A Fresh Look at Zones and Octagons

GRAEME GANGE, Monash University, Australia
ZEQUN MA, The University of Melbourne, Australia
JORGE A. NAVAS, SRI International, USA
PETER SCHACHTE, The University of Melbourne, Australia
HARALD SØNDERGAARD, The University of Melbourne, Australia
PETER J. STUCKEY, Monash University, Australia

Zones and Octagons are popular abstract domains for static program analysis. They enable the automated discovery of simple numerical relations that hold between pairs of program variables. Both domains are well understood mathematically but the detailed implementation of static analyses based on these domains poses many interesting algorithmic challenges. In this paper we study the two abstract domains, their implementation and use. Utilizing improved data structures and algorithms for the manipulation of graphs that represent difference-bound constraints, we present fast implementations of both abstract domains, built around a common infrastructure. We compare the performance of these implementations against alternative approaches offering the same precision. We quantify the differences in performance by measuring their speed and precision on standard benchmarks. We also assess, in the context of software verification, the extent to which the improved precision translates to better verification outcomes. Experiments demonstrate that our new implementations improve the state-of-the-art for both Zones and Octagons significantly.

CCS Concepts: • **Theory of computation → Program analysis**; *Program verification*; *Logic and verification*; *Abstraction*;

Additional Key Words and Phrases: Abstract interpretation, shortest path algorithms, difference-bounds matrix, numerical abstract domains, program analysis, static analysis, variable packing, weakly relational domains

ACM Reference Format:
Graeme Gange, Zequn Ma, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2021. A Fresh Look at Zones and Octagons. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (January 2021), 51 pages. https://doi.org/10.1145/3457885

1 INTRODUCTION

Static program analysis serves many important purposes, including compiler optimisation and program verification. Abstract interpretation allows program invariants over any number of different domains to be inferred, which may allow a compiler to generate more efficient executable code, or

Authors' addresses: Graeme Gange, Monash University, Faculty of IT, Clayton, 3800, Vic. Australia, graeme.gange@monash.edu; Zequn Ma, The University of Melbourne, School of Computing and Information Systems, Parkville, 3010, Vic. Australia, zequnm@student.unimelb.edu.au; Jorge A. Navas, SRI International, Computer Science Laboratory, Menlo Park, 94025-3493, CA, USA, jorge.navas@sri.com; Peter Schachte, The University of Melbourne, School of Computing and Information Systems, Parkville, 3010, Vic. Australia, schachte@unimelb.edu.au; Harald Søndergaard, The University of Melbourne, School of Computing and Information Systems, Parkville, 3010, Vic. Australia, harald@unimelb.edu.au; Peter J. Stuckey, Monash University, Faculty of IT, Clayton, 3800, Vic. Australia, peter.stuckey@monash.edu.

0164-0925/2021/1-ART1 $15.00
https://doi.org/10.1145/3457885

may allow a tool to assure the programmer that certain undesirable runtime conditions cannot arise.

For example, an analysis may determine upper or lower bounds on integer valued program variables—bounds that apply every time execution arrives at a given point in the program. Such an *interval analysis* is an example of an *independent attribute* analysis, so called because each program variable is ascribed a value independently of the rest. An alternative analysis might determine upper and lower bounds on the difference between program variables at each program point. This is a *relational* analysis, because it relates the possible values of variables to one another.

Generally, a relational analysis provides finer grained information about the possible runtime states, compared to an independent attribute analysis, but is typically much more expensive. For example, an analysis determining bounds on individual variables produce data proportional to the number of variables at each program point, while one that determines bounds on the difference between pairs of variables can produce data quadratic in the number of variables.

In principle, very fine grained relational abstract domains are attractive. In practice, however, the computational cost of analysis using domains such as the highly precise polyhedral domain [19] is prohibitive. Miné's development of the Zones and Octagons abstract domains [47–50] was driven by the desire to allow a static analysis tool to reason about relational information without incurring a substantial cost. Miné referred to these cheaper abstract domains as *weakly relational*. The Zones and Octagons domains limit expressiveness to strike a better compromise between precision and computational cost. They limit the possible coefficients that can be used in equations and they relate only pairs of variables, rather than arbitrary tuples. The resulting static analyses had a great impact on practice, as they were made publicly available in open-source libraries such as Apron [36] and PPL [4].

Several other abstract domains have been proposed that fit in the category of weakly relational domains [43, 62]. The theory of weakly relational abstract domains is well developed, and the corresponding analyses have been implemented and re-implemented several times. Nevertheless, these technologies have hardly reached maturity, and scalability remains a challenge. We should expect considerable scope for algorithmic advances, because a relational static analyzer is a complex tool with many interacting parts that call for balance and tuning. While the analyses usually utilise well-studied graph algorithms (for shortest-path problems), the application to static analysis poses its own unique challenges. Some of the necessary operations (lattice-theoretic "join" and "widening") have no natural counterparts in other applications of data structures and algorithms for shortest-path problems. These operations tend to complicate matters and call for application specific solutions. Moreover, typical static analysis workflows display commonly occurring patterns and a great deal of structure to be exploited. For example, we may be able to exploit knowledge of which analysis operations are more frequent, which normally precede which, and so on. We can also utilise what is known about typical analysis runs: That the generated relations are often quite sparse, that variables tend to settle into disjoint clusters, and that many operations in the analysis change only a small subset of the overall set of relations.

Indeed, attempts have been made to take advantage of such patterns and properties. To capitalise on the fact that most pairs of variables are unrelated at most program points (so that maintaining information about every pair is needlessly wasteful), it has been proposed to use *variable packing*, identifying groups of variables that *might* be related. Then independent abstractions can be kept for the separate *packs*. There are various approaches to determining the packs statically, before beginning the analysis (e.g., [8]), or dynamically, as the analysis progresses [59, 61]. Singh et al. [58] have combined dynamic partitioning with implementation techniques tailored to make use of vectorization. This has resulted in an implementation of an "optimized" Octagons domain in the ELINA library [23], OptOctagon, which is considerably faster than the classical Apron

```
int k = 200;
int n = 100;
int x = 0, y = k;

while (x < n) {
 x++;
 y = k + 2*x;
}
assert (y - x >= k);
```

(a)

```
int k = 200;
int n = 100;
int x = 0, y = k;

while (x < n) {
 x++;
 y = k - 2*x;
}
assert (x + y <= k);
```

(b)

Fig. 1. Code snippets to analyze.

implementation [36]. We return to these approaches in Sections 8 and 9 when we compare the state-of-the-art with our own implementation.

We have previously [28] pointed out that the standard implementations of Zones and Octagons fail to utilize the inherent sparsity of the relations involved. This happens because unnecessary density is introduced with the standard graph representations of difference constraints. In response, we developed algorithms based on a "split normal form" for these graphs [28]. Independently, Jourdan [39] proposed a collection of sparsity-preserving algorithms to be utilized in implementations of the Octagons domain.

An analysis using Zones allows for runtime state descriptions of the form $x - y \leq k$ (where x and y are program variables and k is a constant), as well as descriptions of form $x \leq k$ and $x \geq k$. (We can express *conjunctions* of these forms, so, for example, $x = k$ can also readily be expressed.) The Octagons domain extends Zones also to allow descriptions of the form $x + y \leq k$ and $-x - y \leq k$; we shall return to Octagons in Sections 6 and 7, but focus mainly on Zones for now.

For technical reasons it turns out to be advantageous to keep the constraints of form $x - y \leq k$ closed under "tight" entailment, This includes a closure principle to maintain "triangle inequalities", so that, for example, a set $\{x-y \leq k_1, y-z \leq k_2\}$ is extended to $\{x-y \leq k_1, y-z \leq k_2, x-z \leq k_1+k_2\}$. Moreover, a constraint set such as $\{x - y \leq k_1, x - y \leq k_2\}$ is *replaced* by $\{x - y \leq \mathbf{min}(k_1, k_2)\}$ (the tightness principle). Finally, traditional implementations extend a set such as $\{x \geq k_1, y \leq k_2\}$ to also include the entailed difference bound constraint $y - x \leq k_2 - k_1$.

As we shall see in Section 2.2, the constraint sets are conveniently represented as weighted directed graphs, with closure under tight entailment translating to shortest-path calculations, thus allowing the use of well-known graph algorithms. Since the graph that expresses how variables are related in a typical program tends to be sparse, we would prefer a representation that favours sparse graphs.

For an intuition of where unnecessary density comes from, consider the `C` code snippet in Figure 1(a). An analysis using Zones can successfully deduce invariants such as $1 \leq x \leq 100, 202 \leq y \leq 2x + 200$ (at the end of the body of Figure 1(a)'s while loop). In a traditional implementation the appearance of a constraint such as $k = 200$ immediately leads to a large number of other constraints (edges) being introduced as a consequence. Namely, for each variable v (other than k), the implied constraint $k - v \leq 200 - v_{lo}$ should be added, if a lower bound v_{lo} for v can be deduced from the current state of affairs. Similarly, $v - k \leq v_{hi} - 200$ should be added, if an upper bound v_{hi} can be deduced. If a program with m variables starts out by initialising those variables, we have immediately created $O(m^2)$ constraints (in fact we have created a complete graph), even though no interesting relations have been created among the m variables. For the example of Figure 1(a), every

pair of variables gives rise to two constraints, but most of these (such as the relations between k and n) are quite shallow, as they are immediate from the fact that the variables are fixed. Our observation is that there is no good reason to waste space on constraints that are shallow in this sense. The example program in Figure 1(b) illustrates that the same observation applies to Octagons analysis.

In this paper we describe, in detail, algorithms for Zones and Octagons based on carefully crafted sparse-graph representations. Our experimental evaluation shows a clear improvement, compared to other contemporary implementations. Our paper builds on our earlier work [28]. In that paper we discussed only the case of Zones and we:

- introduced new data structures and algorithms, based on refined shortest-path graph algorithms, including a specialised incremental variant of Dijkstra's algorithm;
- proposed a graph representation that used "sparse sets" [9], tailored for efficient implementation of Zones;
- proposed the *split normal form* for weighted digraphs, with the aim of preserving many essential closure properties while avoiding unnecessary "densification" of graphs.

Many details were excluded owing to space limitations, and explanations were brief. In the present paper, we revisit those algorithms, and we extend our ideas to the implementation of Octagons. We provide *complete* sets of (improved) algorithms for better implementation of both domains, including better widening operators. We provide more explanation and examples, relative to the exposition by Gange *et al.* [28]. Importantly, we conduct a comprehensive experimental evaluation of the implementations, to determine the effect on speed and precision. We extend the comparison to also explore the effect of using variable packing as a conjunct to the analysis.

Implementations (including source code) are available as part of the Crab abstract interpretation-based framework [21]. A shared platform allows us to perform two important kinds of comparison: First, using standard sets of benchmarks, we compare our analysis tool against ELINA [58], the state-of-the-art implementation of Zones and Octagons. Second, we compare, in an equitable setting, the relative advantages of Zones and Octagons for the purpose of program verification. Specifically we assess the trade-off involved in choosing between the two: performance, precision, and the degree to which greater precision translates to higher success rates in program verification tasks.

The paper is structured as follows. Section 2 covers concepts and definitions that will be used later in the paper. Section 3 is a study of the Zones abstract domain, with detailed algorithms of the required abstract operations. These rest on classical shortest-path algorithms, but we provide a number of improvements that take advantage of the particular application (abstract interpretation). In Section 4 we consider the desirable case of sparse graphs and consider data structures that can capitalise on sparsity. At first this may seem irrelevant for the application, because program analysis using the Zones domain tends to operate with dense graphs. However, in Section 5 we show that this is a situation that can be rectified. We introduce a new representation (split normal form) that makes Zones analysis consume fewer space and time resources. Section 6 extends the study to the Octagons abstract domain, and in Section 7 we show how similar sparsity-preserving algorithmic improvements can be applied to the Octagons domain. Section 8 gives an account of various experimental evaluations we have made. Section 9 puts our contribution in the context of related work, and Section 10 concludes.

We assume the reader is familiar with order theory and concepts from the field of abstract interpretation [17, 18]. We also assume familiarity with basic graph concepts and algorithms, including the classical shortest-path algorithms [15].

Table 1. Glossary of notation.

$x \xrightarrow{k} y$	Edge from x to y with weight k
$wt_E(x_1, \ldots, x_k)$	Weight of path $x_1 \to x_2 \to \ldots \to x_k$ in E, ∞ if no such path exists.
$E(x)$	Set of edges in E emanating from x
$incid_E(x)$	Set of edges in E incident on x
$E_1\{\oplus, \otimes\}E_2$	Pointwise max/min over edge weights
$E_1 \boxminus E_2$	Symmetric difference of E_1 and E_2
$rev(E)$	Graph E with all edges reversed
$E \setminus \{v\}$	Graph E excluding (all edges incident to) vertex v

2 PRELIMINARIES

2.1 Digraph Operations

Table 1 provides a glossary of notation that we use. Much of what follows deals with operations on weighted directed graphs.

We let $x \xrightarrow{k} y$ denote a directed edge from x to y with weight k. We think of a graph simply as the set E of its edges, tacitly assuming a fixed set V of vertices for all graphs.[1] $wt_E(x, y)$ denotes the weight of the edge $x \to y$ in E (or ∞ if the edge is absent). More precisely,[2]

$$wt_E(x,y) = \begin{cases} 0 & \text{if } x = y \\ k & \text{if } x \xrightarrow{k} y \in E \text{ (and } x \neq y) \\ \infty & \text{otherwise} \end{cases}$$

This is generalized to a directed path $x_1 \to \ldots \to x_k$ as $wt_E(x_1, \ldots, x_k) = \sum_{i=1}^{k-1} wt_E(x_i, x_{i+1})$.

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.

We let $E_1 \oplus E_2$ and $E_1 \otimes E_2$ denote the pointwise maximum and minimum over a pair of graphs, and we let $E_1 \boxminus E_2$ denote the symmetric difference. That is:

$$\begin{aligned} E_1 \oplus E_2 &= \left\{ x \xrightarrow{\max(k_1,k_2)} y \;\middle|\; x \xrightarrow{k_1} y \in E_1,\ x \xrightarrow{k_2} y \in E_2 \right\} \\ E_1 \otimes E_2 &= \left\{ x \xrightarrow{k} y \;\middle|\; x \xrightarrow{k} y \in E_1,\ k \leq wt_{E_2}(x,y) \right\} \cup \left\{ x \xrightarrow{k} y \;\middle|\; x \xrightarrow{k} y \in E_2,\ k < wt_{E_1}(x,y) \right\} \\ E_1 \boxminus E_2 &= (E_1 \otimes E_2) \setminus (E_1 \oplus E_2) \end{aligned}$$

Example 2.1. Let $E_1 = \{x \xrightarrow{3} y, y \xrightarrow{5} z\}$ and let $E_2 = \{x \xrightarrow{7} y, y \xrightarrow{5} z, z \xrightarrow{0} y\}$. Then we have $E_1 \oplus E_2 = \{x \xrightarrow{7} y, y \xrightarrow{5} z\}$, $E_1 \otimes E_2 = \{x \xrightarrow{3} y, y \xrightarrow{5} z, z \xrightarrow{0} y\}$, and $E_1 \boxminus E_2 = \{x \xrightarrow{3} y, z \xrightarrow{0} y\}$. □

In several 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 $y \xrightarrow{k} x$). $incid_E(v)$ is the set of edges in E which are incident on v. We use $E \setminus \{v\}$ as a shorthand for $E \setminus incid_E(v)$, the graph obtained by removing from E every edge involving v. $E \setminus E'$ is the graph obtained from E by deleting all edges in E'. These are simply mathematical definitions; in our implementations these

[1] This is for presentation purposes only. In practice, it is unnecessarily expensive—we instead maintain vertices only for in-scope variables and add or remove vertices as needed.

[2] The definition of wt_E is mainly for presentational purposes; in practice no edge of form (x, x) is ever constructed, no edge will have weight ∞ (rather it will be absent), and there will be at most one weighted edge (x, y) in E.

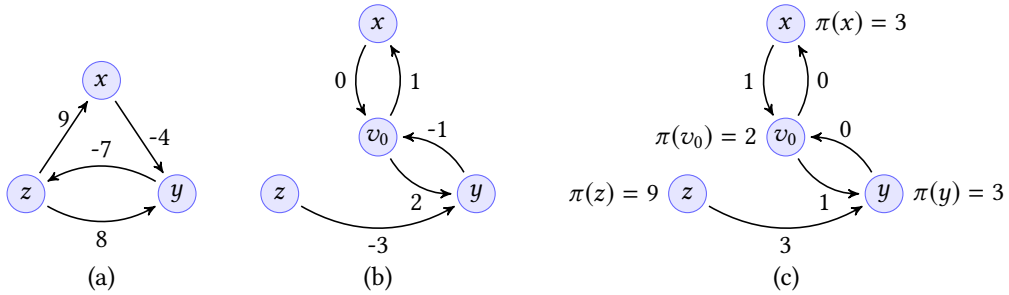


Fig. 2. (a) Difference constraints in graphical form. (b) Constraint graph representing the difference constraints $\{x \in [0,1], y \in [1,2], y - z \leq -3\}$. (c) The *slack* graph (with all non-negative weights) under potential function $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto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

graphs are never explicitly constructed—they merely define different interpretations of an existing graph, and we implement the symmetric difference operator $\boxminus$ directly, rather than via $\oplus$ and $\otimes$.

2.2 Difference Constraints and Weighted Digraphs

A *difference constraint* is of the form $x - y \leq k$, where k is a constant and x and y are variables that range over a numerical domain D ($\mathbb{R}$, $\mathbb{Q}$ or $\mathbb{Z}$). Note that a constraint of the form $x - y \geq k$ can be translated to the $\leq$ form: $y - x \leq -k$. A *difference constraint system* is a conjunction of primitive difference constraints, often represented as a *set* of constraints. Clearly constraints of form $x - y = k$ can be expressed as difference constraint systems. For cases where variables range over $\mathbb{Z}$, we can also map strict inequality to this form. For example, $y - x < k$ is equivalent to $y - x \leq k - 1$. For non-integer domains we usually weaken the strict inequality to be non-strict, for example, $y - x < k$ is weakened to $y - x \leq k$.

Difference constraint systems[3] are conveniently represented as weighted directed graphs, with an edge $x \xrightarrow{k} y$ (that is, $wt_E(x, y) = k$) for each constraint $y - x \leq k$. If E satisfies the "triangle" inequality

$$wt_E(x, z) \leq wt_E(x, y) + wt_E(y, z) \text{ for all } x, y, z \in V \tag{1}$$

then we say that E is *tr-closed*.

It is not hard to see that a difference constraint system has a solution if and only if the corresponding digraph contains no negative-weight cycle. Also note that if the assignment σ of values to variables $v \in V$ satisfies a difference constraint system, then so do the infinitely many assignments of form $\{v \mapsto \sigma(v) + \delta\}$ for any constant δ.

Example 2.2. Figure 2(a) captures the system $\{x - y \geq 4, 7 \leq y - z \leq 8, x - z \leq 9\}$. The negative weight of the cycle $x \to y \to z \to x$ shows the set of constraints is unsatisfiable. □

2.3 Solving Difference Constraints

Let C be a set of difference constraints and let E be a weighted digraph. Define

$$graph(C) \;=\; \left\{ x \xrightarrow{k} y \;\middle|\; y - x \leq k \text{ is a constraint in } C \right\}$$

Say that C *bounds* $y - x$ iff

(1) C is satisfiable, and

[3]For brevity we may drop 'systems' when it does not introduce ambiguity.

(2) $C \models y - x \leq k$ for some $k \in D$.

Let $k \in D$. We say that C *k-bounds* $y - x$ iff

(1) C bounds $y - x$,
(2) $C \models y - x \leq k$, and
(3) for all $k' \in D$, if $C \models y - x \leq k'$ then $k \leq k'$.

