Incremental Dead State Detection in Logarithmic Time

Anonymous Authors

No Institute Given

Abstract. Identifying live and dead states in an abstract transition system is a recurring problem in formal verification; for example, it arises in recent work for efficiently deciding regex constraints in SMT. However, state-of-the-art graph algorithms for maintaining reachability information *incrementally* (that is, as states are visited and before the entire state space is explored) assume that new edges can be added from any state at any time, whereas in many applications, outgoing edges are added from each state as it is explored. To formalize the latter situation, we propose *guided incremental digraphs* (GIDs), incremental graphs which support labeling *closed* states (states which will not receive further outgoing edges). Our main result is that dead state detection in GIDs is solvable in $O(\log m)$ time per edge update for m edges, improving upon $O(\sqrt{m})$ per edge due to Bender, Fineman, Gilbert, and Tarjan (BFGT) for general incremental directed graphs.

We introduce two algorithms for GIDs: one establishing the logarithmic time bound, and a second algorithm to explore a lazy heuristics-based approach. To enable an apples-to-apples experimental comparison, we implemented both algorithms, two simpler baselines, and the state-of-the-art BFGT baseline using a common directed graph interface in Rust. Our evaluation shows 110-530x speedups over BFGT for the largest input graphs over a range of graph classes, random graphs, and graphs arising from regular expression benchmarks.

Keywords: Dead State Detection · Graph Algorithms · Online Algorithms · SMT.

1 Introduction

Classifying states in a transition system as live or dead is a recurring problem in formal verification. For example, given an expression, can it be simplified to the identity? Given an input to a nondeterministic program, can it reach a terminal state, or can it reach an infinitely looping state? Given a state in an automaton, can it reach an accepting state? State classification is relevant in the context of satisfiability modulo theories (SMT) [18,41,3,55,15,36], where theory-specific partial decision procedures often work by exploring the state space to find a reachable path that corresponds to a satisfying string or, more generally, a sequence of constructors. In all of these cases, the core problem is live and dead state detection in a directed graph.

Motivating application. For example, recent approaches for the SMT theory of regular expressions [34,50] rely on regular expression *derivatives* to explore the states of the finite state machine corresponding to the regex incrementally, rather than expanding all states initially which is often prohibitively expensive. This requires solving the incremental live and dead state detection problem in the finite state machine (a directed graph). This is particularly important for regexes with intersection and complement (*extended* regexes [12,23,20]), which have been shown to arise natively in applications of SMT string solvers to security [2,50]). Concretely, consider the regex $(.^*\alpha.^{100})^C \cap (.\alpha)$, where $.$ matches any character, $\cap$ is regex intersection, ^{C} is regex complement, and α matches any digit (0-9). A traditional solver would expand the left and right operands as state machines, but the left operand $(.^*\alpha.^{100})^C$ is astronomically large as a DFA, causing the solver to hang. The derivative-based technique instead constructs the derivative regex: $(.^*\alpha.^{100})^C \cap (.^{100})^C \cap \alpha$. At this stage we have a graph of two states and one edge, where the states are regexes and the edge is the derivative relation. After one more derivative operation, the regex becomes one that is clearly satisfiable as it accepts the empty string.

In order to be efficient, a derivative-based solver needs to identify satisfiable (live) and unsatisfiable (dead) regexes *incrementally* (as the graph is built), because it does not generally construct the entire space before terminating (see the graph update inference rule Upd, p. 626 [50]). Moreover, the nonemptiness problem for extended regexes is non-elementary [51] — and still PSPACE-complete for more restricted fragments — strongly incentivizing the incremental approach. Beyond regexes, we believe that GIDs are general enough to be applicable in a range of future applications.

Prior work. Traditionally, while live state detection can be done incrementally, dead state detection is often done exhaustively (i.e., after the entire state space is explored). For example, bounded and finite-state model checkers based on translations to automata [32,48,13], as well as classical dead-state elimination algorithms [28,7,10], generally work on a fixed state space after it has been fully enumerated. However, exhaustive exploration is prohibitive for large (e.g., exponential or infinite) state spaces which arise in an SMT verification context. Moreover, exhaustive exploration may simply be unnecessary if partial information can be deduced about the states seen so far which already leads to a satisfiable or unsatisfiable result along a given solver path. We also have good evidence that incremental feedback can improve SMT solver performance: a representative success story is the e-graph data structure [54,17] for congruence closure [19,42], which maintains an equivalence relation among expressions incrementally; because it applies to general expressions, it is theory-independent and re-usable. Incremental state space exploration could lead to similar benefits if applied to SMT procedures which still rely on exhaustive search.

However, in order to perform *incremental* dead state detection, we currently lack algorithms which match offline performance. As we discuss in Section 2, the best-known existing solutions would require maintaining strong connected components (SCCs) incrementally. For SCC maintenance and the related sim-

pler problem of cycle detection, $O(m^{3/2})$ amortized algorithms are known for m edge additions [24,4], with some recently announced improvements [5,8]. Note that this is in sharp contrast to $O(m)$ for the offline variants of these problems, which can be solved by breadth-first or depth-first search. More generally, research suggests that there is a computational barrier to what can be determined incrementally in the worst case [22,1].

This paper. To improve on prior results, our key observation is that in many applications, edges are not added adversarially, but *from one state at a time* as the states are explored. As a result, we know when a state will have no further outgoing edges. This enables us to (i) identify dead states incrementally, rather than only after the whole state space is explored; and (ii) obtain more efficient algorithms than currently exist for general graph reachability.

We introduce *guided incremental digraphs* (GIDs), a variation on incremental graphs. Like an incremental directed graph, a guided incremental digraph may be updated by adding new edges between states, or a state may be labeled as *closed*, meaning it will receive no further outgoing edges. Some states are designated as *terminal*, and we say that a state is *live* if it can reach a terminal state and *dead* if it will never reach a terminal state in any extension – i.e. if all reachable states from it are closed. To our knowledge, the problem of detecting dead states in such a system has not been studied by existing work in graph algorithms. Our problem can be solved through solving SCC maintenance, but not necessarily the other way around (see Proposition 1). We provide two new algorithms for dead-state detection in GIDs.

First, we show that the dead-state detection problem for GIDs can be solved in time $O(m \cdot \log m)$ for m edge additions, within a logarithmic factor of the $O(m)$ cost for offline search. The worst-case performance of our algorithm thus strictly improves on the $O(m^{3/2})$ upper bound for SCC maintenance in general incremental graphs. Our algorithm is complex, and utilizes several data structures and existing results in online algorithms: in particular, Union-Find [52] and Henzinger and King's Euler Tour Trees [27]. The main idea is that, rather than explicitly computing the set of SCCs, for closed states we maintain a single path to a non-closed (open) state. This turns out to reduce the problem to quickly determining whether two states are currently assigned a path to the same open state. On the other hand, Euler Tour Trees can solve *undirected* reachability for graphs that are forests in logarithmic time; the challenge then lies in figuring out how to reduce directed connectivity in the graph of paths to *undirected* connectivity in an Euler Tour Trees forest. At the same time, we must maintain this reduction under Union-Find state merges, in order to deal with cycles that are found in the graph along the way.

While as theorists we would like to believe that asymptotic complexity is enough, the truth is that the use of complex data structures (1) can be prohibitively expensive in practice due to constant-time overheads, and (2) can make algorithms substantially more difficult to implement, leading practitioners to prefer simpler approaches. To address these needs, in addition to the logarithmic-time algorithm, we provide a second *lazy* algorithm which avoids

the user of Euler Tour Trees, and only uses union-find. This algorithm is based on an optimization of adding shortcut *jump* edges for long paths in the graph to quickly determine reachability. This approach aims to perform well in practice on typical graphs, and is evaluated in our evaluation along with the logarithmic time algorithm, though we do not prove its asymptotic complexity.

