightarrow{k} y \;\middle|\; \begin{array}{l} (x \xrightarrow{k} y \in E_1 \wedge k \leq wt_{E_2}(x,y)) \vee \\ (x \xrightarrow{k} y \in E_2 \wedge k < wt_{E_1}(x,y)) \end{array} \right\}$$

In typical implementations of difference bound-based domains, the system of relations is encoded as a dense matrix, and closure is obtained by running the Floyd-Warshall algorithm.

The *language* of "difference logic" also appears in an SMT context: SMT(DL) problems query satisfiability over Boolean combinations of difference constraints. The problem of solving SMT(DL) instances is rather different to the problem of static analysis using DBMs, since the former does not compute *joins*—a major issue in the latter. Nevertheless, we take inspiration from SMT(DL) algorithms, in particular Cotton and Maler's approach to dealing with sparse systems of difference constraints [5]. In this approach the graph is augmented with a *potential function* π—a model of the constraint system. When an edge is added, π is revised to find a model of the augmented system. Given a valid potential function, the corresponding concrete state is $\{x \mapsto \pi(x) - \pi(v_0) \mid x \in V\}$. We extend π naturally: $\pi(e)$ denotes the value of expression e under assignment π.

Maintaining the potential function π allows the graph to be reformulated—for any constraint $y - x \leq k$, the *slack* (or *reduced cost* [5]) is given by $slack(y,x) = \pi(x) + k - \pi(y)$. As π is a model, this is non-negative; so shortest paths in the reformulated graph may be computed using Dijkstra's algorithm.

Example 2. Consider again the constraints captured in Fig. 1. Let the potential function $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9\}$. This corresponds to the concrete assignment $\{x \mapsto 1, y \mapsto 1, z \mapsto 7\}$ (as v_0 is adjusted to 0). The slack graph

[2] Our approach works for rational numbers as well. The implementation assumes 64-bit integers and does not currently take over-/under-flow into account.

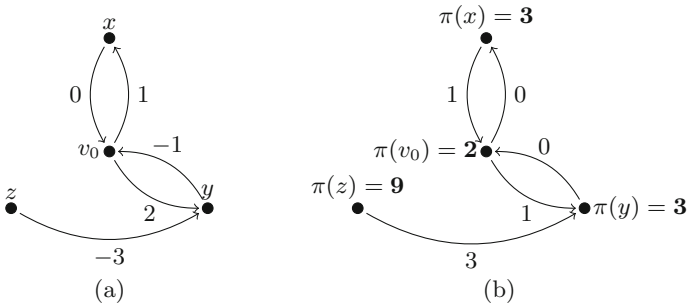


Fig. 1. (a) A graph representing the set of difference constraints $\{x \in [0,1], y \in [1,2], y - z \leq -3\}$. (b) The *slack* graph (with all non-negative weights) under potential function $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9\}$.

under π is given in Fig. 1(b). As every constraint is satisfied by π, all weights in the reformulated graph are non-negative.

If we follow the shortest path from z to y in (b), we find the slack between z and y is 3. We can then invert the original transformation to find the corresponding constraint; in this case, we get $y - z \leq \pi(y) - \pi(z) + slack(z, y) = -3$, which matches the original corresponding path in (a). □

3 Zones Implemented with Sparse DBMs

A critical step for precise relational analysis is computing the closure of a system of relations, that is, finding all implied relations and making them explicit. Unfortunately, this also vastly dominates DBM runtimes [16,20,21]. Closure using Floyd-Warshall is $\Theta(|V|^3)$. In a sparse graph we can use Johnson's algorithm [13] to reduce this to $O(|V||E| + |V|^2 \log |V|)$, but this is still not ideal.

Typical manipulations during abstract interpretation are far from random. The operations we perform most frequently are assigning a fresh variable, forgetting a variable, adding a relation between two variables, and taking the disjunction of two states. Each of these exhibits some structure we can exploit to maintain closure more efficiently. On occasion we also need to perform conjunction or widening on abstract states; we discuss these at the end of this section.

In the following, an abstract state φ consists of a pair $\langle \pi, E \rangle$ of a potential function π, and a sparse graph of difference constraints E over vertices $V \cup \{v_0\}$. Potentials are initially 0. We assume the representation of E supports cheap initialization, and constant time insertion, lookup, removal and iteration; we discuss a suitable representation in Sect. 5.

3.1 Join

For $\langle \pi_1, E_1 \rangle \sqcup \langle \pi_2, E_2 \rangle$, both π_1 and π_2 are valid potentials, so we can choose either. We then collect the pointwise maximum $E_1 \oplus E_2$. If E_1 and E_2 are closed

then so is $E_1 \oplus E_2$, and the overall result is simply $\langle \pi_1, E_1 \oplus E_2 \rangle$. Assuming we can lookup a specific edge in constant time, this takes worst case $\mathrm{O}(\mathbf{min}(|E_1|, |E_2|))$.

3.2 Variable Elimination

To eliminate a variable, we simply remove all edges incident to it. Assuming a specific edge is removed in constant time, this takes $\mathrm{O}(|V|)$ worst case time.

3.3 Constraint Addition

To add a single edge $x \xrightarrow{k} y$, we exploit an observation made by Cotton and Maler [5]: Any newly introduced shortest path must include $x \to y$. The potential repair step is unchanged, but our need to maintain closure means the rest of the algorithm differs somewhat. The closure process is given in Fig. 2.

The potential repair step has worst-case complexity $\mathrm{O}(|V| \log |V| + |E|)$, and restoring closure is $\mathrm{O}(|V|^2)$. This worst-case behaviour can be expected to be rare—in a sparse graph, a single edge addition usually affects few shortest paths.

```
add-edge(⟨π, E⟩, e)
    E' := E ∪ {e}
    π' := restore-potential(π, e, E')
    if π' = inconsistent
        return ⊥
    return ⟨π', E' ∪ close-edge(e, E')⟩

close-edge(x --k--> y, E)
    S := D := ∅
    δ := ∅
    for (s --k'--> x ∈ E)
        if k' + k < wt'_E(s, y)
            δ := δ ∪ {s --k'+k--> y}
            S := S ∪ {s}
    for (y --k'--> d ∈ E)
        if k + k' < wt'_E(y, d)
            δ := δ ∪ {x --k+k'--> d}
            D := D ∪ {d}
    for (s ∈ S, d ∈ D)
        if wt'_E(s, x, y, d) < wt'_E(s, d)
            δ := δ ∪ {s --wt'_E(s,x,y,d)--> d}
    return δ
```

Fig. 2. Algorithm for restoring closure after addition of $x \xrightarrow{k} y$.