Finally, for a set C of difference constraints, define

$$\mathit{consequences}(C) \;=\; \left\{ y - x \leq k \;\middle|\; C \text{ } k\text{-bounds } y - x \right\}$$

and let *tr-close* be the function that takes a weighted digraph and computes its all-pairs shortest paths. Then for satisfiable C

$$\mathit{graph}(\mathit{consequences}(C)) \;=\; \mathit{tr\text{-}close}(\mathit{graph}(C)).$$

That is, if C is satisfiable then C's closure under tight entailment can be determined by well understood graph algorithms. If we take the graph $\mathit{graph}(C)$ for a constraint system C and extend that graph with a fresh vertex v_0, together with an edge $v_0 \xrightarrow{0} v$ for each $v \in V$, then we have the *constraint graph* for C [15]. In the constraint graph, every vertex $v \in V$ is reachable from a single vertex, namely v_0. Determining whether C has a feasible solution comes down to checking whether its constraint graph has a negative-weight cycle, and in the absence of such cycles, a solution to C can be found by solving the shortest-path problem for the graph, taking v_0 as the source. The Bellman-Ford algorithm [7, 26] determines feasibility *and* calculates single-source shortest paths (that is, tight entailment) in time $O(|V||E|)$, where V is the set of vertices and E is the set of edges.

In many uses of difference constraints we also want to allow unary constraints, that is, constraints of the form $x \leq k$ and $x \geq k$. Fortunately, all that is needed for this extension is to utilise the extra vertex v_0 appropriately. We simply assume that v_0 is a "variable" constrained to take the value 0. This way, an edge $v_0 \xrightarrow{k} x$ represents the constraint $x \leq k$, and $x \xrightarrow{k} v_0$ represents $x \geq -k$. We shall refer to such constraints/edges as "bounds constraints" or "bounds relations", and those for proper differences $y - x$ as "binary constraints" or "binary relations".

Example 2.3. Consider the system of constraints $\{x \in [0, 1], y \in [1, 2], y - z \leq -3\}$. The corresponding constraint graph is shown in Figure 2(b). Note that interval constraints $x \in [lo, hi]$ are encoded as edges $v_0 \xrightarrow{hi} x$ and $x \xrightarrow{-lo} v_0$. □

The extension to constraint graphs does not affect how closure under tight entailment can be derived. For example, the constraints $u \leq 4$ and $v \geq 3$ are represented as $u - v_0 \leq 4$ and $v_0 - v \leq -3$, from which we derive $u - v_0 + v_0 - v \leq 4 - 3$, that is, $u - v \leq 1$. Note that this simply uses (1) with $y = v_0$.

Dijkstra's algorithm for the *single-source* shortest path problem is an efficient greedy algorithm, but it does not work in the presence of negative weights. Note that there is no simple reduction of the shortest-path problem for graphs with negative weights to the more tractable positive-weight problem. For example, identifying the smallest negative weight w in the graphs and adding w to all weights is not a valid reduction. However, Nemhauser [51] discovered a reduction that utilises a certain mapping from vertices to values. We call this type of mapping a *potential function*[4].

If the mapping is a model, that is, if it assigns values to vertices in such a way that the constraints represented by the graph are satisfied, then we refer to it as a *valid* potential function.

A valid potential function allows one to translate a graph G into a version G' that has non-negative weights only, while preserving shortest paths. This is useful, because it means Dijkstra's

[4]The term is commonly used in description of certain network flow algorithms and most likely inspired by the concept of electric potential.

algorithm becomes applicable. A shortest path found for G' can then be translated back to a shortest path in G, again using the potential function (we give an example shortly). This reduction is utilised in Johnson's shortest-path algorithm [37], which is specifically designed for the case of sparse graphs and which uses both Dijkstra's algorithm and Bellman-Ford as subroutines. Johnson's algorithm solves the all-pairs shortest path problem in time $O(|V|^2 \lg|V| + |V||E|)$.

Potential functions are also used in a method for solving sparse systems of difference constraints. Cotton and Maler [16] show that a potential function π is particularly useful in *incremental* constraint solving. In constraint solving terms, π allows a constraint graph E to be reformulated: For a constraint $y - x \leq k$ in E, the *slack* (or *reduced cost* [16]) is given by $\pi(x) + k - \pi(y)$. In the *slack graph* for E and π, each edge $x \rightarrow y$ is given weight $k' = \pi(x) + k - \pi(y)$. If π is a model for the constraint set then each k' is non-negative, so shortest paths in the *slack graph* can be found with Dijkstra's algorithm and translated back to shortest paths in the original E.

Example 2.4. Consider again the constraints captured in the graph in Figure 2(b). 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 where each weight is replaced by its slack under π is given in Figure 2(c). 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 Figure 2(c), 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 Figure 2(b). □

Cotton and Maler show how to utilise and maintain π when a single difference constraint is added to a constraint system. Whenever an edge is added, they update π to provide a model of the augmented system (if possible—otherwise unsatisfiability is reported). The operations that we require for the program analysis problem differ somewhat from those covered by Cotton and Maler, but for some operations we can exploit Cotton and Maler's idea.

2.4 Graph Representations

The usual representation of the Zones domain is in terms of *difference bound matrices* (DBMs). A DBM explicitly represents a weighted graph E as a square matrix M with one row and column for each vertex. Its entries record the weights of the edges in E. The element $M[x, y]$ contains k where $x \xrightarrow{k} y \in E$ and ∞ otherwise except that the diagonal $M[x, x]$ is always 0. Commonly implementations of Zones keep this difference bound matrix tr-closed, in the sense defined above for graphs. This makes many abstract operations for Zones very simple to define.

Example 2.5. Consider the set of difference constraints defined in Example 2.3. The corresponding difference bound matrix is shown in Figure 3(a). The closed form of the difference bound matrix is shown in Figure 3(b). The closed form under potential function $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9\}$ is shown in Figure 3(c). Note that now all entries in the matrix are non-negative. □

There are algorithms for finding a so-called transitive reduction of a directed graph [1]. Given a graph G, a transitive reduction of G is a minimal graph G' (not necessarily a sub-graph of G) such that G and G' have the same transitive closure. While the idea can be extended to weighted graphs, and while it promises to promote sparsity, transitively reduced graphs are not adequate for the program analysis problem. The reason is that operations such as lattice-theoretic join rely on certain entailed constraints to be explicit. Implementations of Zones and Octagons therefore traditionally operate with dense graphs. Usually the system of relations is encoded as a dense matrix, and closure is obtained by running the Floyd-Warshall algorithm [25]. A main contribution

	v_0	x	y	z
v_0	0	1	2	∞
x	0	0	∞	∞
y	-1	∞	0	∞
z	∞	∞	-3	0

(a)

	v_0	x	y	z
v_0	0	1	2	∞
x	0	0	2	∞
y	-1	0	0	∞
z	-4	-3	-3	0

(b)

	v_0	x	y	z
v_0	0	0	1	∞
x	1	0	2	∞
y	0	0	0	∞
z	3	3	3	0

(c)

Fig. 3. Example of DBM in the Zones domain for the set of constraints of Example 2.3 in (a) raw form, (b) closed, (c) closed wrt to potential function $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9\}$.

of this paper is to identify an intermediate position, namely a sparse graph representation that is more suitable for the problem at hand.

While the worst-case time complexity for shortest-path closure may be large, the actual execution time can be short if the constraint graph is sparse. Luckily, this is often the case, with many instances of variables being unrelated [58]. Nevertheless, in practice, the constraint graphs tend to be extremely dense at early iterations of analysis, with sparsity only appearing after widening. This "*phantom density*" is induced by variable bounds, as discussed in the introduction. Note that the difference relations that are merely consequences of variable bounds relations do not give any new information to improve precision. However, they do cause the graph to become dense very quickly. In fact, if all variables have both upper and lower bounds, the constraint graph immediately becomes complete.

To avoid the unnecessary density and make tr-closure more efficient, we introduce, in Section 5, *Split Normal Form* [28] which avoids adding bounds-induced relations. This is achieved by omitting information of variable bounds from the graph when restoring tr-closure, then refining variable bounds where possible.

3 ZONES

In this section we provide abstract operations for the Zones domain. The efficiency of an abstract domain depends on how it is used. When used directly for forward analysis of a program, we observe mostly variable assignments and additions of single constraints interspersed with joins; meets are reasonably infrequent, as meet is used only in state transformers for function calls. Hence we target our implementation to the more frequent operations.

For the Zones domain an abstract state φ is either $\bot$ or a pair $\langle \pi, E \rangle$. In the latter case, π is a valid potential function, and E is a sparse graph of difference constraints E over vertices $V \cup \{v_0\}$. The intended meaning $[\![\varphi]\!]$ of an element φ is the set of satisfying valuations:

$$[\![\varphi]\!] = \begin{cases} \emptyset & \text{if } \varphi = \bot \\ \left\{ \mu \in V \to D \;\middle|\; (x \xrightarrow{k} y \in E) \Rightarrow (\mu(v_0) = 0 \wedge \mu(y) - \mu(x) \leq k) \right\} & \text{if } \varphi = \langle \pi, E \rangle \end{cases}$$

Notice the meaning of the abstract state *does not depend on* the potential function π. Indeed we could at any time compute a correct potential function from E using Bellman-Ford. But since the focus of this paper is on efficient implementations of Zones (and Octagons) we include it, as recomputing π is clearly inefficient.

We assume the representation of E supports cheap initialization, as well as constant time insertion, lookup, removal and iteration; we discuss a suitable representation in Section 4.

```
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 each s -k′-> x ∈ E
        if (k′ + k < wt_E(s, y))
            δ := δ ∪ {s -k′+k-> y}
            S := S ∪ {s}
    for each y -k′-> d ∈ E
        if (k + k′ < wt_E(y, d))
            δ := δ ∪ {x -k+k′-> d}
            D := D ∪ {d}
    for each (s, d) ∈ S × 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. 4. Addition of (non-redundant) edge $x \xrightarrow{k} y$, including restoration of closure.

3.1 Ordering

The ordering $\sqsubseteq$ on the abstract domain is induced by the semantic function $[\![\cdot]\!]$: $\varphi \sqsubseteq \varphi'$ iff $[\![\varphi]\!] \subseteq [\![\varphi']\!]$. For non-$\bot$ elements, the test comes down to whether one set of constraints is entailed by another set. An algorithm to decide $E_1 \sqsubseteq E_2$ simply finds the truth value of

$$\forall x, y \in V \cup \{v_0\}(wt_{E_1}(x, y) \leq wt_{E_2}(x, y))$$

3.2 Variable Elimination

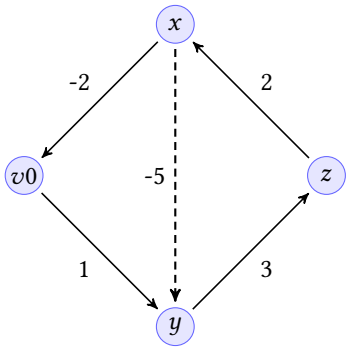
To eliminate a variable x, we simply remove all edges incident to x. Assuming we can remove a specific edge in constant time, this takes worst case O($|V|$) time. The result is clearly tr-closed if the original state was tr-closed.

3.3 Constraint Addition

When adding a single edge $x \xrightarrow{k} y$, we utilise an idea from Cotton and Maler [16]. Figure 4 shows the details. First, we need to repair the potential function (and check for infeasibility in the process). Then function close-edge iterates through edges incoming to x and outgoing from y to check for updated paths across $x \rightarrow y$. The function separates the three cases: edges $s \rightarrow y$, edges $x \rightarrow d$, and edges $s \rightarrow d$, where $s \rightarrow x \in E$ and $y \rightarrow d \in E$. The repair step has worst-case complexity O($|V| \log |V| + |E|$), and restoring closure is O($|V|^2$). In a dense representation this worst case is frequently hit, and even the best case is O($|V|$). Later when we introduce sparse representations we expect this worst-case behaviour to be very infrequent—in a sparse graph, a single edge addition should affect very few shortest paths.

```
restore-potential(π, x -k-> y, E)
    for each v ∈ V
        γ(v) := 0
        π'(v) := π(v)
    γ(y) := π(x) + k − π(y)
    while (min(γ) < 0 ∧ γ(x) = 0)
        s := argmin(γ)
        π'(s) := π(s) + γ(s)
        γ(s) := 0
        for each s -k'-> t ∈ E
            if (π'(t) = π(t))
                γ(t) := min{γ(t), π'(s) + k' − π(t)}
    if (γ(x) < 0)
        return inconsistent
    return π'
```

(a)



(b)

Fig. 5. (a) Cotton and Maler's algorithm for repairing a potential function after addition of an edge $x \xrightarrow{k} y$. (b) Adding an edge requires the potential function to be repaired.

The algorithm that finds a repaired potential function (if one exists) is shown in Figure 5(a). When adding an edge $x \xrightarrow{k} y$, the algorithm first considers y, to see whether $\pi(y)$ needs to be updated. The function γ keeps a record of how potential values are required to change, in the presence of the new edge. When the value of a vertex changes, only its successor vertices are considered for a possible update. This way, large parts of the constraint graph may not need inspection. If the new edge $x \xrightarrow{k} y$ gives rise to a negative-weight cycle, that will manifest itself as a need to decrease the value of $\pi(x)$.

Example 3.1. Consider the constraint system $\{x \geq 2, y \leq 1, x - z \leq 2, z - y \leq 3\}$. The constraint graph is shown as the solid edges in Figure 5(b). Assume the potential values are $\pi(v_0) = 0, \pi(x) = 4, \pi(y) = 1, \pi(z) = 3$. If we now add the constraint $y - x \leq -5$, that is, the edge $x \xrightarrow{-5} y$ shown dashed in Figure 5(b), restore-potential will calculate $\gamma(y) = \pi(x) + k - \pi(y) = -2$ as the adjustment that $\pi(y)$ requires, in order to satisfy the new constraint. The new potential value for y becomes -1, after which the edge $y \xrightarrow{3} z$ can be considered. This leads to $\gamma(z)$ being updated, to $-1 + 3 - 3 = -1$, so that the new potential value for z becomes 2. Finally, the edge $z \xrightarrow{2} x$ is considered: $\gamma(x) = 2 + 2 - 4 = 0$, so no further updates to the potential function take place. □

3.4 Assignment

The key to efficient processing of an assignment statement is this observation: executing $[\![\mathsf{x} := \mathsf{S}]\!]$ can only introduce relationships between x and other variables; it cannot tighten any existing relation.[5] From the current state $\varphi = \langle \pi, E \rangle$, 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

[5] Assuming $\mathcal{E}(\mathsf{S})$ is a total function. Where, say, integer division is partial we first close with respect to x, then enforce the remaining invariants.

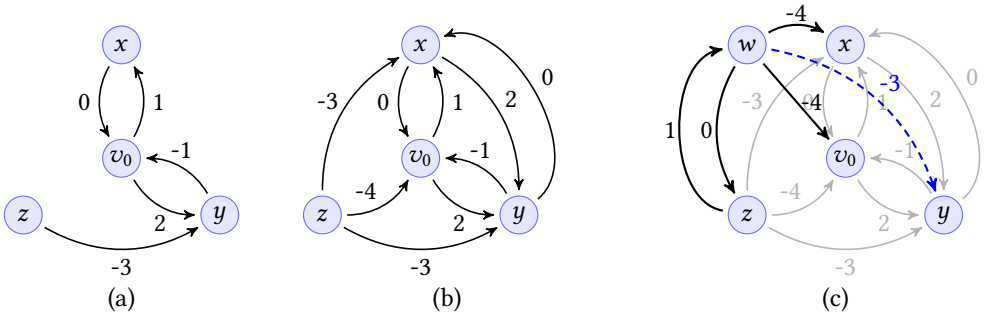


Fig. 6. (a) The graph from Example 2.3. (b) Its closure. (c) Edges introduced after evaluating ⟦w := x + z⟧.

algorithm to collect the consequences. Before we give a detailed algorithm, let us work through an example.

Example 3.2. [28] Consider again the state shown in Figure 2(b). For easy reference it is repeated as Figure 6(a). Its tr-closure is shown in Figure 6(b). We wish to evaluate ⟦w := x + z⟧. Using the potential function from Example 2.4, that is, $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9\}$, we first compute a valid potential for w, from the potentials for x and z:

$$\pi(w) = \pi(v_0) + (\pi(x) - \pi(v_0)) + (\pi(z) - \pi(v_0)) = 2 + (3 - 2) + (9 - 2) = 10$$

Using the natural propagation rules for +, utilising the known bounds for x (0..1) and z (4..∞), the new difference constraints are as follows:

from	$w \le x_{hi} + z_{hi}$	derive	true
from	$w \ge x_{lo} + z_{lo}$	derive	$v_0 - w \le -4$
from	$w - x \le z_{hi}$	derive	true
from	$w - x \ge z_{lo}$	derive	$x - w \le -4$
from	$w - z \le x_{hi}$	derive	$w - z \le 1$
from	$w - z \ge x_{lo}$	derive	$z - w \le 0$

The resulting four new edges are shown in Figure 6(c). Running Dijkstra's algorithm to/from w, we also find the edge $w \xrightarrow{-3} y$, corresponding to $y - w \le -3$ (shown dashed in Figure 6(c)). □

Other arithmetic operators are handled in a similar way. Coefficients (outside of $\{-1, 0, 1\}$) on the right-hand side of an assignments do not prevent extraction of difference constraints. For example, ⟦w := x + 7y⟧ can be transformed to ⟦w := y + x + 6y⟧, from which we can extract bounds on $w - y$ and on $y - w$. Namely, $w \le y + ub(x + 6y)$ and $w \ge y + lb(x + 6y)$, where ub and lb calculate upper and lower bounds, respectively, of expressions. In Figure 7 we assume that, given an assignment w := S, function edges-of-assign produces all such new edges that link w to variables occurring in S.

Example 3.2 suggested the use of Dijkstra's algorithm to establish tr-closure. But, since the incoming E is already tr-closed, we can do better than running a full Dijkstra's algorithm. Assume some shortest path from x to z passes through $[x, u_1, \ldots, u_k, z]$. As E is closed there must be some edge (u_1, z) such that $wt_E(u_1, z) \le wt_E(u_1, \ldots, u_k, z)$; thus, we never need to expand grand-children 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 that has already been expanded. However, recall that π allows us to reframe E 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.

```
assignment(⟨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⟩

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 each i ∈ {1, ..., k}
                e_i := eval-expr(π, s_i)
            return f(e_1, ..., e_k)

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)

close-assignment-fwd(⟨π, E⟩, x)
    for each v ∈ V
        reach(v) := 0
        dist(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 each 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)}
```

Fig. 7. Updating the abstract state under an assignment.

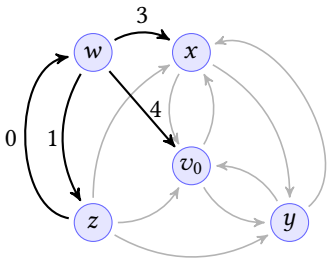


Fig. 8. Edges introduced in Example 3.3, re-cast in terms of slack.

```
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. 9. Computing the meet over sparse graphs. The function compute-potential is defined in Figure 10, close-meet in Figure 11.

Thus, unlike Dijkstra's algorithm, close-assignment-fwd has no need of a priority queue. It instead simply expands children of x in order of increasing slack, collecting the minimum distance to each grandchild. The whole improved algorithm for restoring closure after an assignment is given in Figure 7. The worst-case complexity of this algorithm is $O(|S| \log |S| + |E|)$ (here $|S|$ denotes the number of variables in the expression S). 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 the inverted graph $rev(E)$ is not explicitly computed).

Example 3.3. Consider again the assignment $[\![\mathtt{w} := \mathtt{x} + \mathtt{z}]\!]$ in Example 3.2. The slack graph, with respect to potential function $\pi = \{v_0 \mapsto 2, x \mapsto 3, y \mapsto 3, z \mapsto 9, w \mapsto 10\}$, is shown in Figure 8.