Finally, we implement and empirically evaluate both of our algorithms for GIDs against several baselines in 5.5k lines of code in Rust [37,33]. Our evaluation focuses on the performance of the GID data structure itself, rather than its end-to-end performance in applications. To ensure an apples-to-apples comparison with existing approaches, we put particular focus on providing a directed graph data structure backend shared by all algorithms, so that the cost of graph search as well as state and edge merges is identical across algorithms. We implement two naïve baselines, as well as an implementation of the state-of-the-art solution based on maintaining SCCs, BFGT [4] in our framework. To our knowledge, the latter is the first implementation of BFGT for SCC maintenance. On a collection of generated benchmark GIDs, random GIDs, and GIDs directly pulled from the regex application, we demonstrate a substantial improvement over BFGT for both of our algorithms. For example, for larger GIDs (those with over 100K updates), we observe a 110-530x speedup over BFGT.

Our primary contributions are:

- *Guided incremental digraphs* (GIDs), a formalization of incremental live and dead state detection which supports labeling *closed* states. (Section 2)
- Two algorithms for the state classification problem in GIDs: first, an algorithm that works in amortized $O(\log m)$ time per update, improving upon the state-of-the-art amortized $O(\sqrt{m})$ per update for incremental graphs; and second, a simpler algorithm based on lazy heuristics. (Section 3)
- An open-source implementation of GIDs in Rust and an evaluation which demonstrates two orders of magnitude speedup over BFGT for the largest benchmarks. (Section 4)

Following the above, we expand on the application of GIDs to regex solving in SMT (Section 5) and survey related work (Section 6).

2 Guided Incremental Digraphs

2.1 Problem Statement

An incremental digraph is a sequence of edge updates $\texttt{E}(u, v)$, where the algorithmic challenge in this context is to produce some output after each edge is received (e.g., whether or not a cycle exists). If the graph also contains updates $\texttt{T}(u)$ labeling a state as *terminal*, then we say that a state is *live* if it can reach a terminal state in the current graph. In a *guided* incremental digraph, we also include updates $\texttt{C}(u)$ labeling a state as *closed*, meaning that will not receive any further outgoing edges.

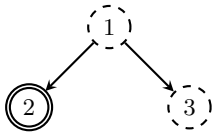

Fig. 1. GID consisting of the sequence of updates $\texttt{E}(1,2)$, $\texttt{E}(1,3)$, $\texttt{T}(2)$. Terminal states are drawn as double circles. After the update $\texttt{T}(2)$, states 1 and 2 are known to be live. State 3 is not dead in this GID, as a future edge may cause it to be live.

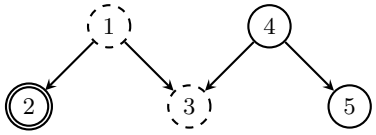

Fig. 2. GID consisting of the sequence of updates $\texttt{E}(1,2)$, $\texttt{E}(1,3)$, $\texttt{T}(2)$, $\texttt{E}(4,3)$, $\texttt{E}(4,5)$, $\texttt{C}(4)$, $\texttt{C}(5)$. Closed states are drawn as solid circles. After the update $\texttt{C}(5)$ (but not earlier), state 5 is dead. State 4 is not dead because it can still reach state 3.

Definition 1. Define a *guided incremental digraph (GID)* to be a sequence of updates, where each update is one of the following:

(i) a new directed *edge* $\texttt{E}(u, v)$;
(ii) a label $\texttt{T}(u)$ which indicates that u is *terminal*; or
(iii) a label $\texttt{C}(u)$ which indicates that u is *closed*, i.e. no further edges will be added going out from u (or labels to u).

The GID is *valid* if the *closed* labels are correct: there are no instances of $\texttt{E}(u, v)$ or $\texttt{T}(u)$ after an update $\texttt{C}(u)$. The *denotation* of G is the directed graph (V, E) where V is the set of all states u which have occurred in any update in the sequence, and E is the set of all (u, v) such that $\texttt{E}(u, v)$ occurs in G. An *extension* of a valid GID G is a valid GID G' such that G is a prefix of G'. In a valid GID G, we say that a state u is *live* if there is a path from u to a terminal state in the denotation of G; and a state u is *dead* if it is not live in *any* extension of G. Notice that in a GID without any $\texttt{C}(u)$ updates, no states are dead as an edge may be added in an extension which makes them live.

We provide an example of a valid GID in Figures 1 and 2 resulting from the following sequence of updates: $\texttt{E}(1, 2)$, $\texttt{E}(1, 3)$, $\texttt{T}(2)$, $\texttt{E}(4, 3)$, $\texttt{E}(4, 5)$, $\texttt{C}(4)$, $\texttt{C}(5)$. Terminal states $\texttt{T}(u)$ are drawn as double circles; closed states, as single circles $\texttt{C}(u)$; and states that are not closed, as dashed circles.

Definition 2. Given as input a valid GID, the *GID state classification problem* is to output, in an online fashion after each update, the set of new live and new dead states. That is, output $\texttt{Live}(u)$ or $\texttt{Dead}(u)$ on the smallest prefix of updates such that u is live or dead on that prefix, respectively.

2.2 Existing Approaches

In many applications, one might choose to classify dead states offline, after the entire state space is enumerated. This leads to a linear-time algorithm via either DFS or BFS, but it does not solve our problem (Definition 2) because it is not incremental. Naïve application of this idea leads to $O(m)$ per update for m updates ($O(m^2)$ total), as we may redo the entire search after each update.

For acyclic graphs, there exists an amortized $O(1)$-time per update algorithm for the problem (Definition 2): maintain the graph as a list of forward and backward edges at each state. When a state v is marked terminal, do a DFS along backward edges to determine all states u that can reach v not already marked as live, and mark them live. When a state v is marked closed, visit all forward-edges from v; if all are dead, mark v as dead and recurse along all backwards edges from v. As each edge is visited only when marking a state live or dead, it is only visited a constant number of times overall (though we may use more than $O(1)$ time on some particular update pass). Additionally, the live state detection part of this procedure still works for graphs containing cycles.

The challenge, therefore, lies primarily in detecting dead states in graphs which may contain cycles. For this, the breakthrough approach from [4] maintains a *condensed* graph which is acyclic, where the vertices in the condensed graph represent strongly connected components (SCCs) of states. The mapping from states to SCCs is maintained using a Union-Find [52] data structure. Maintaining the condensed graph requires $O(\sqrt{m})$ time per update. To avoid confusing closed and non-closed states, we also have to make sure that they are not merged into the same SCC; the easiest solution to this is to withhold all edges from each state u in the graph until u are closed, which ensures that u must be in a SCC on its own. Once we have the condensed graph with these modifications, the same algorithm as in the previous paragraph works to identify live and dead states. Since each edge is only visited when a state is marked closed or live, each edge is visited only once throughout the algorithm, we use only amortized $O(1)$ additional time to calculate live and dead states. While this SCC maintenance algorithm ignores the fact that edges do not occur from closed states $\texttt{C}(u)$, this still proves the following result:

Proposition 1. *GID state classification reduces to SCC maintenance. That is, suppose we have an algorithm over incremental graphs that maintains the set of SCCs in $O(f(m,n))$ total time given n states and m edge additions, where "maintains" means that (i) we can check whether two states are in the same SCC in $O(1)$ time, and (ii) we can iterate over all the states in an SCC, or iterate over the forward-edges or backward-edges from an SCC (to or from other SCCs, respectively) in $O(1)$ time per edge. Then there exists an algorithm to solve GID state classification in $O(f(m,n))$ total time.*

Despite this reduction one way, there is no obvious reduction the other way – from cycle detection or SCCs to Definition 2. This is because, while the existence of a cycle of non-live states implies bi-reachability between all states in the cycle, it does not necessarily imply that all of the bi-reachable states are dead.

Live	Some reachable state from u is terminal.
Dead	All reachable states from u (including u) are closed and not terminal.
Unknown	u is closed, but not live or dead.
Open	u is not live and not closed.
Terminal	A state u labeled by $\texttt{T}(u)$.
Closed	A state u labeled by $\texttt{C}(u)$.
Canonical	A state x such that $\texttt{UF.find}(x) = x$.
u, v, w	States (may or may not be canonical).
x, y, z	Canonical states (i.e., states in the condensed graph).
Successor $\texttt{succ}(x)$	For an unknown, canonical state x, a uniquely chosen v such that (x, v) is an edge, and following the path of successors leads to an open state.