3.4 Assignment

The key to efficient assignment is the observation that executing $[\![\mathtt{x} := \mathtt{S}]\!]$ can only introduce relationships between $\mathtt{x}$ and other variables; it cannot tighten any existing relation.[3] From the current state $\varphi = \langle \pi, G \rangle$, we can compute a valid potential for x simply by evaluating S under π.

We then need to compute the shortest distances to and from x (after adding edges corresponding to the assignment). As π is a valid potential function, we could simply run two passes of Dijkstra's algorithm to collect the consequences.

Example 3. Consider again the state shown in Fig. 1(a). Its closure is shown in Fig. 3(a). To evaluate $[\![\mathtt{w} := \mathtt{x} + \mathtt{z}]\!]$ we compute a valid potential for w from potentials for x and z: $\pi(w) = \pi(v_0) + (\pi(x) - \pi(v_0)) + (\pi(z) - \pi(v_0)) = 10$. Replacing x and z with their bounds, we derive these difference constraints: $\{w \geq 4, x - w \leq -4, z - w \leq 0, w - z \leq 1\}$. These edges are shown in Fig. 3(b). Running a Dijkstra's algorithm to/from w, we also find the transitive $w \xrightarrow{-3} y$ edge (corresponding to $y - w \leq -3$). □

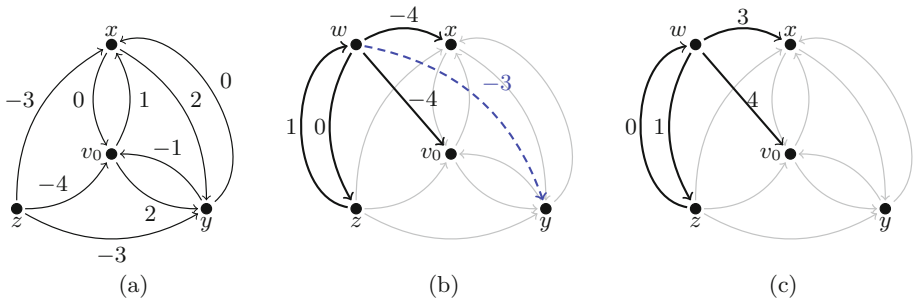


Fig. 3. (a) Closure of the state from Fig. 1(a). (b) Edges after evaluating $[\![\mathtt{w} := \mathtt{z} + \mathtt{y}]\!]$. (c) Edges introduced in Example 4, re-cast in terms of slack.

But as G is closed we can do better. Assume a shortest path from x to z passes through $[x, u_1, \ldots, u_k, z]$. As G is closed there must be some edge (u_1, z) such that $wt_G(u_1, z) \leq wt_G(u_1, \ldots, u_k, z)$; thus, we never need to expand grandchildren of x. The only problem is if we expand immediate children of x in the wrong order, and later discover a shorter path to a child we've already expanded.

However, recall that π allows us to reframe G in terms of slack, which is non-negative. If we expand children of x in order of increasing slack, we will never find a shorter path to an already expanded child. Thus we can abandon the priority queue entirely, simply expanding children of x by increasing slack, and collecting the minimum distance to each grandchild. The algorithm for restoring closure

[3] This assumes $\pi(\mathtt{S})$ is total. For a partial function like integer division we first close with respect to $\mathtt{x}$, then enforce the remaining invariants.

after an assignment is given in Fig. 4. Its worst-case complexity is $O(|S| \log |S| + |E|)$. The assignment $[\![\mathtt{x} := \mathtt{S}]\!]$ generates at most $2|S|$ immediate edges, which we must sort. We then perform a single pass over the grandchildren of x. In the common case where $|S|$ is bounded by a small constant, this collapses to $O(|E|)$ (recall that $rev(E)$ is not explicitly computed).

```
close-assignment-fwd(⟨π, E⟩, x)
    reach(v) := 0 for all v
    reach(x) := 1; Dist(x) := 0; adj := ∅
    for each x -k-> y ∈ E(x) by increasing k − π(y)
        if reach(y)
            Dist(y) := min(Dist(y), k)
        else
            adj := adj ∪ {y}
            reach(y) := 1; Dist(y) := k
        for (y -k'-> z ∈ E(y))
            if reach(z)
                Dist(z) = min(Dist(z), Dist(y) + k')
            else
                adj := adj ∪ {z}
                reach(z) := 1; Dist(z) := Dist(y) + k'
    return {x -Dist(y)-> y | y ∈ adj, Dist(y) < wt_E(x, y)}

close-assignment(⟨π, E⟩, x)
    δ_f := close-assignment-fwd(⟨π, E⟩, x)
    δ_r := close-assignment-fwd(⟨−π, rev(E)⟩, x)
    return δ_f ∪ rev(δ_r)

eval-expr(π, S)
    match S with
        c: return c + π(v_0) % constant
        x: return π(x) − π(v_0) % variable
        f(s_1, ..., s_k): % arithmetic expression
            for (i ∈ {1, ..., k})
                e_i := eval-expr(π, s_i)
            return f(e_1, ..., e_k)

assign(⟨π, E⟩, [[x := S]])
    π' := π[x ↦ π(v_0) + eval-expr(π, S)]
    E' := E ∪ edges-of-assign(E, [[x := S]])
    δ := close-assignment(⟨π', E'⟩, x)
    return ⟨π', E' ⊗ δ⟩
```

Fig. 4. Updating the abstract state under an assignment

Example 4. Recall the assignment in Example 3. The slack graph, with respect to $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9, w \mapsto 10\}$, is shown in Fig. 3(c). Processing outgoing edges of w in order of increasing slack, we first reach z, marking v_0, x and y as reached, with $Dist(v_0) = -4, Dist(x) = -3$ and $Dist(y) = -3$. We then process x, which is directly reachable at distance $Dist(x) = -4$, but find no other improved distances. After finding no improved distances through v_0, we walk through the vertices that have been touched and collect any improved edges, returning $\{y - w \leq -3\}$ as expected. □

3.5 Meet

The meet operation $\langle \pi_1, E_1 \rangle \sqcap \langle \pi_2, E_2 \rangle$ is more involved. We first collect each relation from $E_1 \cup E_2$, but we must then compute an updated potential function, and restore closure. The algorithm is outlined in Fig. 5.

```
meet(⟨π1, E1⟩, ⟨π2, E2⟩)
    E' := E1 ⊗ E2
    π := compute-potential(E', π1)
    if π = inconsistent
        return ⊥
    δ := close-meet(π, E', E1, E2)
    return ⟨π, E' ⊗ δ⟩
```

Fig. 5. Meet on sparse DBMs

The classic approach to computing valid potential functions is the Bellman-Ford [12] algorithm. Many refinements and variants have been described [4], any of which we could apply. We use the basic algorithm, with three refinements:

- π' is initialized from π_1 or π_2.
- Bellman-Ford is run separately on each strongly-connected component.
- We maintain separate queues for the current and next iteration.