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 all 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 overall algorithm is given in Figure 9.

Our method for computing a valid potential function is similar to the approach found in Johnson's algorithm [37]. Johnson's algorithm uses the Bellman-Ford algorithm [7, 26] as a platform for calculating a valid potential function (or determining that none exists). A plethora of refinements and variants are known, see Cherkassky and Goldberg [12], any of which could be applied here. Our

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

Fig. 10. Warm-started Bellman-Ford algorithm. We assume connected components are ordered topologically.

approach is to build the construction of a potential function into the basic Bellman-Ford algorithm, with a few minor refinements:

- π' is initialized from π_1 or π_2 (we say it is "warm started").
- Bellman-Ford is run separately on each strongly-connected component.
- We maintain separate queues for the current and next iteration.

This modified Bellman-Ford algorithm is given in Figure 10. Notice that if $\pi'(x)$ changes but x is still in Q, there is no need to add it to Q'—its successors will already have been updated by the end of the current iteration.

Consider again Figure 9. The straightforward approach to restoring tr-closure of $E = E_1 \otimes E_2$ 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 tr-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. We think of 1 and 2 as separate colours. Observe that if all edges reachable from some vertex v have the same colour then the subgraph from v is already closed.

To see how the colouring can save work, consider the behaviour of Dijkstra's algorithm. We expand some vertex v, adding $\bullet \xrightarrow{k} x$ to the priority queue. Assume the edge $v \xrightarrow{k} x$ originated from the set E_1. At some point, we remove $\bullet \xrightarrow{k} x$ from the queue. Now 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 priority 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.[6]

The complete algorithm for restoring tr-closure is presented in Figure 11. We run Dijkstra's algorithm as usual, except any time we find a minimum-length path to some vertex 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. In practice, the initialization of *dist* and *edge-col* is performed only once and preserved between calls, rather than performed explicitly for each call. Notice that the potential function π passed to close-meet is needed by the chromatic Dijkstra's algorithm, for the maintenance of its priority queue Q.

Example 3.4. Consider the conjunction of two closed states shown in Figure 12(a), one in red and one in blue. To restore closure we run the closure-aware Dijkstra's algorithm from each vertex.

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

Selecting w as origin, we add $\bullet \xrightarrow{0} x$ and $\bullet \xrightarrow{2} y$ to the queue, both marked as reachable via red (dashed) edges. We then process x. As x is reachable via red, we must expand its blue children. The edge $x \xrightarrow{1} y$ provides an improved path to y, so we update the distance and mark y as reachable instead via 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 Figure 12(b). □

3.6 Join

For $\langle \pi_1, E_1 \rangle \sqcup \langle \pi_2, E_2 \rangle$, both π_1 and π_2 are valid potential functions, so we can choose either. We then collect the pointwise maximum $E_1 \oplus E_2$, see Figure 13. If E_1 and E_2 are tr-closed, $E_1 \oplus E_2$ is also tr-closed, so the overall result is simply $\langle \pi_1, E_1 \oplus E_2 \rangle$. Assuming we can look up a specific edge in constant time, this takes worst case $O(\mathbf{min}(|E_1|, |E_2|))$.

3.7 Widening

For widening, we follow the usual practice of discarding *unstable* edges—those edges that have weakened in successive iterates. We consider each edge $x \xrightarrow{k_2} y \in E_2$ (in an ascending sequence, E_2 has fewer edges), and add $x \xrightarrow{k_1} y$ to $E_1 \bigtriangledown E_2$ iff $x \xrightarrow{k_1} y \in E_1$ *and* $k_2 \leq k_1$. Unlike the join, this does *not* necessarily preserve closure, so we must restore closure before subsequent operations.[7]

We omit a formal algorithm here, returning instead to the problem of widening in Section 5.5.

4 SPARSE GRAPH REPRESENTATIONS

So far we have avoided discussion of the underlying graph representation. However, choosing an appropriate representation of the constraint graph is critical for performance. Upon a meet or join, 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 elimination of a variable v, we must remove all edges to or from v.

Conventional representations handle only some of these efficiently. Dense matrices are convenient for updating specific entries, but cannot iterate over only the non-trivial entries. Meet

[6] 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.

[7] Except subsequent widenings, which must use the *un-closed* result.

```
close-meet(π, E, E1, E2)
    for each (x, y) ∈ V²
        edge-col(x, y) := ∅
    for each x -k-> y ∈ E1
        if (wtE(x, y) = k)
            edge-col(x, y) := edge-col(x, y) ∪ {1}
    for each x -k-> y ∈ E2
        if (wtE(x, y) = k)
            edge-col(x, y) := edge-col(x, y) ∪ {2}
    δ := ∅
    for each x ∈ V
        δ := δ ∪ chromatic-Dijkstra(⟨π, E⟩, x)
    return δ
chromatic-Dijkstra(⟨π, E⟩, x)
    for each v ∈ V
        dist(v) := ∞
    Q := init(λx . dist(x) + π(x))
    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) < wtE(x, y))
            δ := δ ∪ {x -dist(y)-> y }
        % Iterate through edges of the other colour
        for each c ∈ {1, 2} \ reach-col(y)
            for each y -k-> z ∈ Ec(y)
                dxyz := dist(y) + k
                if (dxyz = dist(z))
                    reach-col(z) := reach-col(z) ∪ edge-col(y, z)
                if (dxyz < dist(z))
                    dist(z) := dxyz
                    Q.update(z, π)
                    reach-col(z) := edge-col(y, z)
    return δ
```

Fig. 11. Dijkstra's algorithm modified to exploit closed operands.

and join 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 efficiently supports all the operations we require is the *adjacency hash-table*, consisting of a hash-table mapping successors to weights for each vertex, and a hash-set

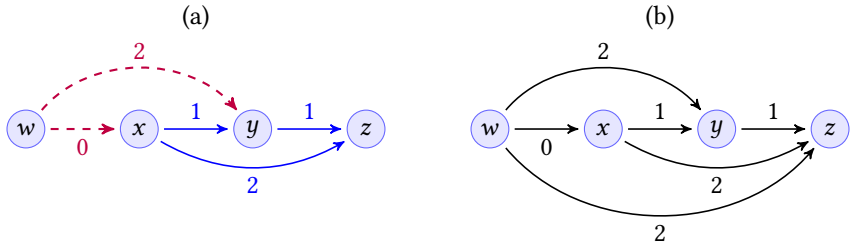


Fig. 12. (a) The conjunction of two tr-closed graphs (E_1, dashed red, corresponds to $\{x - w \leq 0, y - w \leq 2\}$ and E_2, blue, corresponds to $\{y - x \leq 1, z - y \leq 1, z - x \leq 2\}$. (b) The tr-closure of $E_1 \otimes E_2$.

```
join(⟨π1, E1⟩, ⟨π2, E2⟩)
    return ⟨π1, E1 ⊕ E2⟩
```

Fig. 13. The join of two tr-closed graphs.

of the predecessors of each vertex. This would provide the asymptotic behaviour we want but is rather heavyweight, with substantial overheads on operations.

We instead adopt a hybrid representation: weights are stored in a dense *but uninitialized* matrix, and adjacencies are stored using a "sparse-set" structure [9]. 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 by traversing $\{dense[0], \ldots, dense[sz - 1]\}$.

The sparse-set representation trades memory consumption to improve efficiency of primitive operations. It introduces an overhead of roughly 8 bytes per matrix element[8]—2 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. We shall see later, however, that this trade-off typically falls in our favour.

Pseudo-code for primitive sparse-set operations are given in Figure 14. Note that we preserve this invariant:

$$\forall\, i \in [0, sz)\ .\ sparse[dense[i]] = i$$

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

This gives us a graph representation with O(1) constraint addition, removal, lookup and enumeration (with very low constant factors), and $O(|V| + |E|)$ time to initialize or copy. [9]

5 SNF ZONES: IMPROVED PERFORMANCE THROUGH SPLIT NORMAL FORM

In previous sections, we made the key assumption that the abstract states were sparse. During the later stages of analysis, this is typically the case. However, abstract states in early iterations are often complete; Singh et al. [58] observed that the first 50% of analysis iterations were extremely

[8]This assumes that vertex identifiers fit in 16 bits. If there are more than 2^{16} variables in scope at a program point, any approach using a dense matrix is already impractical—instead use the hash-table representation, hope the graph is exceptionally sparse, and consider using a different domain.