Fig. 3. *Top:* Basic classification of GID states into four disjoint categories. *Bottom:* Additional terminology used in this paper.

3 Algorithms

This section presents Algorithm 2, which solves the state classification problem in logarithmic time (Theorem 3); and Algorithm 3, an alternative lazy approach. Both algorithms are optimized versions of Algorithm 1, a first-cut algorithm which establishes the structure of our approach. We begin by establishing some basic terminology shared by all of the algorithms (see Figure 3).

States in a GID can be usefully classified as exactly one of four *statuses*: *live*, *dead*, *unknown*, or *open*, where *unknown* means "closed but not yet live or dead", and *open* means "not closed and not live". Note that a state may be live and neither open nor closed; this terminology keeps the classification disjoint. Pragmatically, for live states it does not matter if they are classified as open or closed, since edges from those states no longer have any effect. However, all dead and unknown states are closed, and no states are both open and closed.

Given this classification, the intuition is that for each unknown state u, we only need *one* path from u to an open state to prove that it is not dead; we want to maintain one such path for all unknown states. To maintain all of these paths simultaneously, we maintain an acyclic directed *forest* structure on unknown and open states where the roots are open states, and all non-root states have a single edge to another state, called its *successor*. Edges other than successor edges can be temporarily ignored, except for when marking live states; these are kept as *reserve* edges. Specifically, we add every edge (u, v) as a backward-edge from v (to allow propagating live states), but for edges not in the forest we keep (u, v) in a reserve list from u. We store all edges, including backward-edges, in the original order (u, v). The reserve list edge becomes relevant only when either (i) u is marked as closed, or (ii) u's successor is marked as dead.

In order to deal with cycles, we need to maintain the forest of unknown states not on the original graph, but on a union-find *condensed graph*, similar to [52]. When we find a cycle of unknown states, we *merge* all states in the cycle by calling the union method in the union-find. We refer to a state as *canonical* if

it is the canonical representative of its equivalence class in the union find; the condensed graph is a forest on canonical states. We use x, y, z to denote canonical states (states in the condensed graph), and u, v, w to denote the original states (not known to be canonical). Following [52], we maintain edges as linked lists rather than sets, and using the original states instead of canonical states; this is important as it allows combining edge lists in $O(1)$ time when merging states.

3.1 First-Cut Algorithm

Algorithm 1 is a first cut based on these ideas. The union-find data structure `UF` provides $\texttt{UF.union}(v_1, v_2)$, $\texttt{UF.find}(v)$, and $\texttt{UF.iter}(v)$: `UF.union` merges v_1 and v_2 to refer to the same canonical state, `UF.find` returns the canonical state for v, and `UF.iter` iterates over states equivalent to v. These use amortized $\alpha(n)$ for n updates, where $\alpha(n) \in o(\log n)$ is the inverse Ackermann function.

We will only merge states if they are bi-reachable from each other, and both unknown. This implies that all states equivalent to a state x have the same status. We also maintain the set of canonical states in `UF` as a set `X`.

The edge maps `res` and `bck` are stored as maps from `X` to linked lists of edges. Each edge (u, v) is always stored using its original states (i.e., edge labels are not updated when states are merged); but we can easily obtain the corresponding edge on canonical states via $(\texttt{UF.find}(u), \texttt{UF.find}(v))$.

Invariants. Algorithm 1 respects the following invariants. We don't care about forward or backward edges from live and dead states; we only need that $\texttt{status}(x)$ is `Live` or `Dead`, The *successor* and *no cycles* invariants are about the forest structure. Lastly, *edge representation* ensures that all edges in the input GID are represented somehow in the current graph.

- *Merge equivalence:* For all states u and v, if $\texttt{UF.find}(u) = \texttt{UF.find}(v)$, then u and v are bi-reachable and both closed. (This implies that u and v are both live, both dead, or both unknown.)
- *Status correctness:* For all u, $\texttt{status}(\texttt{UF.find}(u))$ equals the status of u.
- *Successor edges:* If x is unknown, then $\texttt{succ}(x)$ is defined and is an unknown or open state. If x is open, then $\texttt{succ}(x)$ is not defined.
- *No cycles:* There are no cycles among the set of edges $(x, \texttt{UF.find}(\texttt{succ}(x)))$, over all unknown and open canonical states x.
- *Edge representation:* For all edges (u, v) in the input GID, at least one of the following holds: (i) $(u, v) \in \texttt{res}(\texttt{UF.find}(v))$; (ii) $v = \texttt{succ}(\texttt{UF.find}(u))$; (iii) $\texttt{UF.find}(u) = \texttt{UF.find}(v)$; (iv) u is live; or (v) v is dead.

Theorem 1. *Algorithm 1 is correct.*

The main challenge of the proof (Appendix A.1) is analyzing $\textsc{OnClosed}(\texttt{C}(u))$. The procedure is recursive; on recursive calls, some states are *temporarily* marked `Open`, meaning they are roots in the forest structure. During these recursive calls, we need a slightly generalized invariant: each forest root corresponds to a pending future call to $\textsc{OnClosed}(\texttt{C}(u))$ (i.e., an element of `ToRecurse` for some call on

Algorithm 1 First-cut algorithm.

```
V: a type for states (integers) (variables u, v, ...)
E: the type of edges, equal to (V, V)
UF: a union-find data structure over V
X: the set of canonical states in UF (variables x, y, z, ...)
status: a map from X to Live, Dead, Unknown, or Open
succ: a map from X to V
res and bck: maps from X to linked lists of E
procedure OnEdge(E(u, v))
    x ← UF.find(u); y ← UF.find(v)
    if status(y) = Live then
        OnTerminal(T(x))                          ▷ mark x and its ancestors live
    else if status(x) ≠ Live then                 ▷ status(x) must be Open
        append (u, v) to res(x)
        append (u, v) to bck(y)
procedure OnTerminal(T(v))
    y ← UF.find(v)
    for all x in DFS backwards (along bck) from y not already Live do
        status(x) ← Live
        output Live(x′) for all x′ in UF.iter(x)
procedure OnClosed(C(v))
    y ← UF.find(v)
    if status(y) ≠ Open then return               ▷ y is already live or closed
    while res(y) is nonempty do
        pop (v, w) from res(y); z ← UF.find(w)
        if status(z) = Dead then continue
        else if CheckCycle(y, z) then
            for all z′ in cycle from z to y do z ← Merge(z, z′)
        else
            status(y) ← Unknown; succ(y) ← z;
            return
    status(y) ← Dead; output Dead(y′) for all y′ in UF.iter(y)
    ToRecurse ← ∅
    for all (u, v) in bck(y) do
        x ← UF.find(u)
        if status(x) = Unknown and UF.find(succ(x)) = y then
            status(x) ← Open          ▷ temporary – marked closed on recursive call
            add x to ToRecurse
    for all x in ToRecurse do OnClosed(C(x))
procedure CheckCycle(y, z) returning bool
    while status(z) = Unknown do z ← UF.find(succ(z))   ▷ get root state from z
    return y = z
procedure Merge(x, y) returning V
    z ← UF.union(x, y)
    bck(z) ← bck(x) + bck(y)                      ▷ O(1) linked list append
    res(z) ← res(x) + res(y)                      ▷ O(1) linked list append
    return z
```

Algorithm 2 Logarithmic time algorithm.

```
All data from Algorithm 1; succ: a map from X to E (instead of to V)
EF: Euler Tour Trees data structure providing: EF.add, EF.remove, and EF.connected
procedure OnEdge, Merge as in Algorithm 1
procedure OnTerminal(T(v))
    y ← UF.find(v)
    for all x in DFS backwards (along bck) from y not already Live do
        if status(x) = Unknown then
            (u, v) ← succ(x); delete succ(x); EF.remove(u, v)
        status(x) ← Live; output Live(x′) for all x′ in UF.iter(x)
procedure OnClosed(C(v))
    y ← UF.find(v)
    if status(y) ≠ Open then return
    while res(y) is nonempty do
        pop (v, w) from res(y); z ← UF.find(w)
        if status(z) = Dead then continue
        else if CheckCycle(y, z) then
            for all z′ in cycle from z to y do z ← Merge(z, z′)
        else
            status(x) ← Unknown; succ(x) ← (v, w)
            EF.add(v, w); return    ▷ undirected edge; use original labels (not (x, y))
    status(y) ← Dead; ToRec ← ∅; output Dead(y′) for all y′ in UF.iter(y)
    for all (u, v) in bck(y) do
        x ← UF.find(u)
        if status(x) = Unknown then
            (u′, v′) ← succ(x)
            if UF.find(v′) = y then
                EF.remove(u′, v′); status(x) ← Open; delete succ(x); add x to ToRec
    for all x in ToRec do OnClosed(C(x))
procedure CheckCycle(y, z) returning bool
    return EF.connected(y, z)
```

the stack). After we prove this (generalized) invariant, when OnClosed(C(u)) terminates, we know that there are no more temporary open states, and the forest structure implies that all closed states are correctly marked as unknown.