Fig. 6 shows the modified algorithm. Note that if $\pi'(x)$ changes but x is still in Q, we do not need to add it to Q'—its successors will already have been updated at the end of the current iteration.

The direct approach to restoring closure of E' is to run Dijkstra's algorithm from each vertex (essentially running Johnson's algorithm). However, we can exploit the fact that E_1 and E_2 are already closed. When we collect the pointwise minimum $E_1 \otimes E_2$, we mark each edge as 1, 2 or both, according to its origin. Observe that if all edges reachable from some vertex $\mathtt{v}$ have the same mark, the subgraph from $\mathtt{v}$ is already closed.

Critically, consider the behaviour of Dijkstra's algorithm. We expand some vertex v, adding $v \xrightarrow{k} x$ to the queue. Assume the edge $v \xrightarrow{k} x$ originated from

```
compute-potential(E, π)
    π′ := π
    for (scc ∈ components(E))
        Q := scc
        for (iter ∈ [1, |scc|])
            Q′ := ∅
            while (Q ≠ ∅)
                x := Q.pop()
                for (x --k--> y ∈ E(x))
                    if π′(x) + k − π′(y) < 0
                        π′(y) := π′(x) + k
                        if (y ∈ scc ∧ y ∉ Q ∪ Q′)
                            Q′ := Q′ ∪ {y}
            if Q′ = ∅
                return π′
            Q := Q′
        while (Q ≠ ∅)
            x := Q.pop()
            for (x --k--> y ∈ E(x))
                if π′(x) + k − π′(y) < 0
                    return inconsistent
    return π′
```

Fig. 6. Warm-started Bellman-Ford; SCCs assumed to be ordered topologically.

the set E_1. At some point, we remove $v \xrightarrow{k} x$ from the queue. Let $x \xrightarrow{k'} y$ be some child of x. If $x \xrightarrow{k'} y$ *also* originated from E_1, we know that E_1 also contained some edge $v \xrightarrow{c} y$ with $c \leq k + k'$ which will already be in the queue; thus there is no point exploring any outgoing E_1-edges from x.

We thus derive a specialized variant of Dijkstra's algorithm. The following assumes we can freely iterate through edges of specific colours—this index can be maintained during construction, or partitioning edges via bucket-sort between construction and closure.[4] This "chromatic" variant is given in Fig. 7. We run Dijkstra's algorithm as usual, except any time we find a minimum-length path to some node y, we mark y with the colour of the edge through which it was reached. Then, when we remove y from the priority queue we only explore edges where none of its colours are already on the vertex. Note that nodes in Q are ordered by slack $(\pi(x) + \mathit{Dist}(x) - \pi(y))$, rather than raw distance $\mathit{Dist}(x)$. The initialization of *Dist* and *edge-col* is performed only once and preserved between calls, rather than performed explicitly for each call.

Example 5. Consider the conjunction of closed states in Fig. 8(a). Running the closure-aware Dijkstra's algorithm from each vertex restores closure. Now taking

[4] It is not immediately clear how to extend this efficiently to an n-way meet, as a vertex may be reachable from some arbitrary subset of the operands.

```
chromatic-Dijkstra(⟨π, E⟩, x)
    Dist(v) := ∞ for all v
    δ := ∅
    for each x -k-> y ∈ E(x)
        Dist(y) := k
        Q.add(y)
        reach-col(y) := edge-col(x, y)
    while (Q ≠ ∅)
        y := Q.remove-min()
        if Dist(y) < wt_E(x, y))
            δ := δ ∪ { x -Dist(y)-> y }
        % Iterate through edges of the other colour
        for (c ∈ {1, 2} \ reach-col(y))
            for each y -k-> z in E_c(y)
                d_xyz := Dist(y) + k
                if d_xyz = Dist(z)
                    reach-col(z) := reach-col(z) ∪ edge-col(y, z)
                if d_xyz < Dist(z)
                    Dist(z) := d_xyz
                    Q.update(z)
                    reach-col(z) := edge-col(y, z)
    return δ

close-meet(π, E, E_1, E_2)
    edge-col(x, y) := ∅  for all x, y
    for each i ∈ {1, 2}, x -k-> y ∈ E_i
        if wt_E(x, y) = k
            edge-col(x, y) := edge-col(x, y) ∪ {i}
    δ := ∅
    for each vertex x
        δ := δ ∪ chromatic-Dijkstra(⟨π, E⟩, x)
    return δ
```

Fig. 7. Pseudo-code for Dijkstra's algorithm, modified to exploit closed operands.

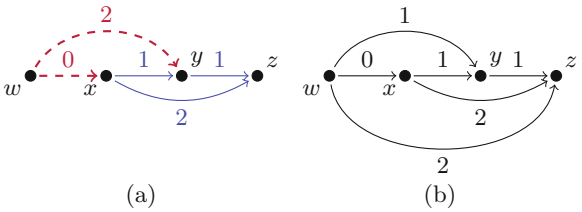


Fig. 8. (a) The conjunction of two closed graphs, $E_1 = \{x - w \leq 0, y - w \leq 2\}$ and $E_2 = \{y - x \leq 1, z - y \leq 1, z - x \leq 2\}$, and (b) the closure $(E_1 \otimes E_2)^*$. (Color figure online)

x as the source, we add $x \xrightarrow{1} y$ and $x \xrightarrow{2} z$ to the priority queue, and mark y and z as reachable via blue (solid) edges. We then pop y from the queue. Variable y is marked as reachable via blue so we need only check purple (dashed) children, of which there are none. We finally pop z, finding the same.

Selecting w as source, we add $w \xrightarrow{0} x$ and $w \xrightarrow{2} y$ to the queue, both marked as reachable via purple (dashed) edges. We then process x. It is reachable via purple, so we must expand its blue children. The edges $x \xrightarrow{1} y$ and $x \xrightarrow{2} z$ respectively give an improved path to y and a path to z, so we update the distances and mark both y and z as reachable instead by blue. This places us in the same state we had before; we finish processing y and z as above. The resulting graph is shown in Fig. 8(b). □

3.6 Widening

For widening we follow the usual practice of discarding *unstable* edges—any edges that were weakened in two successive iterates E_1 and E_2. (Note that for each edge $x \xrightarrow{k_2} y \in E_2$ there is an edge $x \xrightarrow{k_1} y \in E_1$ with $k_1 \leq k_2$.)

To obtain the result $E = E_1 \triangledown E_2$, widening starts from an empty set E. It then considers each edge $x \xrightarrow{k_2} y \in E_2$ in turn and adds $x \xrightarrow{k_1} y$ to E iff $x \xrightarrow{k_1} y \in E_1$ *and* $k_2 \leq k_1$. In the pseudo-code of Fig. 9, this is performed by remove-unstable-edges(E_1, E_2), which also returns a set `is-stable` containing those vertices which have lost no outgoing edges.