[9]We could reduce this to $O(|E|)$ by including an index of non-empty rows, but this adds an additional cost to each lookup.

```
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. 14. Basic operations on sparse sets.

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

Fig. 15. At line 4, the only constraint *not* implied by variable bounds is $x_2 = x_1 + 1$.

dense, sparsity only appearing after widening. But closer scrutiny reveals this initial completeness as a mirage.

Recall the discussion in Section 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 under entailment: if variables are given initial bounds, the abstract state is represented by a complete graph.

This is regrettable, as it undermines the sparsity we intend 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 should only need to care about binary relations that are *not* already implied by variable bounds.

Example 5.1. Consider the program fragment shown in Figure 15. Variables $x_1, \ldots, x_k$ are initialised to constants at line 0. At that point, a direct implementation of Zones (or Octagons) will compute $k(k-1)$ pairwise relations implied by these bounds. During the execution of lines 2 and 3, each of these relations will be updated, despite all inferred relations being simply the consequences of variable bounds.

At line 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 $x_2 = x_1 + 1$. □

We could avoid this phantom density by storing the abstract state in a (possibly weakly) *transitively reduced* form, an approach that has been successful in constraint programming [24] and

SMT [16] contexts.[10] Unfortunately, we are hindered by the need to perform frequent *join* operations. The join of two tr-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. Hence transitive reduction is not a good approach.

Instead, we construct a modified normal form which distinguishes independent properties (edges to/from v_0) from strictly relational properties (edges between program variables). A graph E is in *split normal form* iff:

- $E \setminus \{v_0\}$ is tr-closed.
- For every path $v_0 \to x_1 \to \ldots \to x_m$ in E, there is an edge $v_0 \xrightarrow{k} x_m$ such that $k \leq wt_E(v_0, x_1, \ldots, x_m)$.
- For every path $x_m \to \ldots \to x_1 \to v_0$ in E, there is an edge $x_m \xrightarrow{k} v_0$ such that $k \leq wt_E(x_m, \ldots, x_1, v_0)$.

This means that if E is in split normal form, any shortest path in E from x to y occurs either as an edge $x \xrightarrow{k} y$, or else as a path $x \xrightarrow{k_1} v_0, v_0 \xrightarrow{k_2} y$.

Note that split normal form is *not* a canonical form: 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 $E \setminus \{v_0\}$, but would we gain nothing by doing so, since we already have an efficient entailment test.

A state in the domain of *split constraint graphs* is either $\bot$ or a pair $\langle \pi, E \rangle$ with E a graph in split normal form, and π a valid potential function for E. We must now modify each abstract operation to deal with graphs in split normal form. We refer to the resulting abstract interpretation as "SNF Zones".

5.1 Ordering

With split normal form, the entailment check is slightly more complex. 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 $O(|E_2|)$ time.

5.2 Variable Elimination

Variable elimination is exactly as in Section 3—we just discard all edges incident on variable to be eliminated. Note that this operation preserves split normal form.

5.3 Constraint Addition, Assignment, and Meet

Similarly, the modifications for constraint addition, assignment, and meet are relatively straightforward. The construction of the initial (non-normalized) result and computation of potential function are performed exactly as in Section 3. The difference is that now, when we restore tr-closure, we do so only for $E \setminus \{v_0\}$. To maintain the sparse split normal form with a minimal amount of work, most operations will work incrementally, identifying small sets δ of binary constraints to add or update. However, newly discovered binary constraints δ may have consequences for variable bounds. Since our aim is to also keep variable bounds explicit at all times, we need to find bounds relations that

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

update-bounds$_Z(E, \delta)$
$bound_d := \left\{ v_0 \xrightarrow{k_0+k} d \;\middle|\; v_0 \xrightarrow{k_0} s \in E, s \xrightarrow{k} d \in \delta, k_0 + k < wt_E(v_0, d) \right\}$
$bound_s := \left\{ s \xrightarrow{k_0+k} v_0 \;\middle|\; s \xrightarrow{k} d \in \delta, d \xrightarrow{k_0} v_0 \in E, k_0 + k \leq wt_E(s, v_0) \right\}$
return $\delta \cup bound_d \cup bound_s$

Fig. 16. Extending a set δ of binary constraints to make any implied bounds constraints explicit.

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

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

meet$_Z(\langle \pi_1, E_1 \rangle, \langle \pi_2, E_2 \rangle)$
  $E := E_1 \otimes E_2$
  $\pi :=$ compute-potential(E, π_1)
  if (π = `inconsistent`)
    return $\bot$
  $\delta_D := (E_1 \boxminus E_2) \setminus \{v_0\}$
  $\delta_C :=$ close-meet$(\pi, E \setminus \{v_0\}, E_1, E_2)$
  return $\langle \pi, E \otimes$ update-bounds$_Z(E, \delta_D \cup \delta_C) \rangle$

Fig. 17. Constraint addition, assignment and meet for split normal graphs. The function close-edge was defined in Figure 4, restore-potential in Figure 5(a), close-assignment and eval-expr in Figure 7, compute-potential in Figure 10, and close-meet in Figure 11.

come about as a result of adding binary relations to a graph. The helper function update-bounds$_Z$, shown in Figure 16 does this. It ensures that bounds information is maintained when a set δ of binary constraints are added to E. It will be used in other operations to extend δ with bounds constraints that are consequences of $\delta \cup E$.

Pseudo-code for constraint addition, variable assignment and meet is given in Figure 17.

Only the case of meet has become more complicated. As discussed in Section 3.5, $E_1 \otimes E_2$ does not necessarily provide a closed form of the meet, but a subsequent call to close-meet will identify any binary constraints that need to be added. In turn, such binary constraints may entail new bounds constraints, when combined with existing bounds constraints. The same goes for binary edges present in E_1 but not in E_2 (and vice versa)—when combined with current bounds constraints,

they may lead to new bounds being discovered. Hence we collate all these new binary constraints and call the function update-bounds$_Z$ to add all entailed bounds.

Example 5.2. Consider the constraint sets $E_1 = \{0 \le x, y \le z\}$ and $E_2 = \{x \le y\}$. In this case, the pointwise minimum and the symmetric difference coincide: $E_1 \otimes E_2 = E_1 \boxminus E_2 = \{0 \le x, x \le y, y \le z\}$. The "binary relational" part δ_D of $E_1 \boxminus E_2$ is $\{x \le y, y \le z\}$. In this example, coincidentally, that same set is passed to close-meet, which returns the set $\{x \le z\}$ as the missing binary consequences. All of these added binary consequences ($\{x \le y, y \le z, x \le z\}$) must now be passed to update-bounds$_Z$, so that any entailed bounds constraints can be made explicit: update-bounds$_Z$ returns $\{0 \le y, 0 \le z\}$ as newly discovered variable bounds. □

5.4 Join

In the context of split normal graphs, the computation of $E_1 \sqcup E_2$ becomes more intricate. As in Section 3.6, either potential function may be retained, and edges $v_0 \xrightarrow{k} x$ and $x \xrightarrow{k} v_0$ need no special handling. However, we can no longer simply take pointwise maxima; direct application of the join described in Section 3 may lose precision.

Example 5.3. Consider the join point at program line 4 in Figure 15. In split normal form, the abstract states are:

$$E_1 = \{x \xrightarrow{-1} v_0, v_0 \xrightarrow{1} x, y \xrightarrow{-2} v_0, v_0 \xrightarrow{2} y, \ldots\}$$

$$E_2 = \{x \xrightarrow{-2} v_0, v_0 \xrightarrow{2} x, y \xrightarrow{-3} v_0, v_0 \xrightarrow{3} y, \ldots\}$$

Here E_1 entails $y - x = 1$ through the path $x \xrightarrow{-1} v_0, v_0 \xrightarrow{2} y$; and E_2 entails $y - x = 1$ through $y \xrightarrow{-2} v_0, v_0 \xrightarrow{1} x$. Hence $E_1 \sqcup E_2$ should entail $y - x = 1$.

If we apply the join from Section 3, we obtain:

$$E = \{x \xrightarrow{-1} v_0, v_0 \xrightarrow{2} x, y \xrightarrow{-2} v_0, v_0 \xrightarrow{3} y, \ldots\}$$

which only supports the weaker $1 \le y - x \le 2$. □

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

(a) $x \xrightarrow{k} y \in E_1$ and $x \xrightarrow{k'} y \in E_2$

(b) $\{x \xrightarrow{k'} v_0, v_0 \xrightarrow{k''} y\} \subseteq E_1$ and $x \xrightarrow{k} y \in E_2$ (or the converse)

(c) $\{x \xrightarrow{k_1} v_0, v_0 \xrightarrow{k_2} y\} \subseteq E_1$ and $\{x \xrightarrow{k'_1} v_0, v_0 \xrightarrow{k'_2} y\} \subseteq E_2$, where

$$\mathbf{max}(k_1 + k_2, k'_1 + k'_2) < \mathbf{max}(k_1, k'_1) + \mathbf{max}(k_2, k'_2) \tag{2}$$

The join operation presented in Section 3 collects only those binary relations which are explicit in both E_1 and E_2, that is, case (a). We can find relations of form (b) by walking through E_2, and collecting any edges which are implicit in E_1 (an example is provided later in this section). Case (c) is the one illustrated in Example 5.3, where some invariant is implicit in both operands, but is no longer maintained in the result. We only need to consider the cases where a new (or stronger) binary constraint can be added. The condition for this is given by (2): The right-hand side is the (implicit) weight we will get for $x \rightarrow y$ once we apply $\oplus$ to E_1 and E_2. But a stronger constraint (that is, a smaller weight) may be implied by both of E_1 and E_2 already: $k = \mathbf{max}(k_1 + k_2, k'_1 + k'_2)$ may be smaller than $\mathbf{max}(k_1, k'_1) + \mathbf{max}(k_2, k'_2)$. If that is the case then k is the correct weight to use in the result.

$\mathsf{join}_Z(\langle \pi_1, E_1\rangle, \langle \pi_2, E_2\rangle)$
  $E_1' := E_1 \otimes \mathsf{split\text{-}rels}_Z(E_1, E_2)$
  $E_2' := E_2 \otimes \mathsf{split\text{-}rels}_Z(E_2, E_1)$
  $src_+ := \left\{(x, wt_{E_1}(x, v_0), wt_{E_2}(x, v_0)) \;\middle|\; x \in V \wedge (wt_{E_1}(x, v_0) \neq \infty) \wedge (wt_{E_1}(x, v_0) > wt_{E_2}(x, v_0))\right\}$
  $src_- := \left\{(x, wt_{E_1}(x, v_0), wt_{E_2}(x, v_0)) \;\middle|\; x \in V \wedge (wt_{E_2}(x, v_0) \neq \infty) \wedge (wt_{E_1}(x, v_0) < wt_{E_2}(x, v_0))\right\}$
  $dest_+ := \left\{(y, wt_{E_1}(v_0, y), wt_{E_2}(v_0, y)) \;\middle|\; y \in V \wedge (wt_{E_1}(v_0, y) \neq \infty) \wedge (wt_{E_1}(v_0, y) > wt_{E_2}(v_0, y))\right\}$
  $dest_- := \left\{(y, wt_{E_1}(v_0, y), wt_{E_2}(v_0, y)) \;\middle|\; y \in V \wedge (wt_{E_2}(v_0, y) \neq \infty) \wedge (wt_{E_1}(v_0, y) < wt_{E_2}(v_0, y))\right\}$
  $E' := \mathsf{bound\text{-}rels}(src_+, dest_-) \cup \mathsf{bound\text{-}rels}(src_-, dest_+)$
  $E := E' \otimes (E_1' \oplus E_2')$
  return $\langle \pi_1, E\rangle$

$\mathsf{split\text{-}rels}_Z(E_1, E_2)$
  $E := \emptyset$
  for each $x \xrightarrow{k} y \in (E_2 \setminus \{v_0\})$
    if $(wt_{E_1}(x, v_0, y) < wt_{E_1}(x, y))$
      $E := E \cup \{x \xrightarrow{wt_{E_1}(x, v_0, y)} y\}$
  return E

$\mathsf{bound\text{-}rels}(src, dest)$
  $E := \emptyset$
  for each $(x, k_1, k_1') \in src$
    for each $(y, k_2, k_2') \in dest$
      if $(x \neq y)$
        $E := E \cup \{x \xrightarrow{\max(k_1+k_2, k_1'+k_2')} y\}$
  return E

Fig. 18. Join of abstract states in split normal form. $\mathsf{split\text{-}rels}_Z$ collects binary constraints which are implicit in E_I but explicit in E_R. bound-rels collects the binary constraints that are entailed by compatible bound changes.

The join algorithm is given in Figure 18. The first two lines of code for join_Z extend the input graphs with edges that can be derived as instances of case (b) above. The rest of the function deals with case (c). It uses the fact that the restriction (2) can only hold when

$$(wt_{E_1}(x, v_0) < wt_{E_2}(x, v_0)) \wedge (wt_{E_2}(v_0, y) < wt_{E_1}(v_0, y)) \tag{3}$$

or, alternatively, when

$$(wt_{E_1}(x, v_0) > wt_{E_2}(x, v_0)) \wedge (wt_{E_2}(v_0, y) > wt_{E_1}(v_0, y)) \tag{4}$$

Figure 19 provides the intuition why, if neither (3) nor (4) holds, there is no need to worry about implied binary constraints.

We can collect the relevant pairs of variables by placing them into buckets according to the signs of $wt_{E_1}(v_0, x) - wt_{E_2}(v_0, x)$ and $wt_{E_1}(x, v_0) - wt_{E_2}(x, v_0)$, leading to four different buckets. For example, the bucket src_+ includes a triple (v, k_1, k_2) iff E_1 has a smaller lower bound $(-k_1)$ for v than E_2 has. Relevant binary relations can only come about by combining src_+ and $dest_-$, or combining src_- and $dest_+$. The function bound-rels finds these new derived binary relations.

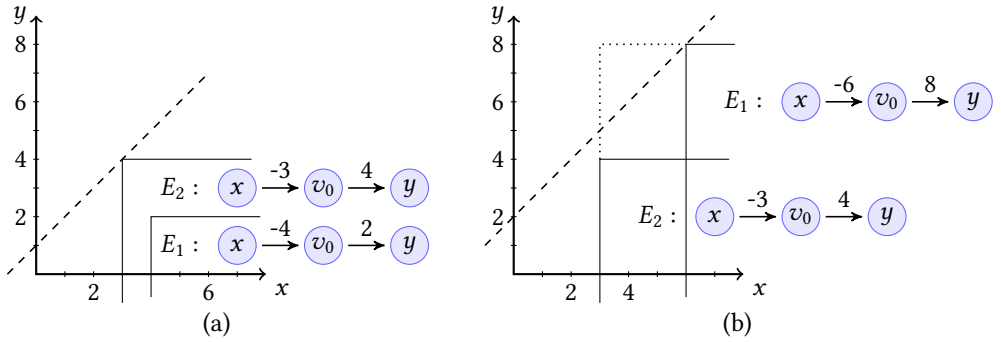


Fig. 19. A graphical illustration of the impact of bounds constraints in the join operation. (a) A case where neither (3) nor (4) holds, which means one of the two graphs subsumes the other (when projected onto the vertex set $\{v_0, x, y\}$). Implicit difference constraints (such as $y - x \leq 1$—the area below the dashed line) can remain implicit. (b) A case where (3) holds. The constraint $y - x \leq 2$ (the area below the dashed line) must be made explicit, as it is not implied by the new bounds ($x \geq 3, y \leq 8$—indicated by the dotted lines) that will be derived for the join.

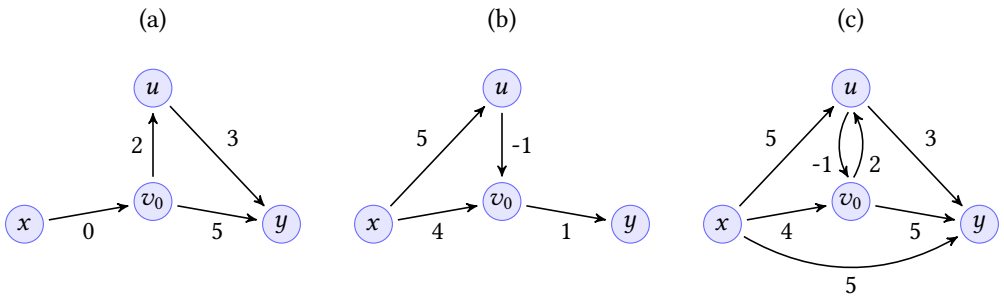


Fig. 20. (a) The constraint graph E_1 for Example 5.4. (b) The graph E_2. (c) The join of the two.

Note the order of construction of E in Figure 18. Each of the initial components E_1, E_1', E_2, E_2' and E' is in split normal form. $E_1' \oplus E_2'$ may not be, but at the point when $E_1' \oplus E_2'$ has been calculated, there is no need to normalize the result. Namely, relations in classes (a) and (b) are preserved by $\oplus$, bounds relations (that is, relations with v_0) are fully closed, and all binary constraints that flow from class (c) are captured in E', and hence will be added immediately. To see that the result is indeed tr-closed, assume $E' \otimes (E_1' \oplus E_2')$ is *not* tr-closed. Then there must be some path $x \xrightarrow{k} y \in E'$, $y \xrightarrow{k'} z \in (E_1' \oplus E_2')$ such that $x \xrightarrow{k+k'} z$ is not in either operand. But that is not possible, because it means there must be a path $x \xrightarrow{c_1} v_0, v_0 \xrightarrow{c_2} y, y \xrightarrow{c_3} z \in E_1$ such that $c_1 + c_2 \leq k$, $c_3 \leq k'$, and then there must also be some path $x \xrightarrow{c_1} v_0, v_0 \xrightarrow{c'} z \in E_1$, with $c' \leq c_2 + c_3$. The same kind of reasoning holds for E_2. Thus $x \xrightarrow{k+k'} z$ must be in $E' \otimes (E_1' \oplus E_2')$.

To illustrate the workings of the join algorithm, let us walk through two examples.

Example 5.4. Consider the constraint sets $C_1 = \{u \leq 2, x \geq 0, y - u \leq 3\}$ and $C_2 = \{u \geq 1, x \geq -4, y \leq 1, u - x \leq 5\}$. The corresponding constraint graphs E_1 and E_2 are shown in Figure 20. The graphs are in split normal form. In particular, the C_1 consequence $y \leq 5$ is explicit in E_1, as it is a bounds constraint, but other C_1 consequences such as $u - x \leq 2$ are not (as this is not required for

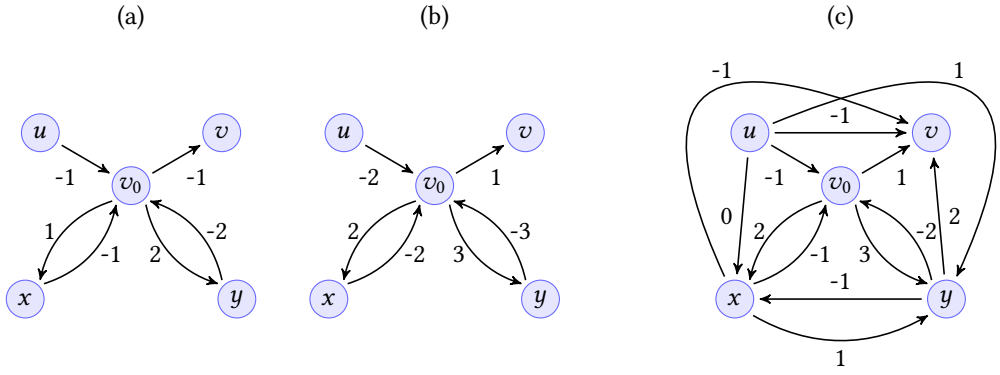


Fig. 21. (a) The constraint graph E_3 for Example 5.5. (b) The graph E_4. (c) The join of the two. Binary relations, including edges for $y - x = 1$, have been added, as these edges cannot be derived from the updated variable bounds (that is, edges to and from v_0).

split normal form). Similarly, the C_2 consequence $x \geq -4$ is explicit in E_2. Importantly, note that $y - x \leq 5$ is a (non-visible) consequence of C_1 as well as of C_2.

The role of split-rels is to extend E_1 with the edge $x \xrightarrow{2} u$ and E_2 with $u \xrightarrow{0} y$. At this point we can take the pointwise maximum of the two graphs. We also find $src_+ = dest_- = \emptyset$, $src_- = \{(x, 0, 4)\}$, and $dest_+ = \{(y, 5, 1)\}$. This identifies $x \xrightarrow{5} y$ as an edge that needs to be made explicit in the result of the join. The resulting graph is shown in Figure 20(c). Note that none of the three resulting binary relations ($u - x \leq 5$, $y - u \leq 3$, $y - x \leq 5$) is a consequence of variable bounds—all need to be explicit. □

Example 5.5. Consider the constraint sets $C_3 = \{u \geq 1, v \leq -1, x = 1, y = 2\}$ and $C_4 = \{u \geq 2, v \leq 1, x = 2, y = 3\}$. The constraint graphs E_3 and E_4 in split normal form are shown in Figure 21. In this example, both graphs have bounds relations only, so $\text{split} - \text{rels}_Z$ plays no role. The bounds constraints for the resulting graph are found by pointwise maximum, as seen in Figure 21(c). It remains to find the implicit binary relations (case (3)). We find $src_- = dest_+ = \emptyset$, $src_+ = \{(u, -1, -2), (x, -1, -2), (y, -2, -3)\}$, and $dest_- = \{(v, -1, 1), (x, 1, 2), (y, 2, 3)\}$. Hence we add the following edges: $u \xrightarrow{-1} v$, $u \xrightarrow{0} x$, $u \xrightarrow{1} y$, $x \xrightarrow{-1} v$, $x \xrightarrow{1} y$, $y \xrightarrow{-2} v$, $y \xrightarrow{-1} x$.

The result is shown in Figure 21(c). Note how the binary relation $y - x = 1$ has been preserved, in spite of being only implicit in the input graphs. □

5.5 Widening

Just as the join operation required extra care, widening for SNF Zones is a delicate operation. A well-known issue in the implementation of widening for classical Zones is the fact that a natural widening operator, which simply removes an edge whose weight changes from one iteration to the next, can interact in an undesirable way with tr-closure. Namely, an edge which is removed by widening may inadvertently be re-introduced by tr-closure, causing non-termination. Miné [45, 46] pointed this out in the original papers on Zones and Octagons, providing concrete examples of the problem. With SNF Zones, the fact that a large portion of binary relations are implicit only exacerbates the problem. Namely, in our sparse representation, there is no way of distinguishing edges which were never present (because they were always implied by bounds) from those which have deliberately been discarded through widening (leaving only a weaker implicit relation).

We therefore take a different approach to widening, using the concept of "isolated" widening [29]. The idea is to separate the termination aspect of widening from the task of finding upper bounds in the abstract domain D. In this approach, one distinguishes an "isolated" widening domain $\mathfrak{W}$, which is related to, but not necessarily the same as, D. The role of $\mathfrak{W}$ is to ensure termination; $\mathfrak{W}$ must be a poset satisfying the condition that every ascending chain in $\mathfrak{W}$ stabilises in finite time, that is, for every sequence $s_0 \leq s_1 \leq s_2 \leq \ldots$ of $\mathfrak{W}$, there is some $k \in \mathbb{N}$ such that $s_k = s_{k+1} = s_{k+2} = \ldots$. We refer to a set with this property as an *acc-poset*.