Complexity. The core inefficiency in Algorithm 1 — what we need to improve — lies in CheckCycle. The procedure repeatedly sets $z \leftarrow$ succ(z) to find the tree root, which in general could be linear time in the number of edges. For example, this inefficiency results in $O(m^2)$ work for a linear graph read in backwards order: E(2, 1), C(2), E(3, 2), C(3), ..., E(n, n-1), C(n). All other procedures use amortized $\alpha(m)$ time per update for m updates (Appendix A.2).

3.2 Logarithmic Algorithm

At its core, CheckCycle requires solving an *undirected* reachability problem on a graph that is restricted to a forest. However, the forest is changed not just by edge additions, but edge additions *and* deletions. While undirected reachability

and reachability in directed graphs are both difficult to solve incrementally, reachability in *dynamic forests* can be solved in $O(\log m)$ time per operation. Our algorithm uses an Euler Tour Forest data structure `EF` of Henzinger and King [27], and is shown in Algorithm 2.

However, this idea does not work straightforwardly – once again because of the presence of cycles in the original graph. We cannot simply store the forest as a condensed graph with edges on condensed states. As we saw in Algorithm 1, it was important to store successor edges as edges into `V`, rather than edges into `X` – this is the only way that we can merge states in $O(1)$, without actually inspecting the edge lists. If we needed to update the forest edges to be in `X`, this could require $O(m)$ work to merge two $O(m)$-sized edge lists as each edge might need to be relabeled in the `EF` graph.

To solve this challenge, we instead store the `EF` data structure on the original states, rather than the condensed graph; but we ensure that *each canonical state is represented by a tree of original states*. When adding edges between canonical states, we need to make sure to remember the original label (u, v), so that we can later remove it using the original labels (this happens when its target becomes dead). When an edge would create a cycle, we instead simply ignore it in the `EF` graph, because a line of connected trees forms a tree.

Summary and invariants. In summary, the algorithm reuses the data, procedures, and invariants from Algorithm 1, with the following important changes: (1) We maintain the EF data structure `EF`, a forest on `V`. (2) The successor edges are stored as their original edge labels (u, v), rather than just as a target state. (3) The procedure OnClosed is rewritten to maintain the graph `EF`. (4) The *successor edges* and *no cycles* invariants use the new `succ` representation: that is, they are constraints on the edges $(x, \texttt{UF.find}(v))$, where $\texttt{succ}(x) = (u, v)$. (5) We add the following two constraints on edges in `EF`, depending on whether those states are equivalent in the union-find structure. Using these invariants, the full proof of Theorem 2 can be found in Appendix A.3.

- *EF inter-edges:* For all *inequivalent* u, v, (u, v) is in the EF if and only if $(u, v) = \texttt{succ}(\texttt{UF.find}(u))$ or $(v, u) = \texttt{succ}(\texttt{UF.find}(v))$.
- *EF intra-edges:* For all unknown canonical states x, the set of edges (u, v) in the EF between states belonging to x forms a tree.

Theorem 2. *Algorithm 2 is correct.*

Theorem 3. *Algorithm 2 uses amortized logarithmic time per edge update.*

Proof. By the analysis of Algorithm 1, each line of the algorithm runs amortized $O(1)$ time other than those in CheckCycle. For the CheckCycle, procedure it now avoids the loop and so also runs amortized $O(1)$ times. Each line is either constant-time, $\alpha(m) = o(\log n)$ time for the `UF` calls, or $O(\log n)$ time for the `EF` calls, so in total the algorithm runs in amortized $O(\log n)$ time per update. □

Algorithm 3 Lazy algorithm.

All data from Algorithm 1; `jumps`: a map from `X` to lists of `V`
procedure OnEdge, OnTerminal OnClosed as in Algorithm 1
procedure CheckCycle(y, z) **returning** `bool`
  return $y =$ GetRoot(z)
procedure GetRoot(z) **returning** `V`
  if `status`$(z) =$ `Open` **then return** z
  if `jumps`(z) is empty **then** push `succ`(z) to `jumps`(z) ▷ set 0th jump
  repeat pop w from `jumps`(z); $z' =$ `UF.find`(w) ▷ remove dead jumps
  until `status`$(z') \neq$ `Dead`
  push z' to `jumps`(z); result $\leftarrow$ GetRoot(z')
  $n \leftarrow$ `length`(`jumps`(z)); $n' \leftarrow$ `length`(`jumps`(z'))
  if $n \leq n'$ **then** push `jumps`$(z')[n-1]$ to `jumps`(z) ▷ set nth jump
  return result
procedure Merge(x, y) **returning** `V`
  $z \leftarrow$ UF.union(x, y)
  `bck`$(z) \leftarrow$ `bck`$(x) +$ `bck`(y); `res`$(z) \leftarrow$ `res`$(x) +$ `res`(y)
  `jumps`$(z) \leftarrow$ `empty`; **return** z

3.3 Lazy Algorithm

While the asymptotic complexity of $\log n$ could be the end of the story, in practice, the cost of the `EF` calls could be a significant overhead. The technical details of Euler-Tour Trees include building an AVL-tree cycle for each tree, where the cycle contains each state of the graph and each edge in the graph twice. It turns out that adding *one edge* to `EF` results in no less than *eight* modifications to the AVL tree: it needs to be split at the source, split at the target, then the edge needs to be added in both directions (u, v) and (v, u) to the cycle, and then these trees need to be glued together. Every one of these operations comes with a rebalancing operation which could do $\Omega(\log n)$ tree rotations and pointer dereferences to visit the nodes in the AVL tree.

As a result of these overheads, in this section, we investigate a simpler, lazy algorithm which avoids `EF` and directly optimizes Algorithm 1. For this, one idea in the right direction is to store for each state a direct pointer to the root which results from repeatedly calling `succ`. But there are two issues with this. First, maintaining this may be difficult (when the root changes, potentially updating a linear number of root pointers). Second, the root may be marked dead, in which case we have to re-compute all pointers to that root.

Instead, we introduce a *jump list* from each state: intuitively, it will contain states after calling successor once, twice, four times, eight times, and so on at powers of two; and it will be updated lazily, at most once for every visit to the state. When a jump becomes obsolete (the target dead), we just pop off the largest jump, so we do not lose all of our work in building the list. We maintain the following additional information: for each unknown canonical state x, a nonempty list of *jumps* $[v_0, v_1, v_2, \ldots, v_k]$, such that v_0 is reachable from x, v_1 is reachable from v_0, v_2 is reachable from v_1, and so on, and $v_1 =$ `succ`(x).

The resulting algorithm is shown in Algorithm 3. The key procedure is GetRoot(z), which is called when adding a reserve edge (y, z) to the graph. The correctness of the algorithm is proven in Appendix A.4. Beyond correctness, the jump list also satisfies a *powers of two* invariant that states that on the path of canonical states from v_0 to v_i, the total number of states (including all states in each equivalence class) is at least 2^i. While this is not necessary for correctness, it is the key to the algorithm's practical efficiency: it follows from this that *if* the jump list is fully saturated for every state, querying GetRoot(z) will take only logarithmic time. However, because the jump lists are updated lazily, this does not establish an asymptotic complexity for the algorithm.