Figure 9 presents the complete pseudo-code for widening. The algorithm is, however, more complex than just removal of edges. The reason is that, unlike the join operation, the removal of edges may fail to preserve closure. Hence closure must be restored before subsequent operations, but at the same time, subsequent widenings must use the *un-closed* result. For this reason, widen produces both. The function close-after-widen is responsible for the restoration of closure. It uses a short-cut similar to the chromatic Dijkstra algorithm. Recall that $E(x)$ denotes the set $E' \subseteq E$ of edges that emanate from x. Consider again running Dijkstra's algorithm from v on $A \triangledown B$. At some point, we reach a vertex w with $A^*(w) = (A \triangledown B)(w)$—that is, all outgoing edges from w are stable in A^*. But since $A \sqsubseteq (A \triangledown B)$, we have $(A \triangledown B)(w) = A^*(w) \sqsubseteq (A \triangledown B)^*(w)$. Thus we do not need to further expand any children reached via w, as any such path is already in the queue. As such, we augment the left operand of the widening (the previous iterate) with a set S_E, threaded through successive widening steps, to indicate which vertices remain stable under closure. Details are provided in Fig. 9.

```
Dijkstra-restore-unstable(⟨π, E⟩, x, is-stable)
    δ := ∅, Dist(v) := ∞ for all v
    for each x -k-> y ∈ E(x)
        Dist(y) := k
        Q.add(y)
        closed(y) := false
    while Q ≠ ∅
        y := Q.remove-min()
        if Dist(y) < wt_E(x, y)
            δ := δ ∪ {x -Dist(y)-> y }
        if not closed(y)
            % Iterate through unstable successors
            for each y -k-> z in E(y)
                d_xyz := Dist(y) + k
                if d_xyz = Dist(z)
                    closed(z) := closed(z) ∨ is-stable(y)
                if d_xyz < Dist(z)
                    Dist(z) := d_xyz
                    Q.update(z)
                    closed(z) := is-stable(y)
    return δ

close-after-widen(⟨π, E⟩, is-stable)
    δ := ∅, S_E' := is-stable
    for each vertex x
        if not is-stable(x)
            δ_x := Dijkstra-restore-unstable(⟨π, E⟩, x, is-stable)
            if delta_x = ∅ then S_E' := S_E' ∪ {x}
            δ := δ ∪ δ_x
    return δ, S_E'

widen(⟨π1, E1⟩|S_E, ⟨π2, E2⟩)
    ⟨E', is-stable⟩ := remove-unstable-edges(E1, E2)
    un-closed := ⟨π2, E'⟩
    δ, S_E' := close-after-widen(un-closed, S_E ∩ is-stable)
    closed := ⟨π2, E' ⊗ δ⟩
    return ⟨un-closed|S_E', closed⟩
```

Fig. 9. Widening on sparse DBMs

4 Zones as Sparse DBMs in Split Normal Form

So far we made the key assumption that abstract states are sparse. During the later stages of analysis, this is typically the case. However, abstract states in early iterations are often very dense; Singh, Püschel and Vechev [20] observed (in the context of Octagon analysis) that the first 50 % of iterations were extremely

dense, with sparsity only appearing after widening. However, at a closer look this density turns out to be a mirage.

Recall the discussion in Sect. 2 on the handling of variable bounds: an artificial vertex v_0 is introduced, and bounds on x are encoded as relations between x and v_0. Unfortunately, this interacts badly with closure - if variables are given initial bounds, our abstract state becomes complete.

```
0 : x_1, ..., x_k := 1, ..., k
1 : if(*)
2 :     x_1 := x_1 + 1
3 :     x_2 := x_2 + 1
4 :
```

Fig. 10. Small code fragment

This is regrettable, as it erodes the sparsity we want to exploit. It is only after widening that unstable variable bounds are discarded and sparsity arises, revealing the underlying structure of relations. Also, all these invariants are trivial—we only really care about relations *not* already implied by variable bounds.

Example 6. Consider the program fragment in Fig. 10. Variables $x_1, \ldots, x_k$ are initialised to constants at point 0. A direct implementation of DBM or Octagons will compute all $k(k-1)$ pairwise relations implied by these bounds. During the execution of lines 2 and 3, all these relations are updated, despite all inferred relations being simply the consequences of variable bounds.

At point 4 we take the join of the two sets of relations. In a direct implementation, this graph is complete, even though there is only one relation that is not already implied by bounds, namely $x_2 = x_1 + 1$. □

One can avoid this phantom density by storing the abstract state in a (possibly weakly) *transitively reduced* form, an approach that has been used to improve performance in SMT and constraint programming [5,11], and to reduce space consumption in model checking [14].[5] But we are hindered by the need to perform *join* operations. The join of two closed graphs is simply $E_1 \oplus E_2$. For transitively reduced graphs, we are forced to first *compute the closure*, perform the pointwise maximum, then restore the result to transitively reduced form. Algorithms exist to efficiently compute the transitive reduction and closure together, but we would still need to restore the reduction after joins.

Instead, we construct a modified normal form which distinguishes independent properties (edges to/from v_0) from strictly relational properties (edges

[5] This terminology may be confusing. The transitive reduction computes the greatest (by $\sqsubseteq$) equivalent representation of R, whereas the usual abstract-domain *reduction* corresponds to the transitive *closure*.

between program variable). An important property of this normal form is that *it preserves strongest invariants involving* v_0.

A graph $G = \langle V, E \rangle$ is in *split normal form* iff:

- $G \setminus \{v_0\}$ is closed, and
- for each edge $v_0 \xrightarrow{k} x$ (or $x \xrightarrow{k} v_0$) in G, the shortest path in G from v_0 to x (resp. x to v_0) has length k.

If G is in split normal form then any shortest path from x to y in G occurs either as an edge $x \xrightarrow{k} y$, or the path $x \xrightarrow{k_1} v_0, v_0 \xrightarrow{k_2} y$. We have $E_1 \sqsubseteq E_2$ iff, for every $x \xrightarrow{k} y \in E_2$, $\mathbf{min}(wt_{E_1}(x, y), wt_{E_1}(x, v_0, y)) \leq k$. Assuming constant-time lookups, this test takes $\mathrm{O}(|E_2|)$ time.

Note that split normal form is *not* canonical: the graphs $\{x \xrightarrow{1} v_0, v_0 \xrightarrow{1} y\}$ and $\{x \xrightarrow{1} v_0, v_0 \xrightarrow{1} y, x \xrightarrow{2} y\}$ are both in split normal form, and denote the same set of relations. We could establish a canonical form by removing edges implied by variable bounds, then re-closing $G \setminus \{v_0\}$. But we gain nothing by doing so, as we already have an efficient entailment test.