$\mathfrak{W}$ is equipped with three operations:

$$\mathbb{W} : \mathfrak{W} \times D \to \mathfrak{W} \qquad \mathit{reflect} : D \to \mathfrak{W} \qquad \mathit{reify} : \mathfrak{W} \to D$$

Here, $\mathbb{W}$ is the widening operator, and the functions *reflect* and *reify* provide the correspondence between D and $\mathfrak{W}$. The function *reflect* lifts an abstract state to initialize an ascending sequence, and *reify* maps the current iterate back onto the abstract domain, in preparation for computing the next step in the sequence.

Definition 5.6 (Isolated widening). Let $(D, \sqsubseteq)$ be an abstraction of poset $(C, \subseteq)$ given by concretisation γ. The quintuple $\langle \mathfrak{W}, \preceq, \mathit{reflect}, \mathit{reify}, \mathbb{W} \rangle$ is an isolated widening for $(D, \sqsubseteq)$ iff $(\mathfrak{W}, \preceq)$ is an acc-poset and the operators $\mathbb{W} : \mathfrak{W} \times D \to \mathfrak{W}$, $\mathit{reflect} : D \to \mathfrak{W}$, and $\mathit{reify} : \mathfrak{W} \to D$ satisfy:

$$\forall x \in D.\ (\gamma(x) \subseteq \gamma(\mathit{reify}(\mathit{reflect}(x)))) \tag{5}$$

$$\forall w \in \mathfrak{W},\ x \in D.\ (w \preceq (w \mathbb{W} x)) \tag{6}$$

$$\forall w \in \mathfrak{W},\ x \in D.\ (\gamma(x) \subseteq \gamma(\mathit{reify}(w \mathbb{W} x))) \tag{7}$$

For details, and motivation for this approach to widening, see Gange *et al.* [29]. In $w \mathbb{W} d$, the left argument (the *widener*) holds "historical" information whereas the right argument (the "widenee") holds "current" information (which may be weakened as a result of widening). Allowing the two to inhabit different (albeit closely related) structures offers certain advantages [29].

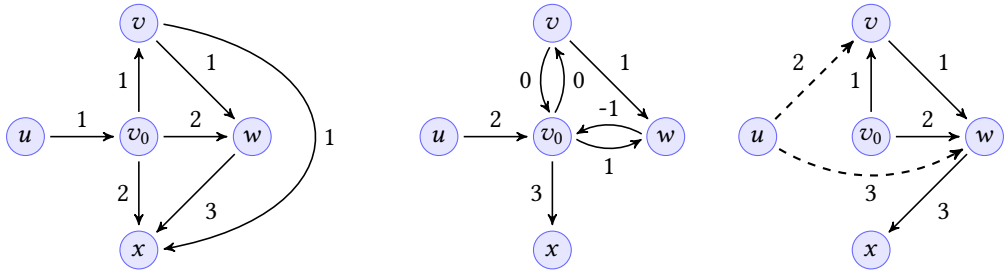
The isolated widening domain $\mathfrak{W}$ consists of a set of weighted edges E (*not* generally tr-closed). We define the ordering $\preceq$ on widener iterates as:

$$E_1 \preceq E_2 \text{ iff } E_1 \supseteq E_2 \vee \left((\mathit{incid}_{E_1}(v_0) \supset \mathit{incid}_{E_2}(v_0)) \wedge \forall\, x \xrightarrow{k} y \in E_2\, (k \geq \mathbf{min}(wt_{E_1}(x, y), wt_{E_1}(x, v_0, y))) \right)$$

This ordering allows a new binary relationship edge to be introduced by $\mathbb{W}$; but only if some bounds edge is discarded at the same time (ensuring the ascending chain property), and the relation was already implied by bounds (so increasing under $\preceq$ is also increasing under $\sqsubseteq$). We shall use this flexibility to allow $\mathbb{W}$ to deal with stable, but implicit, relationships which would otherwise be lost.

If a binary relationship implied by bounds is stable, but the bounds relations themselves no longer are stable, we introduce the binary edge—irrespective of whether a corresponding relationship has been eliminated. Importantly, this can happen only once: $x \to y$ would have been made explicit because one of $wt_E(x, v_0)$ and $wt_E(v_0, y)$ was unstable. After this bound is discarded, the relationship is no longer implied. Thus we still have a well-founded sequence: in each ascending iteration, either the number of bounds edges decreases, or the number of bounds edges is unchanged, and the number of binary edges decreases.

Example 5.7. Consider the graphs E_5 and E_6 given in Figure 22. Figure 22(c) shows the result of computing $E_5 \mathbb{W} E_6$. The algorithm proceeds by first collecting all stable relationships which are explicit in one (e.g., $w \xrightarrow{3} x$) or both (e.g., $v \xrightarrow{1} w$) operands. This yields the solid edges in Figure 22(c). Then stable-implicit collects the relations which are implied by bounds on both sides. As with the join, we examine the direction of bound changes to determine which relations may need to be introduced. In this example, $u \xrightarrow{2} v$ and $u \xrightarrow{3} w$ are implied by bounds on both sides, but would be



(a) E_5 (b) E_6 (c) $E_5 \mathbin{\triangledown} E_6$

Fig. 22. Widening: steps involved in computing $E_5 \mathbin{\triangledown} E_6$. When applied to the result, reify will add edges $v \xrightarrow{4} x$, $u \xrightarrow{6} x$, and $v_0 \xrightarrow{5} x$.

lost when $u \xrightarrow{1} v_0$ is discarded. When we map the result back onto D, *reify* will add the missing edges $v \xrightarrow{4} x$ and $u \xrightarrow{6} x$, plus the missing bound $v_0 \xrightarrow{5} x$, to restore SNF. □

Pseudo-code for isolated widening for SNF Zones is shown in Figure 23. To make *reify* faster, we augment the widener with the potential function π (to avoid re-computing it).

We can also augment $\mathfrak{W}$ to apply a short-cut here similar to the chromatic Dijkstra's algorithm. Consider again running Dijkstra's algorithm from v on $E_1 \mathbin{\triangledown} E_2$ for classical (non-SNF) zones. At some point, we reach a vertex w with $E_1^*(w) = (E_1 \bigtriangledown E_2)(w)$—that is, all outgoing edges from w are stable in E_1^*. But since $E_1 \sqsubseteq (E_1 \bigtriangledown E_2)$, we have $(E_1 \bigtriangledown E_2)(w) = E_1^*(w) \sqsubseteq (E_1 \bigtriangledown E_2)^*(w)$. Thus we do not need to further expand any children reached via w, as any such vertex is already in the priority queue, with equal or better weight.

It is also possible to use information about such stable vertices to obtain early termination for graphs in SNF. For clarity of presentation, we omit this from Figure 23; the necessary changes are similar in flavour to those for chromatic Dijkstra (tracking a set of vertices which are known to be stable).

6 FROM ZONES TO OCTAGONS

The Octagons domain [48] subsumes the Zones domain; it also allows the capturing of constraints of the form $x + y \leq k$ and $-x - y \leq k$. This added expressiveness has made it a popular abstract domain.

6.1 Octagons

Octagons [48] or, as they are known in the constraint programming community, *unit coefficient two variables per inequality* (UTVPI) constraints [35] approximate a concrete state by predicates of the forms $x \leq k$, $x \geq k$, $y - x \leq k$, $x + y \leq k$, and $-x - y \leq k$ where x and y are variables and k is some constant.

Such constraints can be mapped to difference constraints using an idea suggested by Miné [48]: Introduce two versions of each variable x, namely x^+ and x^- where x^+ represents x and x^- represents $-x$. We can then map all Octagons constraints into Zones constraints over these new variables, see Table 2. Correspondingly we have a graph representation over the variable set $V_\pm = \{v^+ \mid v \in V\} \cup \{v^- \mid v \in V\}$. The involution $o(\cdot)$ links the two variable representatives: $o(v^+) = v^-$, and $o(v^-) = v^+$. We refer to a graph over $V_\pm$ as a *Miné graph*.

$(\langle\!\langle \pi_1, E_1 \rangle\!\rangle \mathbin{\overline{\vee}} \langle \pi_2, E_2 \rangle)$:
  $E' :=$ stable-edges(E_1, E_2)
  return $\langle\!\langle \pi_1, E' \rangle\!\rangle$

reflect$(\langle \pi, E \rangle)$:
  return $\langle\!\langle \pi, E \rangle\!\rangle$

reify$(\langle\!\langle \pi, E \rangle\!\rangle)$:
  $\delta_R := \emptyset$
  for each $x \in V$:
    $\delta_R := \delta_R \cup$ Dijkstra$(\langle \pi, E \rangle, x)$
  $\delta_B :=$ close-assignment$(\langle \pi, E \cup \delta_R \rangle, v_0)$
  return $\langle E \cup \delta_R \cup \delta_B \rangle$

stable-edges(E_1, E_2):
  $E_{rel} := \left\{ x \xrightarrow{k_1} y \;\middle|\; x \xrightarrow{k_2} y \in E_2, k_1 = \mathbf{min}(wt_{E_1}(x,y), wt_{E_1}(x,v_0,y), k_2 \le k_1 < \infty \right\}$
    $\cup \left\{ x \xrightarrow{k_1} y \;\middle|\; x \xrightarrow{k_1} y \in E_1, wt_{E_2}(x,v_0,y) \le k_1 \right\}$
  $src_+ := \left\{ (x, wt_{E_1}(x,v_0), wt_{E_2}(x,v_0)) \;\middle|\; x \in V \wedge (wt_{E_1}(x,v_0) \neq \infty) \wedge (wt_{E_1}(x,v_0) > wt_{E_2}(x,v_0)) \right\}$
  $src_- := \left\{ (x, wt_{E_1}(x,v_0), wt_{E_2}(x,v_0)) \;\middle|\; x \in V \wedge (wt_{E_2}(x,v_0) \neq \infty) \wedge (wt_{E_1}(x,v_0) < wt_{E_2}(x,v_0)) \right\}$
  $dest_+ := \left\{ (y, wt_{E_1}(v_0,y), wt_{E_2}(v_0,y)) \;\middle|\; y \in V \wedge (wt_{E_1}(v_0,y) \neq \infty) \wedge (wt_{E_1}(v_0,y) > wt_{E_2}(v_0,y)) \right\}$
  $dest_- := \left\{ (y, wt_{E_1}(v_0,y), wt_{E_2}(v_0,y)) \;\middle|\; y \in V \wedge (wt_{E_2}(v_0,y) \neq \infty) \wedge (wt_{E_1}(v_0,y) < wt_{E_2}(v_0,y)) \right\}$
  $E_{bound} :=$ stable-implicit$(src_+, dest_-) \cup$ stable-implicit$(src_-, dest_+)$
  return $E_{rel} \cup E_{bound}$

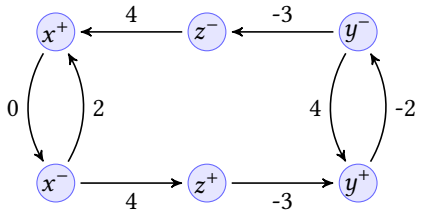
stable-implicit$(src, dest)$:
  $E := \emptyset$
  for$((s, k_1, k_1') \in src, (d, k_2, k_2') \in dest)$:
    if$(k_1 + k_2 \ge k_1' + k_2')$:
      $E := E \cup \{ s \xrightarrow{k_1+k_2} d \}$
  return E

Fig. 23. Isolated widening for split normal graphs. Function close-assignment was given in Figure 7.

Table 2. Translating Octagons constraints to Zones constraints over $V_{\pm}$.

Octagons constraint	Zone constraints
$y - x \le k$	$y^+ - x^+ \le k, x^- - y^- \le k$
$x + y \le k$	$x^+ - y^- \le k, y^+ - x^- \le k$
$-x - y \le k$	$x^- - y^+ \le k, y^- - x^+ \le k$
$x \le k$	$x^+ - x^- \le 2k$
$x \ge k$	$x^- - x^+ \le -2k$

Notice how the need for the variable v_0 disappears as we represent a lower bound of x by an edge from x^+ to x^- and an upper bound by an edge from x^- to x^+. For example, $x \ge 5$, which had the Zones expression $v_0 - x \le -5$, is now expressed as $x^- - x^+ \le -10$. Bounds information resides in edges of form $v \xrightarrow{k} o(v)$.



(a)

	x^+	x^-	y^+	y^-	z^+	z^-
x^+	0	0	∞	∞	∞	∞
x^-	2	0	∞	∞	4	∞
y^+	∞	∞	0	-2	∞	∞
y^-	∞	∞	4	0	∞	-3
z^+	∞	∞	-3	∞	0	∞
z^-	4	∞	∞	∞	∞	0

(b)

Fig. 24. (a) Miné graph representing the Octagons constraints $x \in [0,1]$, $y \in [1,2]$, $y - z \leq -3$, $x + z \leq 4$, and (b) the corresponding difference bound matrix.

Example 6.1. Consider the set of Octagons constraints $x \in [0,1]$, $y \in [1,2]$, $y - z \leq -3$, $x + z \leq 4$. The corresponding difference logic constraints are $x^+ - x^- \leq 2$, $x^- - x^+ \leq 0$, $y^+ - y^- \leq 4$, $y^- - y^+ \leq -2$, $y^+ - z^+ \leq -3$, $z^- - y^- \leq -3$, $x^+ - z^- \leq 4$, $z^+ - x^- \leq 4$. The graph representation is shown in Figure 24(a) and the (traditional) matrix representation is shown in Figure 24(b). □

It will be useful to have a name for the set of edges (in set E) that link the two representatives x^+ and x^- of variable x. We define

$$B_E = \left\{ v \xrightarrow{k} o(v) \in E \;\middle|\; v \in V_\pm \right\}$$

For use in Figure 26 we also define $dual(u \xrightarrow{k} v) = o(v) \xrightarrow{k} o(u)$.

6.2 Closures

In the graph representation, Zones required a closure operation that is nothing but an all-pairs shortest path calculation. The case of Octagons is more complicated and calls for three different closure principles [49]. The following applies to abstract states of the form $\langle \pi, E \rangle$.

First, a graph E must be kept *coherent*, that is, it must respect the intended roles of variables x^+ and x^- (for example, each constraint $x^- - y^- \leq k$ entails $y^+ - x^+ \leq k$). More precisely, E is *coherent* iff

$$wt_E(x,y) = \mathbf{min}(wt_E(x,y), wt_E(o(y),o(x))) \text{ for all } x, y \in V_\pm \tag{8}$$

Second, we have a "triangle" principle similar to (1). We say that E is *tr-closed* iff

$$wt_E(x,z) = \mathbf{min}(wt_E(x,z), wt_E(x,y) + wt_E(y,z)) \text{ for all } x, y, z \in V_\pm \tag{9}$$

It is not hard to see that tr-closure preserves coherence, that is, the tr-closure of a coherent graph is coherent.

Finally, the extraction of difference constraints that are implicit consequences of variable bounds is more cumbersome than in the Zones case. This is because bounds information no longer can be found simply by inspecting edges incident on a single node (v_0). Instead we must take into account how variable pairs v^+, v^- are semantically linked. We say that E is *strengthened* iff

$$wt_E(x,y) = \mathbf{min}\Big(wt_E(x,y), \frac{wt_E(x,o(x)) + wt_E(o(y),y)}{2}\Big) \text{ for all } x, y \in V_\pm \tag{10}$$

The function *strengthen*, defined by

$$strengthen(E) = E \otimes \left\{ x \xrightarrow{k} y \;\middle|\; k = \mathbf{min}\Big(wt_E(x,y), \frac{wt_E(x,o(x)) + wt_E(o(y),y)}{2}\Big) \right\}$$

is a lower closure operator on $(E, \sqsubseteq)$. The function *strengthen* preserves both coherence and tr-closure.

The role of strengthening of E is to make implicit binary relations explicit—those that are consequences of bounds constraints.

Example 6.2. Consider $E = \{x^+ \xrightarrow{-4} x^-, y^- \xrightarrow{6} y^+\}$. Strengthening adds the edge $x^+ \xrightarrow{1} y^+$ (and, coherently, $y^- \xrightarrow{1} x^-$). The effect is to make the binary relation $y - x \leq 1$ explicit (it is a consequence of the variable bounds $x \geq 2$ and $y \leq 3$). □

The combination of properties (8)–(10) is usually referred to as *strong* closure for Octagons [46]. Bagnara *et al.* [3] established that closure of a *coherent* Octagons DBM can be restored by a single application of the Bellman-Ford algorithm and a propagation through the graph with the strengthening step. The time taken for strengthening is linear and relatively negligible when the number of variables becomes larger. However, the time required for tr-closure in the Octagons domain is much worse than for Zones, because the number of vertices is doubled. As this closure is cubic in the number of vertices, a time penalty factor of about 8 can be expected.

7 SNF OCTAGONS

Now we adapt the ideas from Section 5 to Octagons constraints in split normal form. We refer to the resulting abstract interpretation as "SNF Octagons".

As discussed in Section 6, the graph representation of Octagons omit the v_0 vertex, instead representing bounds on a variable x as directed edges between x^+ and x^-. Again, we assume $wt_E(x, x) = 0$ for all $x \in V_\pm$, and we take the absence of an edge (x, y) to mean $wt_E(x, y) = \infty$.

A Miné graph E is in *split normal form* iff:

- $E \setminus B_E$ is coherent and tr-closed.
- For every path $v \to u \to \cdots \to o(v)$ in E, there is an edge $v \xrightarrow{k} o(v)$ such that $k \leq wt_E(v, u, \ldots, o(v))$.

This expresses the same intent as split normal form for Zones. In particular,

- the graph explicitly stores the strongest binary constraints that are derivable from other binary constraints (but not necessarily those that can be derived from variable bounds); and
- all variable bounds are explicit.

7.1 Weak Closure

The concept of tr-closure is exactly the same as for Zones. The role of the potential function is likewise unchanged, as is the way it induces a *slack graph*. However, the potential function is now a function $\pi : V_\pm \to \mathbb{Z}$, and π provides, for a satisfiable set of constraints, a model in the form of $\lambda v.\ (\pi(v^+) + \pi(v^-))/2$ (if an integer solution is required, rounding will provide such a solution). Hence many of the functions we introduced for SNF Zones in Section 3, including close-edge, close-assignment, close-meet, and close-widen, can be reused for SNF Octagons.

Our task now is to define abstract operations for Octagons in split normal form. Loosely, for each abstract operation Op assumed to work on strongly closed graphs, we need to design a corresponding abstract operation Op', such that for each graph E in split normal form, $Op(\mathit{strengthen}(E)) = \mathit{strengthen}(Op'(E))$.

Again, we want operations to act incrementally, utilizing the split normal form. The helper function update-bounds$_O$, shown in Figure 25, is similar to the one for SNF Zones. It ensures that bounds information is not lost when the set δ of binary edges gets added to E. It is used in other operations to extend δ with bounds constraints that are consequences of $\delta \cup E$.

$$\begin{array}{l}\textsf{update-bounds}_O(E,\delta)\\ \quad bound_d := \left\{o(d) \xrightarrow{k_e+k} d \;\middle|\; o(d) \xrightarrow{k_e} s \in E \wedge s \xrightarrow{k} d \in \delta \wedge k_e + k < wt_E(o(d), d)\right\}\\ \quad bound_s := \left\{s \xrightarrow{k_e+k} o(s) \;\middle|\; d \xrightarrow{k_e} o(s) \in E \wedge s \xrightarrow{k} d \in \delta \wedge k_e + k < wt_E(s, o(s))\right\}\\ \quad \textbf{return } \delta \cup bound_d \cup bound_s\end{array}$$

Fig. 25. Updating variable bounds for SNF Octagons, cf. Figure 16.

$$\begin{array}{l}\textsf{add-edge}_O\ (\langle \pi, E\rangle, e)\\ \quad E_{add} := E \cup \{e, dual(e)\}\\ \quad \pi' := \textsf{restore-potential}(\pi, e, E_{add})\\ \quad \textbf{if } (\pi = \texttt{inconsistent})\\ \quad\quad \textbf{return } \bot\\ \quad \delta := \textsf{close-edge}(e, E_{add})\\ \quad \textbf{return} \langle \pi', E_{add} \otimes \textsf{update-bounds}_O(E, \delta)\rangle\end{array}$$

Fig. 26. Adding an edge to an Octagons graph. The function close-edge was defined in Figure 4 and restore-potential was defined in Figure 5.

7.2 Ordering

Entailment checks are implemented as for SNF Zones, *mutatis mutandis*. We have $E_1 \sqsubseteq E_2$ iff, for every $x \xrightarrow{k} y \in E_2$:

$$\mathbf{min}\big(wt_{E_1}(x, y), \frac{wt_{E_1}(x, o(x)) + wt_{E_1}(o(y), y)}{2}\big) \leq k$$

7.3 Variable Elimination

To eliminate a variable $v \in V$ from graph E, we simply remove the two nodes v^+ and v^- and all edges incident on these two nodes. As is the case for SNF Zones, vertex removal preserves split normal form, so no further closure is required.

7.4 Constraint Addition

Adding an Octagons constraint follows the same sequence as in Section 5: construct an updated potential function, then restore split normal form by closing over the new edges. The only change required is, when adding $u \xrightarrow{k} v$, we must preserve coherence by also adding the dual edge $o(v) \xrightarrow{k} o(u)$. Figure 26 gives the algorithm to add an edge to a graph and restore closure. The functions close-edge and restore-potential from Section 3.3 can be reused.

7.5 Assignment

The abstract operation for an assignment statement is analogous to that for SNF Zones. The pseudo-code is shown in Figure 27.

7.6 Meet

The meet operator identifies information that is shared by two abstract states. In classical implementations of Octagons, this can be done simply by taking the point-wise minimum of weights. We, however, must be more careful, just as in the definition of meet_Z. The algorithm for SNF Octagons meet is shown in Figure 28. It takes the point-wise minimum of two graphs, computes a valid

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

Fig. 27. Abstract assignment in the SNF Octagons case. The functions eval-expr and close-assignment were defined in Figure 7.