Theorem 4. *Algorithm 3 is correct.*

4 Experimental Evaluation

The primary goal of our evaluation has been to experimentally validate the performance of GIDs as a data structure in isolation, rather than their use in a particular application. Our evaluation seeks to answer the following questions:

Q1 How does our approach (Algorithms 2 and 3) compare to the state-of-the-art approach based on maintaining SCCs?
Q2 How does the performance of the studied algorithms vary when the class of input graphs changes (e.g., sparse vs. dense, structured vs. random)?
Q3 Finally, how do the studied algorithms perform on GIDs taken from the example application to regexes described in Section 5?

To answer **Q1**, we put substantial implementation effort into a common framework on which a fair comparison could be made between different approaches. To this end, we implemented GIDs as a data structure in Rust which includes a graph data structure on top of which all algorithms are built. In particular, this equalizes performance across algorithms for the following baseline operations: state and edge addition and retrieval, DFS and BFS search, edge iteration, and state merging. We chose Rust for our implementation for its performance, and because there does not appear to be an existing publicly available implementation of BFGT in any other language. The number of lines of code used to implement these various structures is summarized in Figure 4. We implement Algorithms 2 and 3 and compare them with the following baselines:

BFGT The state-of-the-art approach based on SCC maintenance, using worst-case amortized $O(\sqrt{m})$ time per update [4].
Simple A simpler version of BFGT that uses a forward-DFS to search for cycles. Like Algorithm 1, it can take $\Theta(m^2)$ in the worst case.
Naïve A greedy upper bound for all approaches which re-computes the entire set of dead states using a linear-time DFS after each update.

To answer **Q2**, first, we compiled a range of basic graph classes which are designed to expose edge case behavior in the algorithms, as well as randomly

Implementation Component	LoC
Common Framework	1197
Naïve Algorithm	78
Simple Algorithm	98
BFGT Algorithm	265
Algorithm 2 (Ours)	253
Algorithm 3 (Ours)	283
Euler Tour Trees	1510
Experimental Scripts	556
Separated Unit Tests	798
Util	217
Other	69
Total	5324

Category	Benchmark	Source	Qty
Basic	Line		24
	Cycle		24
	Complete		18
	Bipartite		14
	Total		80
Random	Sparse		260
	Dense		130
	Total		390
Regex	RegExLib [9,49]	2061	37
	Handwritten [50]	70	26
	Additional		11
	Total		74

Fig. 4. *Left:* Lines of code for each algorithm and other implementation components. *Right:* Benchmark GIDs used in our evaluation. Where present, the source column indicates the quantity prior to filtering out trivially small graphs.

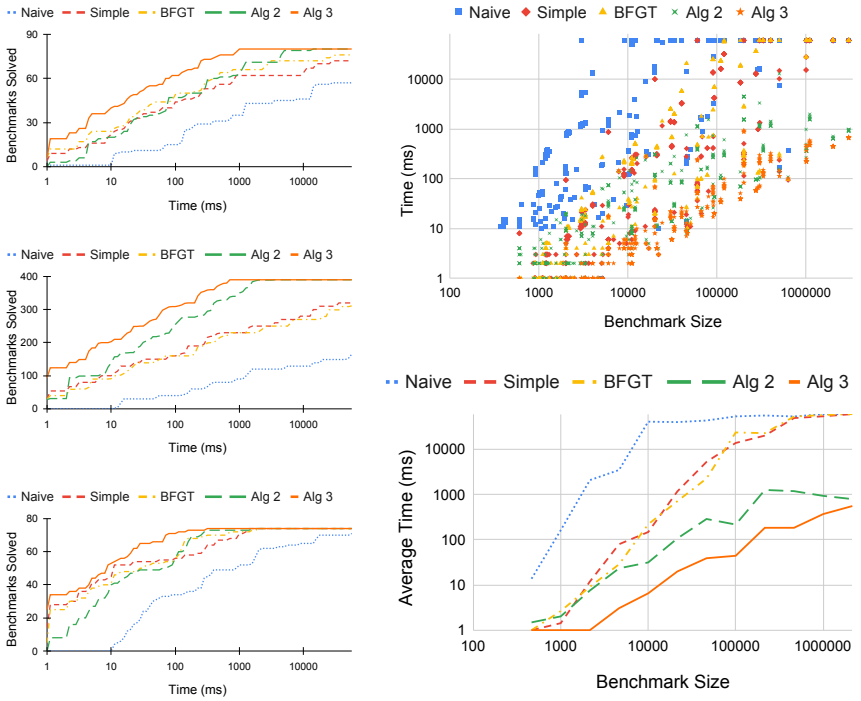

Fig. 5. Evaluation results. *Left:* Cumulative plot showing the number of benchmarks solved in time t or less for basic GID classes (top), randomly generated GIDs (middle), and regex-derived GIDs (bottom). *Top right:* Scatter plot showing the size of each benchmark vs time to solve. *Bottom right:* Average time to solve benchmarks of size closest to s, where values of s are chosen in increments of 1/3 on a log scale.

generated graphs. We focus on graphs with no live states, as live states are treated similarly by all algorithms. Most of the generated graphs come in $2 \times 2 = 4$ variants: (i) the states are either read in a forwards- or backwards- order; and (ii) they are either *dead* graphs, where there are no open states at the end and so everything gets marked dead; or *unknown* graphs, where there is a single open state at the end, so most states are unknown. In the unknown case, it is sufficient to have one open state at the end, as many open states can be reduced to the case of a single open state where all edges point to that one. We include GIDs from line graphs and cycle graphs (up to 100K states in multiples of 3); complete and complete acyclic graphs (up to 1K states); and bipartite graphs (up to 1K states). These are important cases, for example, because the reverse-order line and cycle graphs are a potential worst case for Simple and BFGT.

Second, to exhibit more dynamic behavior, we generated random graphs: sparse graphs with a fixed out-degree from each state, chosen from $1, 2, 3$, or 10 (up to 100K states); and dense graphs with a fixed probability of each edge, chosen from .01, .02, or .03 (up to 10K states). Each case uses 10 different random seeds. As with the basic graphs, states are read in some order and marked closed.

To answer **Q3**, we wrote a backend to extract a GID at runtime from Z3's regex solver [50]. While the backend of the solver is essentially a GID — and so could be passed to our GID implementation dynamically — this setup includes many extraneous overheads, including rewriting expressions and computing derivatives. This makes it difficult to isolate the performance of the GID data structure itself, which is the sole focus of this paper. We therefore instrumented the Z3 solver code to export the (incremental) sequence of graph updates that would be performed during a run of Z3 on existing regex benchmarks. For each regex benchmark, this instrumented code produces a faithful representation of the sequence of graph updates that actually occur in a run of the SMT solver on this particular benchmark. The benchmarks focus on *extended* regexes, rather than plain classical regexes as these are the ones for which dead state detection is relevant (see Section 5). For each regex benchmark, we thus get a GID benchmark for the present paper. We include GIDs for the RegExLib benchmarks [9,49] and the handcrafted Boolean benchmarks reported in [50]. We add to these 11 additional examples designed to be difficult GID cases.

From both the Q2 and Q3 benchmarks, we filter out any benchmark which takes under 10 milliseconds for all of the algorithms to solve (including Naïve), and we use a 60 second timeout. The evaluation was run on a 2020 MacBook Air (MacOS Monterey) with an Apple M1 processor and 8GB of memory.

Correctness. To ensure that all of our implementations our correct, we invested time into unit testing and checked output correctness on all of our collected benchmarks, including several cases which exposed bugs in previous versions of one or more algorithms. In total, all algorithms are vetted against 25 unit tests from handwritten edge cases that exposed prior bugs, 373 unit tests from benchmarks, and 30 module-level unit tests for specific functions.