An abstract state in the domain of *split DBMs* consists of a pair $\langle \pi, G \rangle$ of a graph G in split normal form, and a potential function π for G. We must now modify each abstract operation to deal with graphs in split normal form.

4.1 Abstract Operations for Split Normal Form

Variable elimination is unchanged—we simply discard edges touching x.

The modifications for variable assignment, constraint addition and meet are mostly straightforward. The construction of the initial (non-normalized) result and computation of potential function are performed exactly as in Sect. 3.

We then restore closure as before, but only over $G \setminus \{v_0\}$, yielding the set δ of changed edges. We then finish by walking over δ to restore variable bounds.

Widening is also straightforward. The construction of the *un-closed* component by removing *unstable* edges is done as in Sect. 3.6. For the closed result used in subsequent iterations, we restore closure as before but only over $G \setminus \{v_0\}$ and compute separately the closure for v_0.

Pseudo-code for constraint addition, assignment, meet, and widening are given in Fig. 11.

Computation of $E_1 \sqcup E_2$ for split normal graphs is more intricate. As before, either potential may be retained, and edges $v_0 \xrightarrow{k} x$ and $x \xrightarrow{k} v_0$ need no special handling. But direct application of the join used in Sect. 3 may lose precision.

Example 7. Consider the join at point 4 in Fig. 10. In split normal form, the abstract states are:

$$E_1 = \{x_1 \xrightarrow{-1} v_0, v_0 \xrightarrow{1} x_1, x_2 \xrightarrow{-2} v_0, v_0 \xrightarrow{2} x_2, \ldots\}$$
$$E_2 = \{x_1 \xrightarrow{-2} v_0, v_0 \xrightarrow{2} x_1, x_2 \xrightarrow{-3} v_0, v_0 \xrightarrow{3} x_2, \ldots\}$$

$\mathsf{update\text{-}bounds_F}(E, \delta)$
  $lb := \{v_0 \xrightarrow{k_0+k_e} d \mid v_0 \xrightarrow{k_0} s \in E \wedge s \xrightarrow{k_e} d \in \delta \wedge k_0 + k_e < wt_E(v_0, d)\}$
  $ub := \{s \xrightarrow{k_0+k_e} v_0 \mid s \xrightarrow{k_e} d \in \delta \wedge d \xrightarrow{k_0} v_0 \in E \wedge k_0 + k_e \leq wt_E(s, v_0)\}$
  return $\delta \cup lb \cup ub$

$\mathsf{add\text{-}edge_F}(\langle \pi, E \rangle, e)$
  $E' := E \cup \{e\}$
  $\pi' := \mathit{restore\text{-}potential}(\pi, e, E')$
  if $\pi' =$ `inconsistent`
    return $\bot$
  $\delta := \mathsf{close\text{-}edge}(e, E' \setminus \{v_0\})$
  return $\langle \pi', E' \otimes \mathsf{update\text{-}bounds_F}(E, \delta) \rangle$

$\mathsf{assign_F}(\langle \pi, E \rangle, [\![\mathtt{x} := \mathtt{S}]\!])$
  $\pi' := \pi[x \mapsto \mathsf{eval\text{-}potential}(\pi, \mathtt{S})]$
  $E' := E \cup \mathsf{edges\text{-}of\text{-}assign}(E, [\![\mathtt{x} := \mathtt{S}]\!])$
  $\delta := \mathsf{close\text{-}assign}(\langle \pi', E' \setminus \{v_0\} \rangle, x)$
  return $\langle \pi', E' \otimes \delta \rangle$

$\mathsf{meet_F}(\langle \pi_1, E_1 \rangle, \langle \pi_2, E_2 \rangle)$
  $E' := E_1 \otimes E_2$
  $\pi' := \mathsf{compute\text{-}potential}(E', \pi_1)$
  if $\pi' =$ `inconsistent`
    return $\bot$
  $\delta := \mathsf{close\text{-}meet}(\pi', E' \setminus \{v_0\}, E_1, E_2)$
  return $\langle \pi, E' \otimes \mathsf{update\text{-}bounds_F}(E', \delta) \rangle$

$\mathsf{widen_F}(\langle \pi_1, E_1 \rangle_{|S_E}, \langle \pi_2, E_2 \rangle)$
  $\langle E', \mathit{is\text{-}stable} \rangle := \mathsf{remove\text{-}unstable\text{-}edges}(E_1, E_2)$
  $\mathit{un\text{-}closed} := \langle \pi_2, E' \rangle$
  $\delta, S_{E'} := \mathsf{close\text{-}after\text{-}widen}(\langle \pi_2, E' \setminus \{v_0\} \rangle, \mathit{is\text{-}stable})$
  $\delta := \delta \cup \mathsf{close\text{-}assignment}(\mathit{un\text{-}closed}, v_0)$
  $\mathit{closed} := \langle \pi_2, E' \otimes \delta \rangle$
  return $\langle \mathit{un\text{-}closed}_{|S_{E'}}, \mathit{closed} \rangle$

Fig. 11. Modified algorithms for split normal graphs

In each case the relation $x_2 - x_1 = 1$ is implied by the paths $x_1 \xrightarrow{-1} v_0, v_0 \xrightarrow{2} x_2$ and $x_2 \xrightarrow{-2} v_0, v_0 \xrightarrow{1} x_1$. If we apply the join from Sect. 3, we obtain:

$$E' = \{x_1 \xrightarrow{-1} v_0, v_0 \xrightarrow{2} x_1, x_2 \xrightarrow{-2} v_0, v_0 \xrightarrow{3} x_2, \ldots\}$$

This only supports the weaker relation $0 \leq x_2 - x_1 \leq 2$. □

We could find the missing relations by computing the closures of E_1 and E_2; but this rather undermines our objective. Instead, consider the ways a relation might arise in $E_1 \sqcup E_2$:

1. $x \xrightarrow{k} y \in E_1, x \xrightarrow{k'} y \in E_2$
2. $x \xrightarrow{k} y \in E_1, \{x \xrightarrow{k'} v_0, v_0 \xrightarrow{k''} y\} \subseteq E_2$ (or the converse)
3. $\{x \xrightarrow{k_1} v_0, v_0 \xrightarrow{k_2} y\} \subseteq E_1, \{x \xrightarrow{k'_1} v_0, v_0 \xrightarrow{k'_2} y\} \subseteq E_2$, where

$$\mathbf{max}(k_1 + k_2, k'_1 + k'_2) < \mathbf{max}(k_1, k'_1) + \mathbf{max}(k_2, k'_2)$$

The join of Sect. 3 will collect only those relations which are explicit in both E_1 and E_2 (case 1). We can find relations of the second form by walking through E_1 and collecting any edges which are implicit in E_2. The final case is that illustrated in Example 7, where some invariant is implicit in both operands, but is no longer maintained in the result. The restriction on case 3 can only hold when $wt_{E_1}(x, v_0) < wt_{E_2}(x, v_0)$ and $wt_{E_2}(v_0, y) < wt_{E_1}(v_0, y)$ (or the converse).