```
meet_O(⟨π_1, E_1⟩, ⟨π_2, E_2⟩)
    E := E_1 ⊗ E_2
    π := compute-potential(E, π_1)
    if (π = inconsistent)
        return ⊥
    δ_D := (E_1 ⊟ E_2) \ B_E
    δ_C := close-meet(π, E \ B_E, E_1, E_2)
    return ⟨π, E ⊗ update-bounds_O(E, δ_D ∪ δ_C)⟩
```

Fig. 28. The meet of abstract states in split normal form for Octagons. The function compute-potential was defined in Figure 10, close-meet in Figure 11.

potential, and finally restores closure, mirroring the definition of meet_Z. The compute-potential function is the same Bellman-Ford variant used for Zones (Figure 10). It is instructive to compare the definitions of meet_Z and meet_O.

7.7 Join

As is the case for SNF Zones, we must take care not to lose any implicit relational properties when performing a join operation. A case analysis of the sources of implied binary constraints that must be made explicit runs exactly as in Section 5.4. To discover relevant binary constraints that are entailed by bounds constraints, we proceed as for SNF Zones. The variable pairs can be collected by sorting the variables into groups by $sign(wt_{E_1}(x^+, x^-) - wt_{E_2}(x^+, x^-)$ and $sign(wt_{E_1}(y^-, y^+) - wt_{E_2}(y^-, y^+)$. Owing to coherence, we need only consider edges $x^+ \rightarrow x^-$ and $y^- \rightarrow y^+$ in the search for edges $x \rightarrow y$ to add. The resulting join algorithm for Octagons in split normal form is shown in Figure 29. Note that since both π_1 and π_2 are valid potential functions, we can choose either for the result.

Example 7.1. Consider the constraint sets $C_5 = \{x \geq 4, y \geq 1\}$ and $C_6 = \{x + y \geq 7\}$. The corresponding graphs in split normal form are shown in Figure 30. We find $\text{split-rels}_O(E_1, E_2) = \{x^+ \xrightarrow{-5} y^-, y^+ \xrightarrow{-5} x^-\}$, whereas $\text{split-rels}_O(E_2, E_1) = \emptyset$. Also, in this case, $src_+ = src_- = \emptyset$, and so $E' = \emptyset$. The join in this case is simply

$$\{x^+ \xrightarrow{-8} x^-, y^+ \xrightarrow{-2} y^-, x^+ \xrightarrow{-5} y^-, y^+ \xrightarrow{-5} x^-\} \oplus \{x^+ \xrightarrow{-7} y^-, y^+ \xrightarrow{-7} x^-\}$$

the result if which is shown in Figure 30(c). □

Example 7.2. Consider the constraint sets $C_7 = \{x \geq 1, y = 1, z = 2\}$ and $C_8 = \{x \leq 1, y = 2, z = 3\}$. The corresponding graphs in split normal form are shown in Figure 31. In this example, split-rels finds no new edges of interest. The join algorithm calculates $src_+ = \{(y^+, -1, -2), (z^+, -2, -3)\}$

$\text{join}_O(\langle \pi_1, E_1 \rangle, \langle \pi_2, E_2 \rangle)$
  $E_1' := E_1 \otimes \text{split-rels}_O(E_1, E_2)$
  $E_2' := E_2 \otimes \text{split-rels}_O(E_2, E_1)$
  $src_+ := \left\{ (x, \frac{k_1}{2}, \frac{k_2}{2}) \;\middle|\; x \in V_\pm \wedge (k_1 = wt_{E_1}(x, o(x)) \neq \infty) \wedge (k_2 = wt_{E_2}(x, o(x))) \wedge (k_1 > k_2) \right\}$
  $src_- := \left\{ (x, \frac{k_1}{2}, \frac{k_2}{2}) \;\middle|\; x \in V_\pm \wedge (k_1 = wt_{E_1}(x, o(x))) \wedge (k_2 = wt_{E_2}(x, o(x)) \neq \infty) \wedge (k_1 < k_2) \right\}$
  $E' := \text{bound-rels}\big(src_+, \{(o(x), k, k') \mid (x, k, k') \in src_-\}\big) \cup$
    $\text{bound-rels}\big(src_-, \{(o(x), k, k') \mid (x, k, k') \in src_+\}\big)$
  $E := E' \otimes (E_1' \oplus E_2')$
  return $\langle \pi_1, E \rangle$

$\text{split-rels}_O\ (E_1, E_2)$
  $E := \emptyset$
  for each $x \xrightarrow{k} y \in E_2 \setminus B_E$
    $k' := \frac{wt_{E_1}(x, o(x)) + wt_{E_1}(o(y), y)}{2}$
    if $(k' < wt_{E_1}(x, y))$
      $E := E \cup \{x \xrightarrow{k'} y\}$
  return E

Fig. 29. Join of abstract states in split normal form for Octagons. bound-rels was defined in Figure 18.

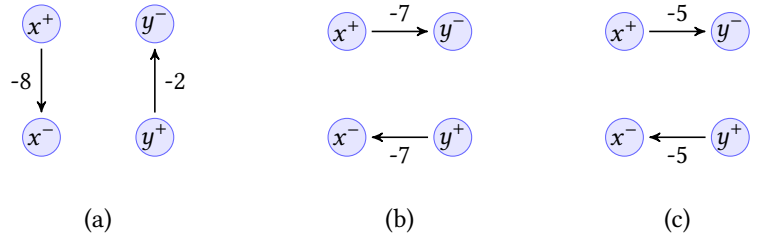


Fig. 30. (a) The graph for C_5 from Example 7.1, in split normal form. (b) The graph for C_6. (c) Their join.

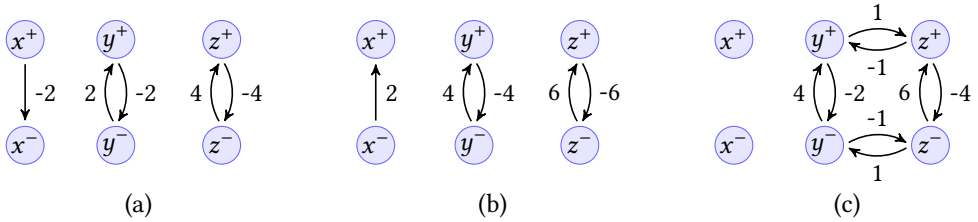


Fig. 31. (a) Split normal form constraint graph for C_7 in Example 7.2. (b) The graph for C_8. (c) Their join.

and $src_- = \{(y^-, 1, 2), (z^-, 2, 3)\}$. From this, bounds-rel deduces the four edges $y^+ \xrightarrow{1} z^+$, $z^+ \xrightarrow{-1} y^+$, $y^- \xrightarrow{-1} z^-$, and $z^- \xrightarrow{1} y^1$ (preserving coherence). Thus the graph E' in this example captures the constraint $z - y = 1$ which was shared, but kept implicit, by the two input graphs. (Other constraints that are implicit in the input graphs, such as $y \leq x$ in E_1 and $x \leq y - 1$ in E_2, will simply fail to remain consequences in the resulting join.) The graph for the join is shown in Figure 31(c). □