Results. Figure 5 shows the results. Algorithm 3 shows significant improvements over the state-of-the-art, solving more benchmarks in a smaller amount

of time across basic GIDs, random GIDs, and regex GIDs. Algorithm 2 also shows state-of-the-art performance, similar to BFGT on basic and regex GIDs and significantly better on random GIDs. On the bottom right, since looking at average time is not meaningful for benchmarks of widely varying size, we stratify the size of benchmarks into buckets, and plot time-to-solve as a function of size. Both x-axis and y-axis are on a log scale. The plot shows that Algorithm 3 exhibits up to two orders of magnitude speedup over BFGT for larger GIDs – we see speedups of 110x to 530x for GIDs in the top five size buckets (GIDs of size nearest to 100K, 200K, 500K, 1M, and 2M).

New implementations of existing work. Our implementation contributes, to our knowledge, the first implementation of BFGT for SCC maintenance. In addition, it is one of the first implementations of Euler Tour Trees, including the AVL tree backing for tours, and likely the first implementation in Rust.

5 Application to Extended Regular Expressions

In this section, we explain how precisely the GID state classification problem arises in the context of derivative-based solvers [50,34]. We first define *extended* regexes (regexes extended with intersection `&` and complement `~`) modulo a symbolic alphabet $\mathcal{A}$ of *predicates* that represent sets of characters. We explain the main idea behind *(symbolic) derivatives* [50] that provides the foundation for incrementally creating a GID. Then we show, through an example, how a solver can incrementally expand derivatives to reduce the satisfiability problem to the GID state classification problem (Definition 2).

Define a *regex* by the following grammar, where $\varphi \in \mathcal{A}$ denotes a predicate:

$$RE ::= \varphi \mid \varepsilon \mid RE_1 \cdot RE_2 \mid RE^* \mid RE_1 \,|\, RE_2 \mid RE_1 \,\&\, RE_2 \mid \sim RE$$

Let R^k represent the concatenation of R k times. A regex R is *nullable* if it matches the empty string. Nullability can be computed inductively over the structure of regexes: for example, ε and R^* are nullable, and $R_1 \,\&\, R_2$ is nullable iff both R_1 and R_2 are nullable. **Terminal** states are the nullable regexes.

A *symbolic derivative* $\delta(R)$ of a regex R (defined formally in Appendix A.5) is a binary tree whose leaves are regexes and internal nodes are labeled by predicates from $\mathcal{A}$. A binary tree with root node labeled φ and two immediate subtrees t_1 and t_2 is written $(\varphi \,?\, t_1 : t_2)$; it means that φ is *true* in t_1 and *false* in t_2. A branch with the *accumulated branch condition* ψ from the root of t to one of its leaves R' defines a transition $R \xrightarrow{\psi} R'$ – *provided* ψ *is satisfiable in* $\mathcal{A}$ – and the edge (R, R') is added to the GID. The number of leaves of $\delta(R)$ is the *out-degree* $deg^+(R)$ of R and is independent of the size of the concrete alphabet. Thus, we label R as **closed** when $deg^+(R)$ edges have been added from R.

In summary, to apply Definition 1 to regexes, states are regexes, and edges are transitions to their derivatives. A **live** state here is thus a regex that reaches a nullable regex via 0 or more edges. This implies that there exists a concrete string matching it. Conversely, **dead** states are always empty, i.e. they match no

strings, but can reach other dead states, creating strongly connected components of closed states none of which are live. For example, the *false* predicate $\bot$ of $\mathcal{A}$ serves as the regex that matches *nothing* and is a trivially *dead* state. Thus $\tilde{}\bot$ is equivalent to $.^*$, where $.$ is the *true* predicate and is a trivially *live* state.

5.1 Reduction from Incremental Regex Emptiness to GIDs

For simplicity, suppose we want to determine the satisfiability of a single regex constraint $s \in R$, where s is a string variable and R is a concrete regex. (This is not overly restrictive – any number of simultaneous regex constraints for a string s can be combined into single regex constraint by using the Boolean operations of regexes.) For example, let $L = \tilde{}(.^*\alpha.^{100})$ and $R = L \,\&\, (.\alpha)$, where α is the "is digit" predicate that is true of characters that are digits (often denoted `\d`). The solver manipulates regex membership constraints on strings by unfolding them [50]. The constraint $s \in R$, that essentially tests nonemptiness of R with s as a witness, becomes

$$(s = \epsilon \wedge \mathit{Nullable}(R)) \vee (s \neq \epsilon \wedge s_{1..} \in \delta_{s_0}(R))$$

where, $s \neq \epsilon$ since R is not nullable, $s_{i..}$ is the suffix of s from index i, and

$$\delta(R) = \delta(L) \mathbin{\&\!\&} \delta(.\alpha) = (\alpha\,?\,L \,\&\, \tilde{}(.^{100})\,:\,L) \mathbin{\&\!\&} \alpha = (\alpha\,?\,L \,\&\, \tilde{}(.^{100}) \,\&\, \alpha\,:\,L \,\&\, \alpha)$$

Let $R_1 = L \,\&\, \tilde{}(.^{100}) \,\&\, \alpha$ and $R_2 = L \,\&\, \alpha$. So R has two outgoing transitions $R \xrightarrow{\alpha} R_1$ and $R \xrightarrow{\neg\alpha} R_2$ that contribute the edges (R, R_1) and (R, R_2) into the GID. Note that these edges depend only on R and not on s_0.

We continue the search incrementally by checking the two branches of the if-then-else constraint, where R_1 and R_2 are again not nullable (so $s_{1..} \neq \epsilon$):

$$\begin{aligned}
&s_0 \in \alpha \wedge s_{2..} \in \delta_{s_1}(R_1) \quad \vee \quad s_0 \in \neg\alpha \wedge s_{2..} \in \delta_{s_1}(R_2)\\
&\delta(R_1) = (\alpha\,?\,L \,\&\, \tilde{}(.^{100}) \,\&\, \tilde{}(.^{99})\,:\,L \,\&\, \tilde{}(.^{99})) \mathbin{\&\!\&} (\alpha\,?\,\varepsilon\,:\,\bot) = (\alpha\,?\,\varepsilon\,:\,\bot)\\
&\delta(R_2) = (\alpha\,?\,L \,\&\, \tilde{}(.^{100})\,:\,L) \mathbin{\&\!\&} (\alpha\,?\,\varepsilon\,:\,\bot) = (\alpha\,?\,\varepsilon\,:\,\bot)
\end{aligned}$$

It follows that $R_1 \xrightarrow{\alpha} \varepsilon$ and $R_2 \xrightarrow{\alpha} \varepsilon$, so the edges (R_1, ε) and (R_2, ε) are added to the GID where ϵ is a trivial terminal state. In fact, after R_1 the search already terminates because we then have the path $(R, R_1)(R_1, \epsilon)$ that implies that R is live. The associated constraints $s_0 \in \alpha$ and $s_1 \in \alpha$ and the final constraint that $s_{2..} = \epsilon$ can be used to extract a concrete witness, e.g., $s =$ `"42"`.

Soundness of the algorithm follows from that if R is nonempty ($s \in R$ is *satisfiable*), then we eventually arrive at a nullable (terminal) regex, as in the example run above. To achieve *completeness* – and to eliminate dead states as early as possible – we incrementally construct a GID corresponding to the set of regexes seen so far (as above). After all the feasible transitions from R to its derivatives in $\delta(R)$ are added to the GID as edges (WLOG in one batch), the state R becomes closed. *Crucially, due to the* ***symbolic form*** *of $\delta(R)$, no derivative is missing.* Therefore R is known to be empty precisely as soon as R is

detected as a dead state in the GID. An additional benefit is that the algorithm is independent of the size of the universe of $\mathcal{A}$, that may be very large (e.g. the Unicode character set), or even infinite. We get the following theorem that uses finiteness of the closure of symbolic derivatives [50, Theorem 7.1]:

Theorem 5. *For any regex R, (1) If R is nonempty, then the decision procedure eventually marks R live. (2) If R is empty, then the decision procedure marks R dead at the earliest stage that it is know to be dead, and terminates.*

6 Related Work

Incremental graph algorithms. Online graph algorithms are typically divided into problems over *incremental* graphs (where edges are added), *decremental* graphs (where edges are deleted), and *dynamic* graphs (where edges are both added and deleted), with core data structures discussed in [39,21]. For incremental directed graphs, important problems include *transitive closure*, *cycle detection*, *topological ordering*, and *strongly connected component (SCC) maintenance*. Maintaining a topological order of dynamic DAGs is studied in [35]. An online algorithm for topological sorting that is experimentally shown to be preferable for *sparse* graphs is discussed in [46], and in a related article [45] it is also discussed how to extend such an algorithm to detect strongly connected components. One major application for these algorithms is in pointer analysis [44]. Topological order of incremental DAGs is studied in [25], presenting two different algorithms, one for *sparse graphs* and one for *dense graphs* – the algorithms are also extended to work with SCCs. The sparse algorithm was subsequently simplified in [4] and is the basis of our implementation named BFGT in Section 4. A unified approach of several algorithms based on [4] is presented in [14] that uses a notion of *weak topological order* and a labeling technique that estimates transitive closure size. Further extensions of [4] are studied in [6,8] based on randomization. Transitive closure of *decremental* DAGs is studied in [47] that improve upon some algorithms presented earlier in [26].