We can collect suitable pairs by collecting the variables into buckets according to $sign(wt_{E_1}(v_0, x) - wt_{E_2}(v_0, x))$ and $sign(wt_{E_1}(x, v_0) - wt_{E_2}(x, v_0))$. We then walk through the compatible buckets and instantiate the resulting relations.

split-rels(E_I, E_R)
  $E_{IR} := \emptyset$
  for $(x \xrightarrow{k} y \in (E_R - v_0))$
    if$(wt_{E_I}(x, v_0, y) < wt_{E_I}(x, y))$
      $E_{IR} := E_{IR} \cup \{x \xrightarrow{wt_{E_I}(x,v_0,y)} y\}$
  return E_{IR}

bound-rels(src, $dest$)
  $E_{II} := \emptyset$
  for $((x, k_1, k'_1) \in src,\ (y, k_2, k'_2) \in dest, x \neq y)$
    $k_{xy} := \mathbf{max}(k_1 + k_2, k'_1 + k'_2)$
    $E_{II} := E_{II} \cup \{x \xrightarrow{k_{xy}} y\}$
  return E_{II}

split-join($\langle \pi_1, E_1 \rangle, \langle \pi_2, E_2 \rangle$)
  $\pi' := \pi_1$
  $E_{I_1} :=$ split-rels(E_1, E_2)
  $E_{I_2} :=$ split-rels(E_2, E_1)
  $E_{1+} :=$ close-meet$(\pi_1, E_{I_1} \otimes E_1, E_{I_1}, E_1)$
  $E_{2+} :=$ close-meet$(\pi_2, E_{I_2} \otimes E_2, E_{I_2}, E_2)$
  for $(s \in \{+, -\})$
    $src_s := \{(x, wt_{E_1}(x, v_0), wt_{E_2}(x, v_0)) \mid sign(wt_{E_1}(x, v_0) - wt_{E_2}(x, v_0)) = s\}$
    $dest_s := \{(y, wt_{E_1}(v_0, y), wt_{E_2}(v_0, y)) \mid sign(wt_{E_1}(v_0, y) - wt_{E_2}(v_0, y)) = s\}$
  $E_{I_{12}} :=$ bound-rels$(src_+, dest_-) \cup$ bound-rels$(src_-, dest_+)$
  return $\langle \pi', E_{I_{12}} \otimes (E_{1+} \oplus E_{2+}) \rangle$

Fig. 12. Pseudo-code for join of abstract states in split normal form. split-rels collects edges which are implicit in E_I but explicit in E_R. bound-rels collects relations implied by compatible bound changes.

This yields the join algorithm given in Fig. 12. Note the order of construction for E'. Each of the initial components E_1, E_{I_1}, E_2, E_{I_2} and $E_{I_{12}}$ are split-normal. Augmenting E_1 with implied properties E_{I_1} allows us to use the chromatic Dijkstra algorithm for normalization. There is no need to normalize when computing $E_{1+} \oplus E_{2+}$, as relations in classes (1) and (2) will be preserved, relations with v_0 are fully closed, and relations of class (3) will be imposed later.

Due to the construction of $E_{I_{12}}$, normalization of the result again comes for free. Assume $E_{I_{12}} \otimes (E_{1+} \oplus E_{2+})$ is *not* closed. Then there must be some path $x \xrightarrow{k} y \in E_{I_{12}}$, $y \xrightarrow{k'} z \in (E_{1+} \oplus E_{2+})$ such that $x \xrightarrow{k+k'} z$ is not in either operand. But that means there must be some path $x \xrightarrow{c_1} v_0, v_0 \xrightarrow{c_2} y, y \xrightarrow{c_3} z \in E_1$ such that $c_1 + c_2 \leq k$, $c_3 \leq k'$, so there must also be a path $x \xrightarrow{c_1} v_0, v_0 \xrightarrow{c'} z \in E_1$, with $c' \leq c_2 + c_3$. The same holds for E_2. Thus $x \xrightarrow{k+k'} z$ must be in $E_{I_{12}} \otimes (E_{1+} \oplus E_{2+})$.

5 Sparse Graph Representations

So far we avoided discussing the underlying graph representation. The choice is critical for performance. For $\sqcap$ or $\sqcup$, we must walk pointwise across the two graphs; during closure, it is useful to iterate over edges incident to a vertex, and to examine and update relations between arbitrary pairs of variables. On variable elimination, we must remove all edges to or from v. Conventional representations support only some of these operations efficiently. Dense matrices are convenient for updating specific entries but cannot iterate over only the non-trivial entries. $\sqcap$ and $\sqcup$ must walk across the entire matrix—even copying an abstract state is always a $O(|V|^2)$ operation. Adjacency lists support efficient iteration and handle sparsity gracefully, but we lose efficiency of insertion and lookup.

A representation which supports all the required operations is the *adjacency hash-table*, consisting of a hash-table mapping successors to weights for each vertex, and a hash-set of the predecessors of each vertex. This offers the asymptotic behaviour we want but is rather heavy-weight, with substantial overheads on operations. Instead we adopt a hybrid representation; weights are stored in a dense *but uninitialized* matrix, and adjacencies are stored using a "sparse-set" structure [2]. This improves the efficiency of primitive operations, for a reasonable space cost. It introduces an overhead of roughly 8 bytes per matrix element[6]—two bytes each for the *sparse* and *dense* entry for both predecessors and successors. For 64-bit weights, this doubles the overall memory requirements relative to the direct dense matrix.

A sparse-set structure consists of a triple $(dense, sparse, sz)$ where *dense* is an array containing the elements currently in the set, *sparse* is an array mapping elements to the corresponding indices in *dense*, and sz the number of elements in the set. We can iterate through the set using $\{dense[0], \ldots, dense[sz-1]\}$.