```
compute-potential-integer(E, π)
    π := compute-potential(E, π)
    E′ := ∅
    for each x --y--> k ∈ E
        if (π(x) + k − π(y) = 0)
            E′ := E′ ∪ {x --0--> y}
    for each x ∈ V
        if (x⁺ and x⁻ are in the same SCC of E′)
            if (π(x⁺) − π(x⁻) is odd)
                return inconsistent
    return π
```

Fig. 32. Additional checks to ensure integer satisfiability of Octagons constraints.

7.8 Widening

The use of isolated widening carries over to Octagons, and the algorithms from Figure 23 can be adapted, in the exact same manner as the join operation from Zones was adapted to the Octagons case. We omit the details.

7.9 Handling Integer Constraints

Using Zones for integer variables is straightforward. If all constants arising in difference constrains are integer that will only generate integer bounds which are optimal. So for the Zones abstract domain it does not matter whether the variables involved are supposed to range over integers, rational numbers, or reals. For Octagons, the case is different, because so-called tightening is required: The strong closure of Octagons over integers also requires that the Miné graph is *tightened*, that is, it satisfies (11) below.

$$\forall v \in V_{\pm}, wt_E(v, o(v)) = 2 \times \lfloor \frac{wt_E(v, o(v))}{2} \rfloor \tag{11}$$

Tightening is required because the edge $x^- \xrightarrow{k} x^+$ corresponds to $x^+ - x^- \leq k$, or $2x \leq k$ and hence $x \leq \lfloor k/2 \rfloor$. Hence if k is odd, the bound in x needs to be rounded down. The same thing applies with edges $x^+ \xrightarrow{k} x^-$.

To adjust Octagons for integers we need to make two adjustments. First we need to extend the compute-potential function to also enforce integer, rather than just real, satisfiability. This is based on a method devised by Lahiri and Musuvathi [41], which simply checks that any zero length cycles in the Octagons graph do not force x^+ and x^- to be an odd distance apart, effectively forcing x to take a non-integer value $\frac{\pi(x^+)+\pi(x^-)}{2}$. The pseudo-code giving the extended compute-potential for integers is shown in Figure 32.

We also need to ensure that new bounds are tightened. The only way new non-tightened bounds can arise is by tr-closure of binary edges, not bounds edges. In the meet algorithm of Figure 28, δ_b contains all such new edges. Thankfully we can simply tighten these bounds individually since, by a result by Schutt and Stuckey ([54] Lemma 6), we know that any consequent bound tightening will have been created by the tr-closure. The pseudo-code for this is shown in Figure 33.

tighten(E)
return $\left\{ v \xrightarrow{2\lceil \frac{k}{2} \rceil} o(v) \;\middle|\; v \xrightarrow{k} o(v) \in B_E \right\} \cup (E \setminus B_E)$

Fig. 33. Tightening bounds for Octagons constraints.

8 EXPERIMENTS AND RESULTS

In this section, we provide a report on the experiments we have conducted to evaluate the relative speed and precision of Zone and Octagon implementations. At a first glance, given implementations that are sound, performing this kind of comparison is a straightforward matter. If different implementations provide identical functionality for the abstract operations (say, the theoretically optimal precision) then the comparison should in principle be about speed only. However, in practice there are confounding factors at play:

(1) It is necessary, in large-scale experiments, to impose restrictions on the use of resources—we need to allow for "time out" or "memory out" as possible outcomes for a given problem instance. As a result, speed will sometimes determine precision.
(2) Widening is an abstract operation that, by its nature, does not come in an optimal, or even "natural" version. Different implementations will make different widening decisions, with different speed/precision consequences. In the context of experimental resource restrictions, there is not even a simple tradeoff at play, between speed and precision. For example, a relatively slow implementation that readily surrenders precision, either by resorting to widening earlier, or by making larger (less precise) widening steps, may well end up delivering the more precise result, simply because an alternative implementation, albeit generally faster, ends up exhausting its resources while chasing the more precise result. Note that an analysis that is implemented as Kleene iteration is not an "anytime" analysis; stopped prematurely, the analysis will not generally have arrived at a correct result, and hence it has to report "don't know" whenever it exhausts its resources.
(3) "Precision" in itself is rarely the goal. Program analysis is usually performed for a purpose, and it is the *application* that determines whether "more precise" will translate to a better outcome. Following Singh *et al.* [58], we evaluate precision in the context of program verification.

8.1 Research Questions

We aim to answer three questions:

RQ1: Is Split Normal Form effective at exploiting sparsity?
RQ2: How does Split Normal Form widening affect precision in practice?
RQ3: How does Variable Packing interact with Split Normal Form?

The first two questions illuminate the relative advantages offered by different data structures and algorithms, as used in two well understood program analyses. Relatively simple controlled experiments can provide answers. The third question investigates an alternate approach to tackling (phantom) density, namely so-called variable packing.

8.2 Implementation

SNF Zones and Octagons have been implemented in Crab [21], a parametric framework for modular construction of abstract interpreters. Crab provides both intra- and inter-procedural analyses with a number of numerical abstract domains, including the ELINA domains. Both Zones and Octagons support several graph representations: adjacency hash-table, adjacency Patricia trees, and a hybrid

representation using dense matrices for weights and sparse sets. For Zones, we have only used the hybrid representation, since it is the most efficient, as reported in earlier work [28]. For Octagons, however, the hybrid representation sometimes consumes too much memory. Hence we also evaluate the Octagons implementation using Patricia Tries to represent the adjacency lists of the SNF graphs.

8.3 Experiment Design

8.3.1 Choice of Baseline Abstract Domains. For Octagons, the 2015 paper by Singh *et al.* [58] established the "OptOctagon" implementation (now part of the ELINA library) as the fastest available at the time. An experimental evaluation against the standard Apron implementation showed significant, sometimes order-of-magnitude speedup (see Section 9.5). While OptOctagon was not directly compared against PPL [4], the authors pointed out that PPL and Apron use very similar data structures and algorithms. As part of a subsequent 2018 paper by Singh and colleagues [59], a Zones implementation was added to the ELINA library. As these ELINA analyses [23] remain the state of the art, we compare our Zones and Octagons implementations against them. The ELINA implementations that are available on the Crab platform [21] allow for a meaningful comparison.

Variable packing, that is, grouping variables into packs and only inferring relationships between variables in the same pack, has long been a favoured approach to speeding up Zones and Octagons analyses. Therefore, we also present a comparison to variable packing here, using the most precise variant of variable packing [61]. As Venet and Brat [61], we compute the packs dynamically, without a size limit. A union-find data structure is used to maintain equivalence classes of variables, based on discovered data dependencies. Lattice operations such as join, meet, narrowing and widening may merge equivalence classes. All the transfer functions can be implemented component-wise (that is, separately on each equivalence class) after the equivalences classes have been updated accordingly. For instance, after an operation $x := f(y, z)$, all three variables x, y, and z are in the same equivalence class. Note that, in the worst case, all program variables could be merged into a single equivalence class, which would be the same as not using packs. The implementation of the variable packing domain is not part of Crab, but we have made it available at https://zenodo.org/record/4740814#.YJQiOeZOmCM.

8.3.2 Choice of Benchmarks. The current state of practice of benchmarking Zones, Octagons and allied analyses is somewhat patchy, with no large commonly used set of benchmarks being available. Much of the literature uses benchmark sets sourced from flight or space applications, and these benchmark sets are not publicly available. We make use of the limited supply of programs with reliable assertions that we find publicly available.

Since we conduct comparisons with the state-of-the-art ELINA implementations, we begin by utilising the benchmark set used in ELINA publications: the 2758 programs from SV-COMP, except we use the SV-COMP 2019 [60] version. We have chosen the categories that primarily rely on numerical reasoning: ControlFlowInteger, Arrays, Loops, and DeviceDrivers64. We only consider programs expected to be safe since Crab cannot prove a program is definitely unsafe, and since, for each unsafe program in the benchmark suite, there is already a safe version.

The SV-COMP suites do not include benchmarks with large numbers of variables, and they tend to involve many loops. This skews experiments somewhat, making much use of widening, which generates sparse DBMs. Moreover, the programs were selected for the SV-COMP competition to challenge path-sensitiveness and pointer reasoning capabilities of the verification tools. In our evaluation we soon found that they do not usually require invariants involving difference constraints, and even more rarely do they require binary octagon constraints. In fact, a simple interval analysis produced similar results to Zones and Octagons. Hence, while they test fixed-point

finding prowess, the SV-COMP benchmarks alone do not provide a serious stress test for Zones and Octagons analysis.

To mitigate the limitations of the SV-COMP benchmarks and to explore how well Split Normal Form performs on programs that produce larger and denser DBMs, and cases where binary constraints are crucial for successful verification, we include a second benchmark set: all of the 260 publicly available eBPF sample programs [22]. An eBPF (extended Berkeley Packet Filter) program uses a subset of C to implement Linux kernel extensions. These programs are all free of loops and functions. Moreover, they can only access a fixed set of memory regions, known at compile time, making eBPF programs very amenable to abstract interpretation. These programs are still challenging, as an analysis must track binary relations between program registers, and it must reason precisely about memory contents. Each memory location is mapped to a dimension in the numerical domain, and therefore, the underlying numerical domain must represent a large number of dimensions. That makes analysis challenging with both Zones and Octagons. All eBPF programs are annotated with assertions to check for memory safety and information flow security [30].

8.3.3 C and eBPF Static Analyzers. Crab does not directly analyze C or eBPF programs. Instead, Crab analyzes programs represented in its own intermediate representation (CrabIR) which are more amenable to static analysis.

For analysis of SV-COMP programs (written in C), we utilize Clam [13] which translates from LLVM bitcode to CrabIR. Clam runs a pointer analysis [40] to statically partition the heap into memory regions that can be translated to uni-dimensional arrays supported by the Crab language. We choose the array smashing domain [8] parameterized by a reduced product of a Boolean and a numerical domain: (for example, Zones or Octagons). All functions are aggressively inlined.

For the analysis of eBPF programs, we use the tool PREVAIL [52] to translate eBPF programs to CrabIR. PREVAIL uses a precise Crab memory domain [30] parameterized by a Crab numerical relational domain. The memory domain is used to model each program memory region and it maps each memory region content to a dimension in the relational numerical domain. The memory domain keeps track of which memory cells might be affected by a memory write and whether the write can be modelled as a "strong update" or not.

8.3.4 Reproducibility. Experiments have only involved publicly available benchmark suites. Every experiment has been carried out on a 2.1GHz AMD Opteron processor 6172 with 32 cores and 64GB on a Ubuntu 18.04 Linux machine. From those 32 cores, we used 16 cores to run multiple instances of Crab in parallel, but each instance was executed sequentially. We have compared four DBM-based implementations: our Zones and Octagons in Split Normal Form (SNF Zones and SNF Octagons) and the Zones and Octagons provided by ELINA (ELINA Zones [59] and ELINA Octagons [58])[11]. We use SNF Octagons and SNF Octagons (PT) to refer to Octagons in Split Normal Form using the hybrid representation for adjacency lists, and Patricia Tries, respectively. For SNF Zones we only use the hybrid representation.

8.4 Results

In this section and the next, we present the experimental results as they pertain to each research question.

8.4.1 ***RQ1: Is Split Normal Form effective at exploiting sparsity?*** Figure 34 shows the efficiency of all the domains on the set of SV-COMP programs. From the initial 2758 programs, we removed those on which the Clam front-end reached timeout or memory limits of 8GB, ending up

[11] Available at https://github.com/eth-sri/ELINA, commit `6f5928694c1a2f16c36769bbf161c356648628eb`. Accessed on December 9th, 2020.

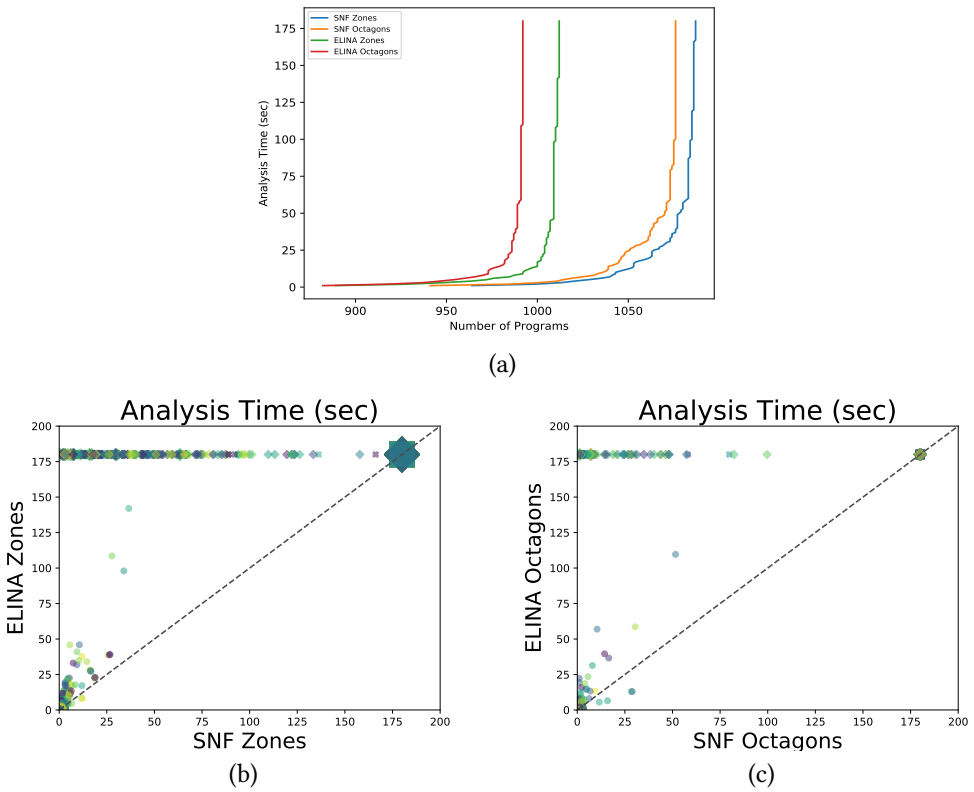


Fig. 34. Experiments on SV-COMP programs with timeout of 180 seconds and memory limit of 8GB. The marker ● represents domains finished before exhausting resources, ✖ represents timeout, and ◆ represents memory-out. The size of a marker reflects the number of scatter points at that location.

with 2549 programs. We tried two different timeouts of 3 and 5 minutes. The plots obtained were remarkably similar, suggesting that our results are not sensitive to the choice of timeout limit. The results shown are for a timeout of 3 minutes. At the top (a), a cactus plot compares the analysis time (in seconds) of the domains. Below that, scatter plots compare analysis time (in seconds) of (b) SNF Zones against ELINA Zones, and (c) SNF Octagons against ELINA Octagons.

Figure 35 shows the efficiency of all the domains on the set of eBPF programs. For these programs, timeouts were not needed to achieve termination in a reasonable time. At the top (a), a cactus plot compares analysis time (in seconds) of the domains on a logarithmic scale. Below that are scatter plots comparing analysis time (in seconds) of (b) SNF Zones against ELINA Zones, and (c) SNF Octagons against ELINA Octagons.

The plots in Figure 34(b-c) show that maintaining sparsity is crucial: Each domain in Split Normal Form is significantly faster than the corresponding ELINA domain. Similar conclusion can be drawn from plots in Figure 35(b-c), even if DBMs are much more dense due to the lack of widening. The implementations of the domains that use Split Normal Form are significantly faster than the corresponding ELINA implementations (approximately one order of magnitude).

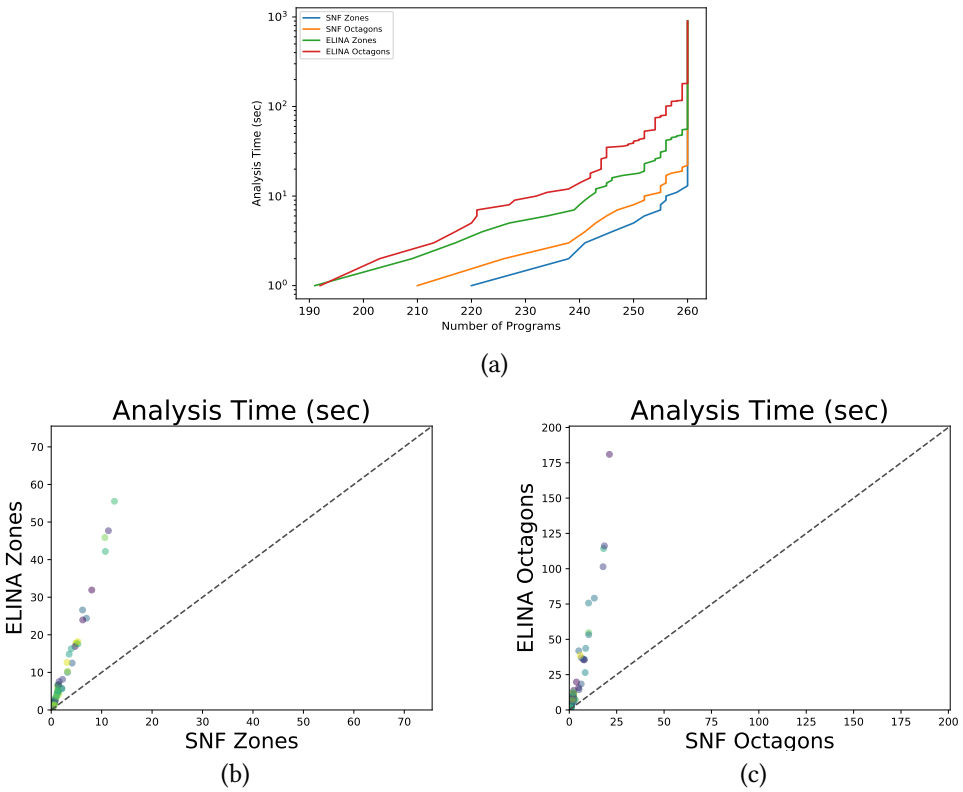


Fig. 35. Experiments on eBPF programs.

8.4.2 ***RQ2: How does Split Normal Form widening affect precision in practice?*** To answer this question, we focus on SV-COMP benchmarks, since eBPF programs are free of loops. In principle, if unlimited resources were available, we should not expect to see large differences between different implementations, assuming transfer functions are more or less identical. However, as discussed, resource limits can interfere with results, and widening can be implemented differently. Moreover, some abstract operations (assignment, for example) are less than straightforward and could conceivably be done differently in different implementations.

Table 3 compares the precision of SNF Zones, SNF Octagons, ELINA Zones, and ELINA Octagons for the SV-COMP test suite. Column Prog is the total number of programs after filtering out front-end timeouts and crashes. Columns TO and MO are the number of timeouts and memory-outs of the analyses, respectively. Proven Safe is the number of programs proven safe, Inconclusive is the number of programs an analysis cannot prove safe, Assertions is the total number of assertions checked by the analysis (if the analysis finished successfully), and Proven Assertions is the total number of proven assertions.

Table 4 uses the same data, but it shows the pairwise comparisons of SNF Zones against ELINA Zones and SNF Octagons against ELINA Octagons analyses. For each pair, we restrict our attention to the programs that were analyzed by both analyses without exhausting the time and memory limits.

Table 3. Comparing the precision of SNF Zones, SNF Octagons, ELINA Zones, and ELINA Octagons on SV-COMP programs with timeout of 180 seconds and memory limit of 8GB.

Domain	Prog	Errors		Proven Safe	Inconclusive	Assertions	Proven Assertions
		TO	MO				
SNF Zones	2549	169	353	**1622**	405	17355	**15972**
SNF Octagons	2549	163	448	1553	385	11605	10522
ELINA Zones	2549	166	751	1290	342	2423	1899
ELINA Octagons	2549	219	739	1264	327	2146	1659

Table 4. Pairwise comparisons for precision of SNF Zones against ELINA Zones and SNF Octagons against ELINA Octagons on SV-COMP programs with timeout of 180 seconds and memory limit of 8GB but considering only programs for which both domains of each pair terminated before resources are exhausted.

Domain	Prog	Proven Safe	Inconclusive	Assertions	Proven Assertions
SNF Zones	1632	**1296**	336	2423	**1918**
ELINA Zones	1632	1290	342	2423	1899
SNF Octagons	1591	1264	327	2146	1659
ELINA Octagons	1591	1264	327	2146	1659

Table 5. Pairwise comparison for precision of SNF Zones against ELINA Zones and SNF Octagons against ELINA Octagons on each widening point in SV-COMP programs with timeout of 180 seconds and memory limit of 8GB but considering only programs for which each pair of domains terminated before resources are exhausted.

	Widening Points	Same Precision	SNF More Precise	ELINA More Precise
Zones	20016	18975	1022	19
Octagons	17769	17696	41	14

Table 5 shows data similar to those of Table 4 by comparing SNF Zones against ELINA Zones and SNF Octagons against ELINA Octagons. Table 5, however, shows the precision differences at each widening point.

The tables suggest that there is no inherent difference between ELINA Octagons and SNF Octagons when it comes to precision. If there are differences in widening strategy, they do not seem to interfere significantly with the analysis overall. The tables show, however, a clear loss of precision for ELINA Zones, relative to SNF Zones. As discussed, widening is not the only possible source of differences, and the observed differences could possibly be due to implementation details for abstract transformers—abstract assignment in particular leaves room for variation.

8.5 *RQ3: How does Variable Packing interact with Split Normal Form?*

A final round of experiments were designed to contrast performance of the different abstract domain implementations when combined with variable packing, to see how the different approaches to reducing density compare.

Variable packing is a technique commonly used to improve the scalability of analyses based on relational numerical abstract domains. The idea is to group program variables into “packs” and then only infer relationships between variables in the same pack. Pack membership may be determined before the analysis (static variable packing) or during the analysis (dynamic variable

packing). The advantage is that analysis time (which tends to be cubic in the number of variables) is reduced considerably, provided packs can be kept small. On the other hand, packing comes with a non-negligible overhead, and its use makes the analysis sensitive to whatever criterion is used for pack determination. The effect on precision is due to the fact that any actual relationship between two variables is lost once the two have been placed in separate packs. Note that the extreme case of variable packing, where each pack is a singleton, effectively gives us interval analysis [27].

8.5.1 Performance. Figures 36-38 show again the efficiency of all the domains on the set of SV-COMP and eBPF programs, respectively, but this time comparing performance of each domain with and without variable packing.

For the SV-COMP programs, note that packing applied to the SNF abstract domains usually leads to a slowdown. This is in sharp contrast to what happens with the ELINA domains and an important lesson from the experiment. It appears that packing removes information that was not needed in the first place. Much of the benefit offered by variable packing is simply removal of phantom density from the DBM representation—something that is already accomplished by the SNF representations. Unlike the case of Zones, Figure 36(c) shows that packing can improve memory consumption of SNF Octagons at the expense of losing precision. We also compare, in Figure 37(a)-(b), the efficiency of SNF Octagons and APRON Octagons using variable packing (Pack+APRON Octagons). This experiment verifies the immense value of variable packing as a companion for a simpler Octagon implementation (APRON). Figure 37(a) shows that Pack+APRON Octagons is sometimes faster and consumes much less memory than our SNF Octagons. Figure 37(b) shows the impact of using Patricia Tries instead of the hybrid representation for SNF adjacency lists (SNF Octagons (PT)). The impact is large: the use of Patricia Tries can significantly reduce memory consumption of SNF Octagons, at the expense of a slower analysis.

For the eBPF programs (Figure 38), variable packing does lead to an increase in efficiency with respect to the SNF abstract domains, although only a slight one for SNF Zones. Again this is contrasted by the ELINA domains where the efficiency gained by packing is again significant. The reason for the difference is that the information loss from variable packing makes the whole analysis task simpler, so later abstract operations are simpler.

8.5.2 Precision. Table 6 compares precision of SNF Zones against Pack+SNF Zones, SNF Octagons against Pack+SNF Octagons, ELINA Zones against Pack+ELINA Zones, ELINA Octagons against Pack+ELINA Octagons, SNF Octagons (PT) against Pack+APRON Octagons on SV-COMP programs. Note that for the SV-COMP programs there is not a great difference in precision, but this is expected since the programs don't require much relational reasoning in any case.

Regarding eBPF programs, each domain without variable packing was able to prove correct all programs except for three, which require disjunctive reasoning. All five domains with variable packing, Pack+SNF Zones, Pack+SNF Octagons, Pack+ELINA Zones, Pack+ELINA Octagons, and Pack+APRON Octagons fail to prove the same 21 programs. This provides a stark contrast to the SV-COMP programs, and demonstrates that variable packing can introduce significant imprecision.

9 RELATED WORK

While there have been precursors for the use of difference constraints in program analysis (see Bagnara's thesis [2]), the development of the Zones and Octagons abstract domains is primarily due to Miné [44–49]. Miné provided detailed algorithmic design and analysis, intersecting the two important fields of abstract interpretation and constraint solving. On the abstract interpretation side, the so-called "weakly relational" abstract domains, including Zones and Octagons, were a response to the high runtime cost of using the much more expressive polyhedral abstract domain developed and investigated by Cousot and Halbwachs [19].

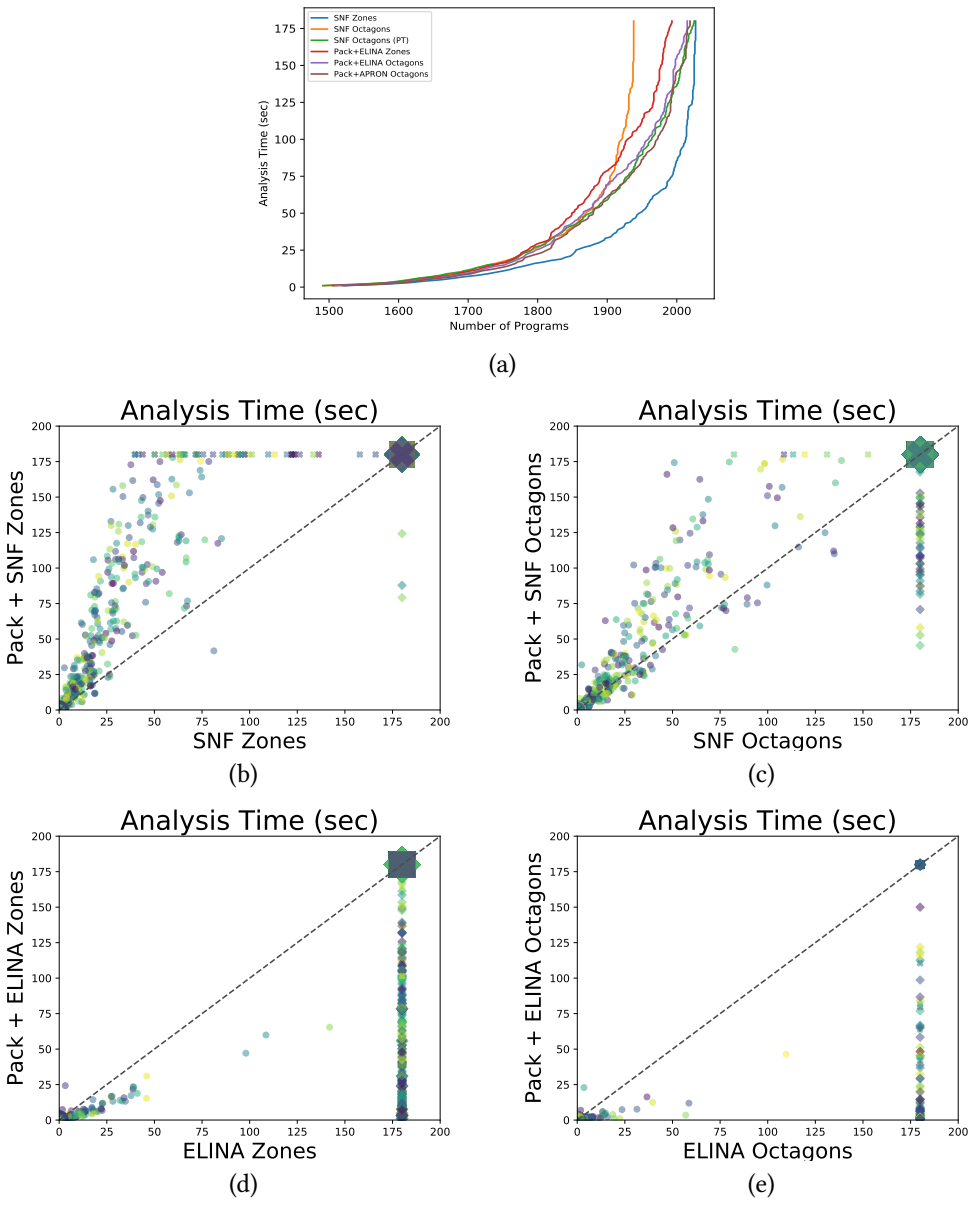


Fig. 36. Experiments with variable packing on SV-COMP programs with timeout of 180 seconds and memory limit of 8GB. The marker ● represents domains finished before exhausting resources, ✖ represents timeout, and ◆ represents memory-out. The size of a marker reflects the number of scatter points at that location.

The name “Zones” appears to have been used first by Miné in 2002 [47], but previous uses are found in the model checking literature. The domain is presented as an instance of a general scheme for the construction of certain relational abstract domains from non-relational “basis” domains, thus

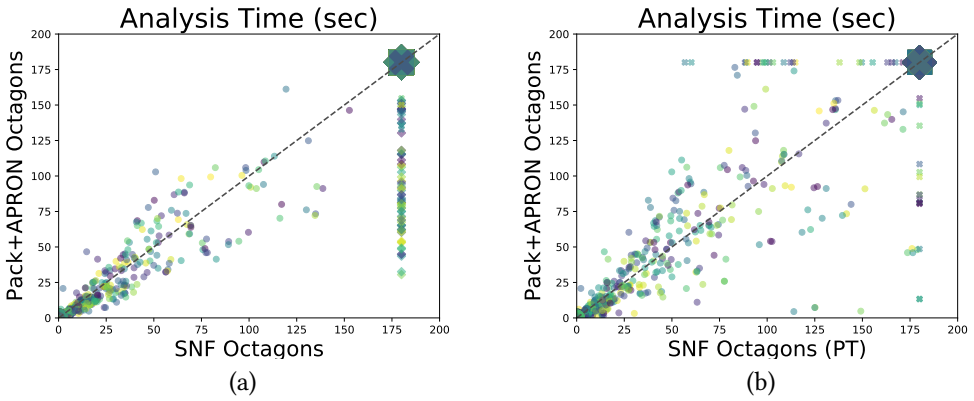


Fig. 37. Experiments with APRON and variable packing on SV-COMP programs with timeout of 180 seconds and memory limit of 8GB. The marker ● represents domains finished before exhausting resources, ✖ represents timeout, and ◆ represents memory-out. The size of a marker reflects the number of scatter points at that location.

Table 6. Pairwise comparisons for precision of SNF Zones against Pack+SNF Zones, SNF Octagons against Pack+SNF Octagons, ELINA Zones against Pack+ELINA Zones, ELINA Octagons against Pack+ELINA Octagons, SNF Octagons (PT) against Pack+APRON Octagons on SV-COMP programs with timeout of 180 seconds and memory limit of 8GB but considering only programs for which both domains of each pair terminated before resources are exhausted.

Domain	Prog	Proven Safe	Inconclusive	Assertions	Proven Assertions
SNF Zones	1977	1585	392	12846	11734
Pack+SNF Zones	1977	1582	395	12846	11711
SNF Octagons	1932	1547	385	10357	9274
Pack+SNF Octagons	1932	1542	390	10357	9251
ELINA Zones	1632	1290	342	2423	1899
Pack+ELINA Zones	1632	1289	343	2423	1888
ELINA Octagons	1591	1264	327	2146	1659
Pack+ELINA Octagons	1591	1259	332	2146	1640
SNF Octagons (PT)	2000	1600	400	24331	21257
Pack+APRON Octagons	2000	1595	405	24331	21232

putting Miné's earlier work [44, 45] in a broader context. "Octagons" appear under that name in a paper from 2001 [46], although that domain too is given detailed coverage already in Miné's masters thesis [44]. Implementations were made available through the Apron static analysis library [36].

Miné's work borrowed ideas from the model checking and constraint solving communities [32, 42, 55]. However, the program analysis problem is somewhat different to the problems addressed by those communities, because program analysis uses constraint sets as *descriptions* of possible runtime states. The focus is not entirely on the *solutions* to constraints. In particular, operations such as join and widening are of no interest to constraint solving, but are essential components in program analysis—to describe the runtime states that obtain at points of control flow confluence, and to guarantee termination of analysis.

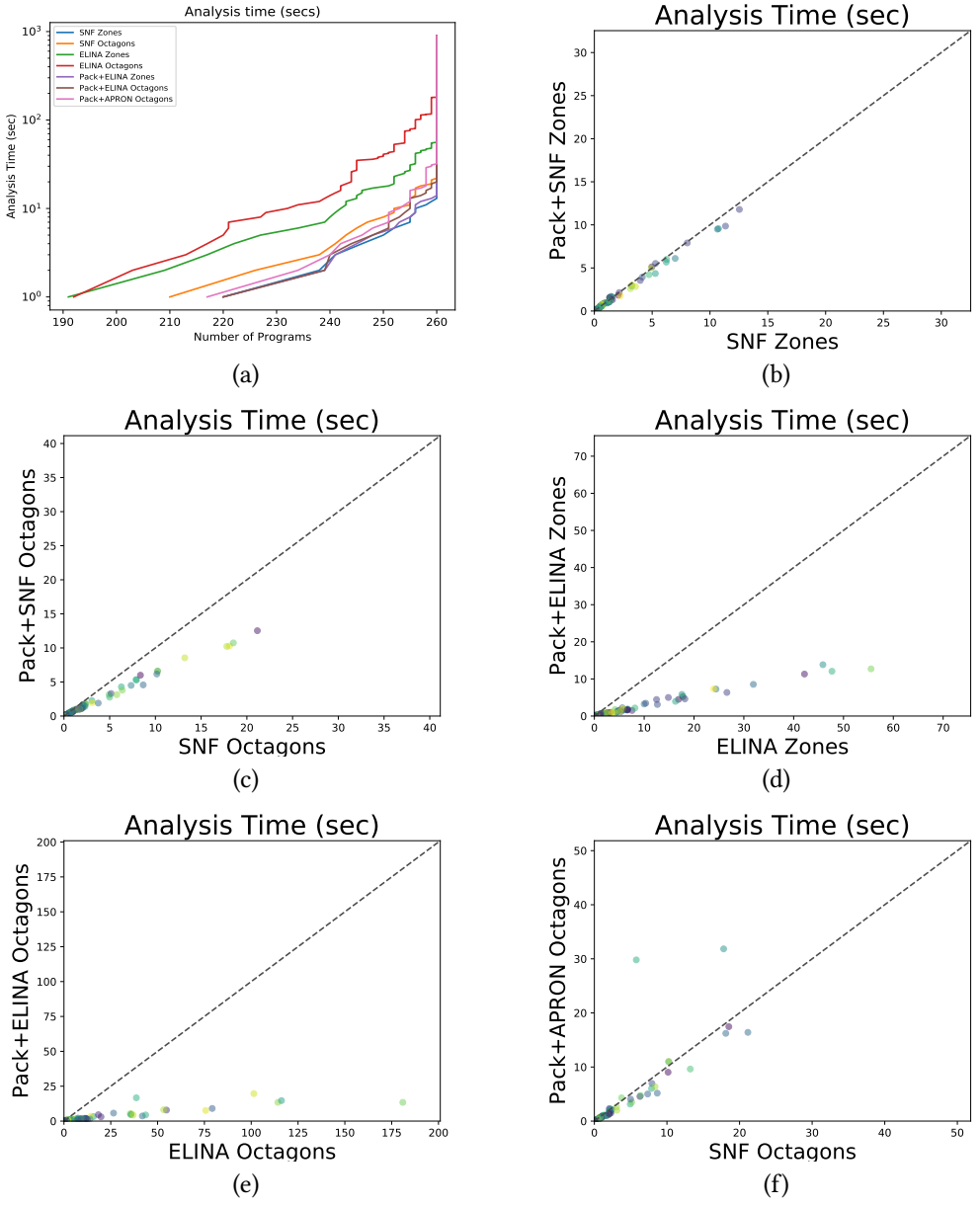


Fig. 38. Experiments on eBPF programs with variable packing.

9.1 Similar Abstract Domains

Simon, King and Howe [57] explored the use of more expressive TVPI (two variables per inequality) constraints for program analysis. TVPI lifts the limit on coefficients that Octagons constraints impose, namely that coefficients belong to the set $\{-1, 0, 1\}$. Simon, King and Howe [57] provided polynomial-time algorithms for TVPI abstract operations, including widening.

A different generalisation of Octagons is the Octahedron abstract domain [14] which allows for constraints that involve more than just two variables but maintains the limitation on coefficients that they must be in $\{-1, 0, 1\}$. Algorithms for the abstract operations are given, based on a decision diagram data structure invented for the purpose.