Data structures for SMT. *UnionFind* [52] is a foundational data structure in SMT. *E-graphs* [54,17,19,42] are used to ensure *functional extensionality*, where two expressions are equivalent if their subexpressions are equivalent. In both cases the maintained relation is an *equivalence* relation. In contrast, maintaining live and dead states involves tracking reachability rather than equivalence. While reachability is a basic problem in many core SMT algorithms, to the best of our knowledge, the formulation of the problem we consider here is new.

Dead state elimination in automata. A DFA or NFA may be viewed as a GID, so state classification in GIDs solves dead state elimination in DFAs and NFAs, while additionally allowing for incremental state updates. Dead state elimination is also known as *trimming* [28] and plays an important role in automata *minimization* [7,29,43,38] (see also [31,40,30,10,11]; for the symbolic case, [16]). The *incremental* minimization algorithm [53] can be halted at any point and produce a partially minimal DFA which is equivalent to the starting DFA, but here the DFA itself is not incremental – the DFA does not change.

References

1. Abboud, A., Williams, V.V.: Popular conjectures imply strong lower bounds for dynamic problems. In: 2014 IEEE 55th Annual Symposium on Foundations of Computer Science. pp. 434–443. IEEE (2014)
2. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994, https://doi.org/10.23919/FMCAD.2018.8602994
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: Cvc4. In: International Conference on Computer Aided Verification. pp. 171–177. Springer (2011)
4. Bender, M.A., Fineman, J.T., Gilbert, S., Tarjan, R.E.: A new approach to incremental cycle detection and related problems. ACM Transactions on Algorithms **12**(2), 14:1–14:22 (Dec 2015). https://doi.org/10.1145/2756553, http://arxiv.org/abs/1112.0784
5. Bernstein, A., Chechi, S.: Incremental topological sort and cycle detection in expected total time. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 21–34. SIAM (2018)
6. Bernstein, A., Chechik, S.: Incremental topological sort and cycle detection in o(m∗sqrt(n)) expected total time. In: Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 21–34. SODA'18, Society for Industrial and Applied Mathematics (2018)
7. Berstel, J., Boasson, L., Carton, O., Fagnot, I.: Minimization of automata (2011), handbook of Automata
8. Bhattacharya, S., Kulkarni, J.: An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 2509–2521. SIAM (2020)
9. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. In: Fontaine, P., Goel, A. (eds.) SMT'12. pp. 76–86 (2012)
10. Blum, N.: An $0(n \log n)$ implementation of the standard method for minimizing n-state finite automata. Information Processing Letters **57**, 65–69 (1996)
11. Brzozowski, J.A.: Canonical regular expressions and minimal state graphs for definite events. In: Proc. Sympos. Math. Theory of Automata. pp. 529—-561. New York (1963)
12. Caron, P., Champarnaud, J.M., Mignot, L.: Partial derivatives of an extended regular expression. In: Language and Automata Theory and Applications, LATA 2011. LNCS, vol. 6638, pp. 179–191. Springer (2011)
13. Clarke, E., Grumberg, O., Hamaguchi, K.: Another look at LTL model checking. In: International Conference on Computer Aided Verification. pp. 415–427. Springer (1994)
14. Cohen, E., Fiat, A., Kaplan, H., Roditty, L.: A Labeling Approach to Incremental Cycle Detection. arXiv e-prints (Oct 2013)
15. CVC4: (2020), https://github.com/CVC4/CVC4
16. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. ACM SIGPLAN Notices – POPL'14 **49**(1), 541–553 (2014). https://doi.org/10.1145/2535838.2535849

17. De Moura, L., Bjørner, N.: Efficient e-matching for smt solvers. In: International Conference on Automated Deduction. pp. 183–198. Springer (2007)
18. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Communications of the ACM **54**(9), 69–77 (2011)
19. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. Journal of the ACM (JACM) **27**(4), 758–771 (1980)
20. Ellul, K., Krawetz, B., Shallit, J., Wang, M.W.: Regular expressions: New results and open problems. J. Autom. Lang. Comb. **10**(4), 407–437 (2005)
21. Eppstein, D., Galil, Z., Italiano, G.F.: Dynamic graph algorithms. Algorithms and theory of computation handbook **1**, 9–1 (1999)
22. Fan, W., Hu, C., Tian, C.: Incremental graph computations: Doable and undoable. In: Proceedings of the 2017 ACM International Conference on Management of Data. pp. 155–169 (2017)
23. Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. arXiv preprint arXiv:0802.2869 (2008)
24. Haeupler, B., Kavitha, T., Mathew, R., Sen, S., Tarjan, R.E.: Incremental cycle detection, topological ordering, and strong component maintenance. ACM Transactions on Algorithms (TALG) **8**(1), 1–33 (2012)
25. Haeupler, B., Kavitha, T., Mathew, R., Sen, S., Tarjan, R.E.: Incremental cycle detection, topological ordering, and strong component maintenance. ACM Transactions on Algorithms **8**(1.3), 1–33 (January 2012). https://doi.org/10.1145/2071379.2071382
26. Henzinger, M., King, V.: Fully dynamic biconnectivity and transitive closure. In: Proceedings of the 36th Annual Symposium on Foundations of Computer Science. pp. 664–672. Milwaukee, WI (1995)
27. Henzinger, M.R., King, V.: Randomized fully dynamic graph algorithms with polylogarithmic time per operation. Journal of the ACM (JACM) **46**(4), 502–516 (1999)
28. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (1979)
29. Hopcroft, J.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Kohavi, Z. (ed.) Theory of machines and computations, Proc. Internat. Sympos., Technion, Haifa, 1971. pp. 189–196. Academic Press, New York (1971)
30. Hopcroft, J.E., Ullman, J.D.: Formal languages and their relation to automata. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1969)
31. Huffman, D.: The synthesis of sequential switching circuits. Journal of the Franklin Institute **257**(3–4), 161–190,275–303 (1954)
32. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3), 291–314 (2001)
33. programming language., T.R.: (2020), https://www.rust-lang.org/
34. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings? In: FroCoS 2015: Frontiers of Combining Systems. LNCS, vol. 9322, pp. 135–150. Springer (2015)
35. Marchetti-Spaccamela, A., Nanni, U., Rohnert, H.: Maintaining a topological order under edge insertions. Information Processing Letters **59**(1), 53–58 (1996). https://doi.org/https://doi.org/10.1016/0020-0190(96)00075-0, https://www.sciencedirect.com/science/article/pii/0020019096000750
36. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning sat solvers. In: Handbook of satisfiability, pp. 131–153. ios Press (2009)
37. Matsakis, N.D., Klock, F.S.: The rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014)