Fig. 13 shows the sparse-set operations. We preserve the invariant $\forall i \in [0, sz)$. $sparse[dense[i]] = i$. This means for any element k' outside the set,

[6] This assumes 16-bit vertex identifiers; if more than 2^{16} variables are in scope at a program point, any dense-matrix approach is already impractical.

```
elem((dense, sparse, sz), k)
    return sparse[k] < sz ∧ dense[sparse[k]] = k

add((dense, sparse, sz), k)
    sparse[k] := sz
    dense[sz] := k
    sz := sz + 1

remove((dense, sparse, sz), k)
    sz := sz − 1
    k′ := dense[sz]
    dense[sparse[k]] := k′
    sparse[k′] := sparse[k]
```

Fig. 13. Sparse-set operations

either $sz \leq sparse[i]$, or $dense[sparse[k']]$ points to some element other than k'—without making any assumptions about the values in *sparse* or *dense*. So we only need to allocate memory for *sparse* and *dense*, and initialize sz.

The result is a representation with O(1) addition, removal, lookup and enumeration (with low constant factors) and $O(|V| + |E|)$ time to initialize/copy (we can reduce this to $O(|E|)$ by including an index of non-empty rows, but this adds an additional cost to each lookup).

While the sparse set incurs only a modest overhead (vs a dense matrix), this is still wasteful for extremely sparse graphs. So we choose an adaptive representation. Adjacency lists are initially allocated as unsorted vectors. When the length of a list exceeds a small constant (viz. 8) we allocate the *sparse* array, turning the list into a sparse set. This yields equivalent performance (both asymptotically and in practice), with considerably smaller memory footprint for very sparse graphs.

6 Experimental Results

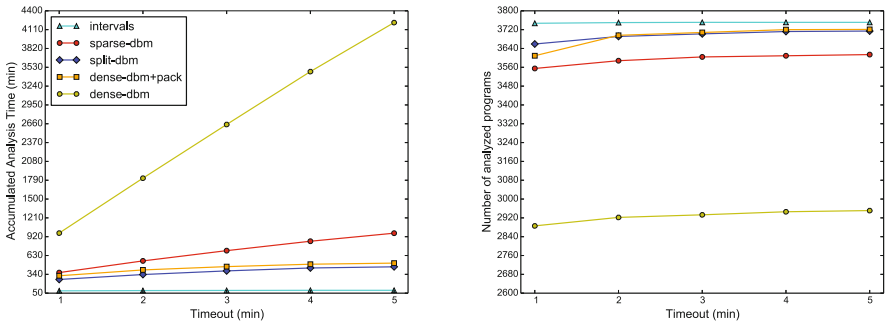
We have implemented the Zones abstract domain using both sparse DBMs and sparse DBMs in Split Normal Form. We now compare their performance, and we evaluate the major algorithmic choices discussed in Sects. 3, 4 and 5. All the DBM-based alternatives have been implemented and integrated in CRAB, a language-agnostic static analyzer based on abstract interpretation[7]. For the experiments we use 3753 programs from SV-COMP 2016 [1]. We focus on seven program categories that challenge numerical reasoning: Simple, ControlFlowInteger, Loops, Sequentialized, DeviceDrivers64, ECA, and ProductLines. Since

[7] Code is available from the authors upon request.

all programs are written in C they have first been translated[8] to LLVM [15] bitcode and then to a simple language consisting of assume, assignments, arithmetic operations, and goto statements understood by CRAB using the CRAB-LLVM tool. The ultimate goal of the analyzer is to infer inductive invariants at each basic block. All the experiments were carried out on a 2.1 GHz AMD Opteron processor 6172 with 32 cores and 64 GB on a Linux machine.

We first compare DBM-based implementations of Zones: dense-dbm [17] uses a dense matrix with incremental Floyd-Warshall ([3], Fig. 7, INCREMENTAL-DIFFERENCEV2) for both constraint addition and assignment; sparse-dbm uses sparse DBMs (Sect. 3); and split-dbm uses sparse DBMs but splits independent and relational invariants as described in Sect. 4. As performance baseline, we also compare with intervals, a classical implementation of intervals. Finally we add dense-dbm+pack which enhances dense-dbm with dynamic variable packing [21]. Note that dense-dbm, sparse-dbm, and split-dbm have the same expressiveness, so they infer the same invariants. The performance results of dense-dbm+pack are not really comparable with the other DBM implementations: the analysis is less precise since it only preserves relationships between variables in the same pack. For our benchmark suite, intervals inferred the same invariants as dense-dbm+pack in 19 % of the cases, whereas dense-dbm+pack infers the same as split-dbm in only 29 % of the cases.

Figure 14 shows the results, using an 8 GB memory limit and various timeouts ranging from 1 to 5 min. Five implementations are compared. Figure 14(a) shows, for each, the accumulated time in minutes. Figure 14(b) shows how many programs were analyzed without exceeding the timeout. If the analyzer exhausted the resources then the timeout value has been counted as analysis time. The comparison shows that sparse-dbm is significantly faster than dense-dbm ($4.3x$)



(a) (b)

Fig. 14. Performance of several DBM-based implementations of *Zones* over 3753 SV-COMP'16 benchmarks with 8GB memory limit.

[8] We tried to stress test the DBM implementations by increasing the number of variables in scope through inlining. We inlined all function calls unless a called function was recursive or could not be resolved at compile time.

while being able to analyze many more programs (> 500). Moreover, split-dbm outperforms sparse-dbm both in the number of analyzed programs (100 more) *and* analysis times ($2x$). Note that compared to the *far less precise* dense-dbm +pack, split-dbm can analyze almost the same number of programs while at the same time being faster ($1.1x$).

Table 1. Performance of different versions of split-dbm, applied to SV-COMP'16 programs, with a timeout of 60 s and a memory limit of 8 GB.

	Graph Representation				Assign		Meet	
	Total: 3753				Total: 3753		Total: 4079	
	hash	trie	ss	adapt-ss	close-edge	close-assign	johnson	chromatic
Analyzed	3637	3545	3632	3662	3659	3662	4020	4028
TOs	116	208	68	91	91	91	91	48
MOs	0	0	53	0	3	0	0	3
Time	318	479	252	265	269	265	160	151

In another three experiments we have evaluated the major algorithmic and representation choices for split-dbm. Table 1 shows the results. Analyzed is the number of programs for which the analyzer converged without exhausting resources. TOs is the number of time-outs, MOs is the number of cases where memory limits were exceeded, and Time is the accumulated analysis time in minutes. Again, MOs instances were counted as time-outs for the computation of overall time. (We do not repeat these experiments with sparse-dbm since it differs from split-dbm only in how independent properties are treated separately from relational ones—hence similar results should be expected for sparse-dbm.)