Numerous other abstract domains have been proposed which, compared to Octagons, provide incomparable expressiveness. Generally the idea is to sacrifice some precision, in favour of better performance. Some methods abandon the systematic tr-closure of relations (and work around the resulting lack of a normal form for constraints). Constraints implied by closure may be discovered lazily, or not at all. This is the case, for example, with Logozzo and Fähndrich's *Pentagon* domain [43]. *Template constraints* [53] offer constraints of the same general form as polyhedral analysis, but generally restrict, up front, the number (and shape) of constraints that can be in play.

9.2 Variable Packing

Variable packing provides an alternative approach to improving program analysis based on weakly relational domains. It also rests on the observation that, usually, only few pairs or clusters of variables are related. Variable packing consists of grouping variables into "buckets" or *packs* according to some criterion, such as their joint appearance in an assignment statement. Packs may be allowed to overlap. Then, instead of keeping a single zone or octagon as a program state description, *variable packing* methods keep a set, each describing the relations that hold in a given pack. With variable packing, one may hope to pay a superlinear computational cost only for lower-dimensional sub-spaces. This idea was utilised in the Astrée analyzer [20], where it was found to help scalability to a considerable extent. A dynamic variant was used in the C Global Surveyor [61].

Heo *et al.* [34] have suggested that machine learning can be helpful as part of a pre-analysis, to determine, up front, variable clusters that are suitable for packing[12]. It is, however, worth stressing that all approaches to variable packing are lossy, in the sense that optimal analysis would require optimal choice of packs, and no variable packing approach guarantees that.

Simon and King [56] showed that, also in the case of polyhedral analysis [19] is it possible to capitalise on the typical sparsity. They utilise, algorithmically, the fact that a given variable usually appears in very few inequalities, albeit not through a change of representation, as we do. Their key observation is that, for a sparse system of linear inequalities, variable elimination can be performed efficiently with the Fourier-Motzkin approach, and, moreover, the calculation of convex hulls can be done through clever use of variable elimination. As a result, Simon and King can avoid the traditional "double description" representation and instead implement a polyhedral analysis that is entirely matrix-based.

The Gauge domain proposed by Venet [62] can be seen as a combination of a similar kind of dimensionality restriction and the use of weakly relational domains. In the Gauge domain, relations are only maintained between program variables and specially introduced "loop counter" variables. Variable packing is also applied by Singh *et al.* [58, 59] whose contributions we discuss in Section 9.5.

In Section 8.5 we saw the idea of variable packing working well for ELINA abstract domains, but failing to have much impact for SNF domains. This should not be surprising: The role of variable packing is to combat unnecessary density, but SNF domains are designed to preserve sparsity throughout an analysis, that is, to avoid the introduction of phantom density in the first place.

[12] A related use of machine learning was recently proposed [33] to identify cases where the result of join operations can be simplified, without undue loss of precision of the overall analysis.

9.3 Algorithmic Improvements

Important improvements to the performance of weakly-relation abstract domains were suggested by Bagnara *et al.* [3]. The improvements benefited Octagons in particular, owing to the discovery of a more efficient closure algorithm which substantially reduced the required number of coefficient operations, compared to the algorithm described by Miné [49]. (However, similar to most of the previous and subsequent work, the new algorithms still relied on the expensive strong closure.) The improvement was implemented in the Parma Polyhedra Library (PPL) [4].

Bagnara *et al.* also designed better widening operators for these domains [3], allowing some precision to be surrendered to improve analysis performance in the case of slow convergence. The latter work identified the redundancy created by variable bounds, but only in the context of widening. Later again, Bagnara *et al.* [5] reconsidered the representation of weakly relational domains, proposing *transitive reduction* as a canonical form ("shapes"). Widening was then defined with reference to the operand "shapes" rather than the graph itself. This way, redundant relations could be ignored while performing widening, preventing any redundancy from being inherited in the result. This idea too has been implemented in the PPL [4]. Since our split normal form avoids the aforementioned redundancy, for us, standard widening results in a *non-redundant* system of constraints, so no special widening is required.[13]

Chawdhary, Robbins and King [11] also present algorithmic improvements for the Octagons domain, assuming the standard matrix-based implementation (built on DBMs). Their focus is on the common use case where a single constraint is added/changed, that is, the explicit goal is an improved algorithm for *incremental* closure. Chawdhary, Robbins and King implemented incremental closure algorithms in OCaml, including for strong closure, as well as the case of $D = \mathbb{Z}$, and compared these experimentally, using randomly generated feasible Octagon constraints.

9.4 Phantom Density and Data Structure Improvements

All of the related work discussed so far is fundamentally based on the use of data structures that are most suitable for dense graphs. The observation about "phantom density" made in the introduction of the present paper suggests that Zones and Octagons analysis should be able to capitalise on the inherent sparsity in typical problem instances, as illustrated in Example 5 and verified experimentally in Section 8. The classical graph representations provide an elegant way of capturing unary and binary constraints simultaneously, but it does not follow that they also lead to the most efficient implementations.

While the focus of the present paper has been on Zones and Octagons, the idea of a split normal form can be applied more generally to abstract domains where entailed constraint systems are stored in a closed (or *saturated*) form. This is dependent on the underlying representation of the domain. As presented by Simon, King and Howe [57], the TVPI domain is (just like octagons) represented as a tr-closed graph, and the techniques proposed in the present paper could equally be applied to TVPI. Similarly, the Octahedron [14] domain maintains a decision-diagram representation of all nontrivial unit inequalities, which could likely benefit from a split representation. However, the applicability of the idea only goes so far. While the convex polyhedron [19] domain also tracks a set of constraints for each state (paired with a dual set of generators), its constraint system is maintained in an irredundant (not closed) form, so splitting out monadic properties will not help this abstract domain.

The "phantom density" observation goes back to Gange *et al.* [28], in the context of Zones analysis and, independently, Jourdan [38, 39], in the context of Octagons analysis. Jourdan's

[13] Gange *et al.* [29] discuss various problems that flow from a misalignment of syntax-based widening methods and their semantic underpinning, and propose instead a more general view of widening.

dissertation [38] goes well beyond a study of the Octagons abstract domain. Its focus is on *verified* static analysis, using the Coq proof assistant to establish the soundness of Verasco, an analysis tool [38] for C#minor. Verasco also offers interval analysis, expression linearization [48], and symbolic equalities, to improve the analysis of, for example, non-strict Boolean operations.

Jourdan observes that, although a strongly closed DBM gives the best abstract state, the strong closure of Octagons often causes abstract states to be dense because of the *strengthening* step. He aims, as we do, for preservation of weak closure and a set of operations that can preserve sparsity. The notion of weak closure proposed by Jourdan is similar to the split normal form closure defined in Section 7.1: both are used to avoid the density generated by the strengthening step. Jourdan [38] showed that the abstract operators for the Octagons domain do not lose precision in the presence of weak closure.

Similarly, the join operation given by Jourdan (for Octagons) is closely related to what we present in Figure 29, although Jourdan's operation is expressed at a much higher level. For other operations, such as meet and constraint addition, our versions are necessarily more complex, owing to the maintenance of potential functions—a price we pay for highly efficient operations overall. Jourdan does not provide implementation details for abstract assignment. He observes that operations other than join and constraint addition can remain unchanged for weakly closed Octagons, without loss of precision—formal proofs of this are provided in his dissertation [38].

There are two main differences between Jourdan's work (on Octagons analysis) and the work presented in this paper. The first is our use of bespoke data structures and algorithms designed specifically to exploit sparsity and enable minimal traversal of graphs (including the use of potential functions). The second is our ability to run meaningful experiments, comparing with similar analysis tools, and to assess the impact, in practice, of more sophisticated data structures and algorithms for this type of static analysis.

A rather different approach to reducing the size of matrices used to represent Zones and Octagons is proposed by Chawdhary and King [10]. As in Apron's Octagons implementation, matrix symmetry (or coherence, in the case of Octagons) is utilized to obtain a half-matrix form, maintained as an array of size $2n(n+1)$ in the case of Octagons representations in the presence of n variables. Then the array elements (GMP rationals) are replaced by pointers to two smaller arrays, one in which the rational values are stored, but without repetition, and another, to help identify the matrix elements that map to each rational value. The idea is to capitalise on an expected repetition of many rational values in the matrix. It is given a limited experimental evaluation. The approach does not address what we consider the main obstacle to better performance, namely the commitment to a dense representation. As with other previous work, Jourdan's excepted, it ignores the inherent sparsity of the problem (or the "phantom density" that manifests itself).

9.5 Exploiting Parallelism

For their implementation of the Octagons domain, Banterle and Giacobazzi [6] pioneered the use of graphics hardware in the service of program analysis. The matrix representation of Octagons constraints was treated as a 2D texture and the Octagons abstract operations were implemented as graphics operations, utilising data parallelism where possible.

Parallelization is also an aim of the "OptOctagon" approach of Singh *et al.* [58]. OptOctagon was a novel approach to Octagons analysis which has subsequently become part of the ELINA library. Several of the ideas mentioned above are utilized in OptOctagons. For example, the implementation makes use of dynamic variable packing (see also [59]). It uses, as a starting point, an Apron-like "half-matrix" representation of DBMs. But a central feature in OptOctagons is the ability to switch, dynamically, from one kind of DBM representation to another (chosen among "top", "dense", "sparse", and "decomposed"). The hypothesis is that a different representation may be

suited for different stages of an analysis. Hence a measure of sparsity is continually calculated, and this measure dictates when a switch is to be made to a different, more suitable, representation. The abstract operations can then be specialised for the different representations, leading to very different behaviours and performance, in particular compared to (the usually expensive) closure. Importantly (and the reason why we discuss the work in this sub-section on parallelism), for the dense representation, techniques borrowed from high-performance linear algebra can be utilized, including vectorization [31].

Singh *et al.* [58] evaluate OptOctagons experimentally, by comparing against Apron across some 40 sizeable benchmark programs. Very large speedups are evident. However, the data provided also demonstrate very clearly that phantom density is a real issue. A graph ([58], Figure 7) shows the running time of the successive closure operations for one random benchmark. As the authors comment, "the DBMs are dense in the beginning but become sparse due to widening midway through the analysis." But, as we have argued in this paper, the phenomenon of dense DBMs early on is both unwanted and unnecessary.

In contrast, we see Zones/Octagons types of program analysis as essentially-sparse graph problems. In Section 5 we have argued that the density observed by Singh *et al.* [58]) is artificial—it stems from a failure to separate independent properties from truly relational properties. Our approach is therefore almost diametrically opposite that of Singh *et al.* [58] as we choose to exploit the innate sparsity as best we can. For that reason, we have found a comparison between the two implementations a very worthwhile exercise. Many of the ideas from each would seem to be orthogonal. We hope the present paper will encourage further development, possibly by combining the insights from both approaches.

10 CONCLUSION AND FUTURE WORK

In this paper, we have addressed the problem of scalable implementation of weakly relational abstract domains. We have described the reductions to graph shortest-path problems that underlie the classical work on the Zones and Octagons abstract domains. Traditional implementations have suffered from scalability problems, partly due to how constraint graphs have been represented. We have argued that much of the graph density that quickly creeps in during analysis is primarily an artefact of a poor choice of representation.

An alternative *Split Normal Form* permits separate handling of non-relational and relational properties, both in Zones [28] and Octagons [39]. In principle, this should allow for much improved implementations of both Zones and Octagons. This paper contains complete sets of algorithms, utilising split normal form, for all relevant abstract operations in each of the two abstract domains. Implementations are available as open source [21].

To explore whether the in-principle advantages of Split Normal Form translate to better performance in practice, we have conducted extensive experiments based on programs from eBPF and SV-COMP 2019. These programs have assertions, which allows us to evaluate not only efficiency, but also the levels of precision obtained, and in particular, the impact of widening. Regarding efficiency, we have evaluated the implementations against the state-of-the-art analyzer ELINA [58, 59] which also implements both the Zones and the Octagons domains. We have been able to compare like with like, since ELINA's approach has been implemented as part of the Crab analyzer.

The evaluation shows that it really is important to use a representation that prevents bounds information from inadvertently polluting relational information. The idea behind Split Normal Form is to keep bounds information at arm's length from relational information. In the traditional implementations, the phenomenon we have referred to as "phantom density" clearly manifests itself in the analysis of real-world programs.

Evaluation also shows that much of the benefit of variable packing is the removal of phantom density from the DBM representation, and therefore, Split Normal Form does not significantly benefit from it. Variable packing comes with a potential loss of precision in case the grouping of variables is sub-optimal. This makes Split Normal Form domains good alternatives to the conjunction of Zones/Octagons and variable packing. The performance characteristics of the SNF domains match those of variable packing, yet come with better performance guarantees.

On the other hand, the abstract operations presented in this paper are rather more intricate than the classical operations defined using dense representations. It is also worth pointing out that the use of variable packing, as a general conjunct to abstract domains, is a more flexible approach, being applicable also to polyhedral analysis, for example. This is important to stress, since in many cases it is the limited expressiveness of Zones and Octagons that prevents successful use, including in program verification.

As to the question of how different widening algorithms affect precision in practice, Table 5 suggests that Split Normal Form and ELINA widening are incomparable. For Octagons, the differences on SV-COMP 2019 benchmarks are negligible, as each leads to the same number of assertions proved. However, for Zones, SNF appears to achieve higher precision.

While the approach of Singh *et al.* [58] is based on the same goals and observations as the present work, the two approaches are almost diametrically opposed. ELINA remains committed to a dense matrix representation, even if it is not used exclusively. The application of sophisticated techniques such as vectorization and the use of variable packing ("decomposition") does not appear to be sufficient to make up for the disadvantages brought by a dense representation. We instead tackle the problem by taking the greatest possible advantage of the natural sparsity of the constraints involved, leading to entirely different data structures and algorithms. Techniques such as vectorization can improve running times by a constant factor. But for graph manipulation, data structures and algorithms that can capitalise on inherent sparsity have a greater potential to deliver efficiency at a scale beyond constant factors.

Of course the two approaches are not mutually exclusive. Perhaps the main conclusion from our work is that there still appears to be scope for better engineered Zones and Octagons analysis.

ACKNOWLEDGMENTS

We wish to thank the three anonymous reviewers for their meticulous work. Their many insightful suggestions improved the paper considerably. Graeme Gange has been supported by the Australian Research Council under Discovery Early Career Researcher Award DE160100568. Jorge Navas has been supported by the US National Science Foundation under grant number 1816936.

REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 1 (2):131–137, 1972.

[2] R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages.* PhD thesis, Università di Pisa, 1997.

[3] R. Bagnara, P. M. Hill, E. Mazzi, and E. Zaffanella. Widening operators for weakly-relational numeric abstractions. In C. Hankin and I. Siveroni, editors, *Static Analysis: Proceedings of the 12th International Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2005.

[4] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1-2):3–21, 2008.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[6] F. Banterle and R. Giacobazzi. A fast implementation of the Octagon abstract domain on graphics hardware. In H. Riis Nielson and G. Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 315–332. Springer, 2007.

[7] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[8] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. ACM Press, 2003.

[9] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.

[10] A. Chawdhary and A. King. Compact difference bound matrices. In B.-Y. E. Chang, editor, *Programming Languages and Systems (APLAS'17)*, volume 10695 of *Lecture Notes in Computer Science*, pages 471–490. Springer, 2017.

[11] A. Chawdhary, E. Robbins, and A. King. Incrementally closing Octagons, 2016. Version 1, https://arXiv.org/format/1610.02952.

[12] B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85(2):277–311, 1999.

[13] Clam team. Clam: Crab for Llvm Abstraction Manager. Available at https://github.com/seahorn/crab-llvm.

[14] R. Clarisó and J. Cortadella. The Octahedron abstract domain. In R. Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[16] S. Cotton and O. Maler. Fast and flexible difference constraint propagation for DPLL(T). In A. Biere and C. P. Gomes, editors, *Theory and Applications of Satisfiability Testing (SAT 2006)*, volume 4121 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2006.

[17] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[18] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[19] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.

[20] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[21] Crab. CoRnucopia of ABstractions: A language-agnostic library for abstract interpretation. Available at https://github.com/seahorn/crab.

[22] EBPF. A set of EBPF programs. Available at https://github.com/vbpf/ebpf-samples.

[23] ELINA team. ELINA: ETH LIbrary for Numerical Analysis. Available at https://github.com/eth-srl/ELINA.

[24] T. Feydy, A. Schutt, and P. J. Stuckey. Global difference constraint propagation for finite domain solvers. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 226–235. ACM Press, 2008.

[25] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962.

[26] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[27] G. Gange, J. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Transactions on Programming Languages and Systems*, 37(1):1:1–1:35, 2015.

[28] G. Gange, J. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Exploiting sparsity in difference-bound matrices. In X. Rival, editor, *Static Analysis: Proceedings of the 23rd International Symposium*, volume 9837 of *Lecture Notes in Computer Science*, pages 189–211. Springer, 2016.

[29] G. Gange, J. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Dissecting widening: Separating termination from information. In A. W. Lin, editor, *Proceedings of the 17th Asian Symposium on Programming Languages and Systems*, volume 11893 of *Lecture Notes in Computer Science*, pages 95–114. Springer, 2019.

[30] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019.

[31] S.-C. Han, F. Franchetti, and M. Püschel. Program generation for the all-pairs shortest path problem. In *Proceedings of the 2006 International Conference on Parallel Architectures and Compilation Techniques*, pages 222–232. IEEE Comp. Soc., 2006.

[32] W. Harvey and P. J. Stuckey. A unit two variable per inequality integer constraint solver for constraint logic programming. In *Proceedings of the Australasian Computer Science Conference*, pages 102–111, 1997.

[33] J. He, G. Singh, M. Püschel, and M. Vechev. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM Symposium on Programming Language Design and Implementation*, pages 1112–1127. ACM Press, 2020.

[34] K. Heo, H. Oh, and H. Yang. Learning a variable-clustering strategy for Octagon from labeled data generated by a static analysis. In X. Rival, editor, *Static Analysis*, volume 9837 of *Lecture Notes in Computer Science*, pages 237–256.

Springer, 2016.
[35] J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap. Beyond finite domains. In *Proceedings of the International Workshop on Principles and Practices of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 86–93. Springer, 1994.
[36] B. Jeannet and A. Miné. A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.
[37] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.
[38] J.-H. Jourdan. *Verasco: A Formally Verified C Static Analyzer*. PhD thesis, Université Paris Diderot, 2016.
[39] J.-H. Jourdan. Sparsity preserving algorithms for Octagons. *Electronic Notes in Theoretical Computer Science*, 331:57–70, 2017.
[40] J. Kuderski, J. A. Navas, and A. Gurfinkel. Unification-based pointer analysis without oversharing. In *Proceedings of the 19th Conference on Formal Methods in Computer-Aided Design (FMCAD 2019)*, pages 37–45. FMCAD, Inc., 2019.
[41] S. K. Lahiri and M. Musuvathi. An efficient decision procedure for UTVPI constraints. In B. Gramlich, editor, *Frontiers of Combining Systems*, pages 168–183. Springer, 2005.
[42] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings of the 18th International Symposium on Real-Time Systems*, pages 14–24. IEEE Comp. Soc., 1997.
[43] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 184–188. ACM Press, 2008.
[44] A. Miné. Représentation d'ensembles de contraintes de somme ou de différence de deux variables et application à l'analyse automatiques de programmes. Master's thesis, École Normale Supérieure, Paris, 2000.
[45] A. Miné. A new numerical abstract domain based on difference-bound matrices. In O. Danvy and A. Filinski, editors, *Programs as Data Objects*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.
[46] A. Miné. The Octagon abstract domain. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 310–319. IEEE Comp. Soc., 2001.
[47] A. Miné. A few graph-based relational numerical abstract domains. In M. Hermenegildo and G. Puebla, editors, *Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 117–132. Springer, 2002.
[48] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Paris, 2004.
[49] A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
[50] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In E. A. Emerson and K. S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2006.
[51] G. Nemhauser. A generalized permanent label setting algorithm for the shortest path between specified nodes. *Journal of Mathematical Analysis and Applications*, 38(2):328–334, 1972.
[52] Prevail team. PREVAIL: A Polynomial-Runtime EBPF Verifier using an Abstract Interpretation Layer. Available at https://github.com/vbpf/ebpf-verifier.
[53] S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear relations analysis. In R. Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2004.
[54] A. Schutt and P. J. Stuckey. Incremental satisfiability and implication for UTVPI constraints. *INFORMS Journal of Computing*, 22(4):514–527, 2010.
[55] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, 1981.
[56] A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In C. Hankin, editor, *Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2005.
[57] A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation: Proceedings of the 12th International Workshop*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2003.
[58] G. Singh, M. Püschel, and M. Vechev. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 303–313. ACM Press, 2015.
[59] G. Singh, M. Püschel, and M. T. Vechev. A practical construction for decomposing numerical abstract domains. *Proceedings of the ACM on Programming Languages*, 2(POPL):55:1–55:28, 2018.
[60] SVCOMP. Competition on software verification (SV-COMP), 2019. http://sv-comp.sosy-lab.org/2019/. Benchmarks available at https://github.com/sosy-lab/sv-benchmarks/c.
[61] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 231–242. ACM Press, 2004.
[62] A. J. Venet. The Gauge domain: Scalable analysis of linear inequality invariants. In P. Madushan and S. A. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 139–154. Springer, 2012.