1. **Representation impact:** hash uses hash tables and sets as described in Sect. 5; trie is similar but uses Patricia trees [19] instead of hash tables and sets; (adapt-) ss is the (adaptive) hybrid representation using dense matrices for weights and sparse sets, also described in Sect. 5.
2. **Algorithms for abstract assign** `x:=e`**:** close-edge implements the assignment as a sequence of edge additions (Fig. 2). close-assign implements the algorithm described in Fig. 4. We evaluated both options using the adapt-ss graph representation.
3. **Algorithms for meet:** johnson uses Johnson's algorithm for closure while chromatic uses our Chromatic Dijkstra (Fig. 7). We evaluated the two after committing to the best previous options, that is, adapt-ss and close-assign. NB: To increase the number of meets we did not inline programs here. This is the reason why the total number of programs has increased—LLVM can compile more programs in reasonable time when inlining is disabled.

These last three experiments justify the choices (adapt-ss, close-assign, and chromatic) used in the implementation whose performance is reported in Fig. 14.

7 Related Work

Much research has been motivated by the appeal, but ultimately lacking scalability, of program analyses based on relational domains. Attempts to improve the state of the art fall in three rough classes.

Approaches based on *dimensionality restriction* attempt to decrease the dimension k of the program abstract state by replacing the full space with a set of subspaces of lower-dimensional sub-spaces. Variables are separated into "buckets" or *packs* according to some criterion. *Variable packing* was first utilised in the Astree analyser [8]. A dynamic variant is used in the C Global Surveyor [21].

Others choose to sacrifice precision, establishing a new trade-off between performance and expressiveness. Some abandon the systematic transitive closure of relations (and work around the resulting lack a normal form for constraints). Constraints implied by closure may be discovered lazily, or not at all. Used with their Pentagon domain, Logozzo and Fähndrich [16] found that this approach established a happy balance between cost and precision. The Gauge domain proposed by Venet [22] can be seen as a combination of this approach and dimensionality restriction. In the Gauge domain, relations are only maintained between program variables and specially introduced "loop counter" variables.

Finally, one can focus on algorithmic aspects of classical abstract domains and attempt to improve performance without modifying the domain (as in this paper). Here the closest related work targets Octagons rather than Zones. Chawdhary, Robbins and King [3] present algorithmic improvements for the Octagon domain, assuming a standard matrix-based implementation (built on DBMs). They focus on the common use case where a single constraint is added/changed, that is, their goal is an improved algorithm for *incremental* closure.

8 Conclusion and Future Work

We have developed new algorithms for the Zones abstract domain and described a graph representation which is tailored for efficient implementation across the set of needed abstract operations. We have introduced *split normal form*, a new graph representation that permits separate handling of independent and relational properties. We have provided detailed descriptions of how to implement the necessary abstract operations for this representation. Performance results show that, despite having more involved operations (particularly the join operation), the resulting reduction in density is clearly worth the effort.

Evaluation on SV-COMP'16 instances shows that sparse-dbm and split-dbm are efficient, performing *full relational analysis* while being no more expensive than less expressive variable packing methods.

Standard implementations of Octagons are also based on DBMs [3,18]. A natural application of the algorithmic insights of the present paper is to adapt our ideas to Octagon implementation. This should be particularly interesting

in the light of recent implementation progress based on entirely different ideas. For their improved Octagon implementation, Singh, Püschel and Vechev [20] commit to a (costly) dense matrix representation in order to enable abstract operations that are cleverly based on vectorization technology. (They also check for disconnected subgraphs, and decompose the matrix into smaller, packing-style, sub-components when this occurs.) In contrast we see Zone/Octagon types of program analysis as essentially-sparse graph problems. In Sect. 4 we have argued that the density observed elsewhere (including [20]) is artificial—it stems from a failure to separate independent properties from truly relational properties. Our approach is therefore almost diametrically opposite that of [20] as we choose to exploit the innate sparsity as best we can. A comparison should be interesting.

Acknowledgments. We acknowledge the support from the Australian Research Council through grant DP140102194. We would like to thank Maxime Arthaud for implementing the non-incremental version of dense difference-bound matrices as well as the variable packing technique.

References

1. Competition on software verification (SV-COMP) (2016). http://sv-comp.sosy-lab.org/2016/. Benchmarks https://github.com/sosy-lab/sv-benchmarks/c. Accessed 30 Mar 2016
2. Briggs, P., Torczon, L.: An efficient representation for sparse sets. ACM Lett. Program. Lang. Syst. **2**(1–4), 59–69 (1993)
3. Chawdhary, A., Robbins, E., King, A.: Simple and efficient algorithms for octagons. In: Garrigue, J. (ed.) APLAS 2014. LNCS, vol. 8858, pp. 296–313. Springer, Heidelberg (2014)
4. Cherkassky, B.V., Goldberg, A.V.: Negative-cycle detection algorithms. Math. Program. **85**(2), 277–311 (1999)
5. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the Fourth ACM Symposium Principles of Programming Languages, pp. 238–252. ACM Press (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the Sixth ACM Symposium Principles of Programming Languages, pp. 269–282. ACM Press (1979)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does Astrée scale up? Formal Methods Syst. Des. **35**(3), 229–264 (2009)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: Proceedings of the Fifth ACM Symposium Principles of Programming Languages, pp. 84–97. ACM Press (1978)
10. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011)

11. Feydy, T., Schutt, A., Stuckey, P.J.: Global difference constraint propagation for finite domain solvers. In: Proceedings of the 10th International ACM SIGPLAN Conference Principles and Practice of Declarative Programming, pp. 226–235. ACM Press (2008)
12. Ford, L.R., Fulkerson, D.R.: Flows in Networks. Princeton University Press, Princeton (1962)
13. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. J. ACM **24**(1), 1–13 (1977)
14. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: compact data structure and state-space reduction. In: Proceedings of the 18th International Symposium Real-Time Systems, pp. 14–24. IEEE Computer Society (1997)
15. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis and transformation. In: Proceedings of the International Symposium Code Generation and Optimization (CGO 2004), pp. 75–86. IEEE Computer Society (2004)
16. Logozzo, F., Fähndrich, M.: Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In: Proceedings of the 2008 ACM Symposium Applied Computing, pp. 184–188. ACM Press (2008)
17. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
18. Miné, A.: The octagon abstract domain. High. Ord. Symbolic Comput. **19**(1), 31–100 (2006)
19. Okasaki, C., Gill, A.: Fast mergeable integer maps. In: Notes of the ACM SIGPLAN Workshop on ML, pp. 77–86, September 1998
20. Singh, G., Püschel, M., Vechev, M.: Making numerical program analysis fast. In: Proceedings of the 36th ACM SIGPLAN Conference Programming Language Design and Implementation, pp. 303–313. ACM (2015)
21. Venet, A., Brat, G.: Precise and efficient static array bound checking for large embedded C programs. In: Proceedings of the 25th ACM SIGPLAN Conference Programming Language Design and Implementation, pp. 231–242. ACM Press (2004)
22. Venet, A.J.: The gauge domain: scalable analysis of linear inequality invariants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 139–154. Springer, Heidelberg (2012)