38. Mayr, R., Clemente, L.: Advanced automata minimization. In: POPL'13. pp. 63–74 (2013)
39. Mehlhorn, K.: Data Structures and Algorithms, Graph Algorithms and NP-Completeness, vol. 2. Springer (1984)
40. Moore, E.F.: Gedanken-experiments on sequential machines. Automata studies, Annals of mathematics studies pp. 129—153 (1956)
41. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS'08. pp. 337–340. LNCS, Springer (2008)
42. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM (JACM) **27**(2), 356–364 (1980)
43. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM Journal on Computing **16**(6), 973–989 (1987)
44. Pearce, D.J.: Some directed graph algorithms and their application to pointer analysis. Ph.D. thesis, Imperial College, London (2005)
45. Pearce, D.J., Kelly, P.H.J.: A dynamic algorithm for topologically sorting directed acyclic graphs. In: Proceedings of the Workshop on Efficient and experimental Algorithms (WEA). LNCS, vol. 3059, pp. 383–398. Springer (2004)
46. Pearce, D.J., Kelly, P.H.J.: A dynamic topological sort algorithm for directed acyclic graphs. ACM Journal of Experimental Algorithmics **11**(1.7), 1–24 (2006)
47. Roditty, L., Zwick, U.: Improved dynamic reachability algorithms for directed graphs. SIAM Journal on Computing **37**(5), 1455–1471 (2008). https://doi.org/10.1137/060650271
48. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: International SPIN Workshop on Model Checking of Software. pp. 149–167. Springer (2007)
49. SMT: (2012), https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/nbjorner-microsoft.automata.smtbenchmarks.zip
50. Stanford, C., Veanes, M., Bjørner, N.: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 620–635 (2021)
51. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time (preliminary report). In: Proceedings of the fifth annual ACM symposium on Theory of computing. pp. 1–9 (1973)
52. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. JACM **22**, 215–225 (1975)
53. Watson, B.W., Daciuk, J.: An efficient incremental DFA minimization algorithm. Nat. Lang. Eng. **9**(1), 49–64 (Mar 2003). https://doi.org/10.1017/S1351324903003127, http://dx.doi.org/10.1017/S1351324903003127
54. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: fast and extensible equality saturation. Proceedings of the ACM on Programming Languages **5**(POPL), 1–29 (2021)
55. Z3: (2020), http://research.microsoft.com/projects/z3

A Appendix

A.1 Proof of Theorem 1 (Section 3)

Proof. We need to argue that all of the invariants described above are preserved. This implies correctness of the algorithm by the status correctness invariant, since it implies that live and dead states are labeled (and output) correctly.

Upon receiving $\texttt{E}(u, v)$ or $\texttt{T}(u)$, this may cause some dead, unknown, or open states to change status to live, but does not change the status of any other states. As `bck` stores all backwards-edges (not just `succ` edges), live states are marked correctly. This preserves the forest invariants because if an unknown or open state is marked live, so are all its predecessors. This also preserves the edge representation invariant, either by adding new edges to case (i) and case (iv) (for `E` updates) or *moving* edges from cases (i)-(iii) to case (iv) (for both `E` and `T` updates).

The procedure $\texttt{C}(u)$ is recursive; during the procedure and on recursive calls, some states are *temporarily* marked `Open`, meaning they are roots in the forest structure. During these recursive calls, we need a slightly generalized invariant: each forest root corresponds to a pending future call to $\textsc{OnClosed}(\texttt{C}(u))$ (i.e., an element of `ToRecurse` for some call on the stack) and is a state that needs to either find a successor or be marked dead. This means that the `status` labels for unknown- and open-labeled states are not necessarily correct during recursive calls; we are only concerned with preserving the forest structure, so that they will be correct after all calls complete. Note in particular that after merging cycles of unknown states via $\textsc{Merge}(x, y)$, the new root is marked `Open` to preserve the generalized invariant on recursive calls.

Upon receiving $\texttt{C}(u)$, without loss of generality we may assume $\texttt{status}(u) = \texttt{Open}$ (the opposite only occurs if the input GID has duplicate $\texttt{C}(u)$ updates, or if $\texttt{C}(u)$ occurs for a live state). At the top of the while loop, there are three cases:

- If a reserve edge is available, and its target is dead, then we discard it. This preserves edge representation because that edge still satisfies (iv).
- If a reserve edge is available and its target is not dead, then we see if adding that edge creates a cycle by calling `CheckCycle`. If no, then the two trees are distinct, so we add an edge between them. This preserves edge representation by moving that edge from case (i) to case (ii). If yes, then we collapse the cycle in the forest to a single state by repeated calls to $\textsc{Merge}(x, y)$. This preserves edge representation by moving the edges in the cycle from case (ii) to case (iii). The forest structure of the `succ` is also preserved because if a graph is a forest, then it remains a forest after merging adjacent states.
- Finally, if there are no reserve edges left, then because of edge representation, and because there is no successor from u, all edges from u must lead to dead states (case (v)) and therefore u is dead. This case splits the tree rooted at u into one tree for each of its predecessors, preserving the forest structure. Each of the `succ` edges from predecessors are deleted; this preserves edge representation by moving those edges from case (ii) to case (v).

In any case, the generalized invariant for recursive calls is preserved, and all dead states are labeled correctly (given the invariant), so we are done. □

A.2 Complexity of Algorithm 1 (Section 3)

Besides CheckCycle, all other parts of the algorithm run in amortized $\alpha(m)$ time per update for m updates (using vectors to represent the maps `fwd`, `bck`,

and `succ` for $O(1)$ lookups). The OnTerminal calls and loop iterations only run once per edge in the graph when the target of that edge is marked live or terminal. Likewise, the procedure OnClosed is called conservatively: the cost of each call can be assigned *either* to the target of an edge being marked dead, *or* to an edge being merged as part of a cycle. Both marking dead and merging can only happen once for a given edge, so this is $O(1)$ per edge.

A.3 Proof of Theorem 2 (Section 3)

Proof. Observe that the `EF` inter-edges constraint implies that `EF` only contains edges between unknown and open states, together with isolated trees. In the modified OnTerminal procedure, when marking states as live we remove inter-edges, so we preserve this invariant.

Next we argue that given the invariants about `EF`, for an *open* state y the CheckCycle procedure returns true if and only if (y, z) would create a directed cycle. If there is a cycle of canonical states, then because canonical states are connected trees in `EF`, the cycle can be lifted to a cycle on original states, so y and z must already be connected in this cycle without the edge (y, z). Conversely, if y and z are connected in `EF`, then there is a path from y to z, and this can be projected to a path on canonical states. However, because y is open, it is a root in the successor forest, so any path from y along successor edges travels only on backwards edges; hence z is an ancestor of y in the *directed* graph, and thus (y, z) creates a directed cycle.

This leaves the OnClosed procedure. Other than the `EF` lines, the structure is the same as in Algorithm 1, so the previous invariants are still preserved, and it remains to check the `EF` invariants. When we delete the successor edge and temporarily mark $\texttt{status}(x) = \texttt{Open}$ for recursive calls, we also remove it from `EF`, preserving the inter-edge invariant. Similarly, when we add a successor edge to x, we add it to `EF`, preserving the inter-edge invariant. So it remains to consider when the set of canonical states changes, which is when merging states in a cycle. Here, a line of canonical states is merged into a single state, and a line of connected trees is still a tree, so the intra-edge invariant still holds for the new canonical state, and we are done. □

A.4 Proof of Theorem 4

Invariants. In addition to all invariants from Algorithm 1, we maintain the following invariants for *every* unknown canonical state x, where $\texttt{jumps}(x)$ is a list of states $v_0, v_1, v_2, \ldots, v_k$.

- *First jump:* if the jump list is nonempty, then $v_0 = \texttt{succ}(v)$.
- *Reachability:* v_{i+1} is reachable from v_i for all i.
- *Powers of two:* on the path of canonical states from v_0 to v_i, the total number of states (including all the states in each equivalence class) is at least 2^i.

Proof. We use the invariant on the jump list mentioned: that v_1 is the successor, v_2 is reachable from v_1, v_3 is reachable from v_2, and so on, using only forward edges added to the graph (not reserve edges). I.e., $v_1, v_2, \ldots$ is some sublist of the states along the path from an unknown state to its root, potentially followed by some dead states. We need to argue that the subprocedure GetRoot (i) receives the same verdict as repeatedly calling `succ` to find a cycle in the first-cut algorithm and (ii) preserves the jump list invariant. Popping dead states from the jump list clearly preserves the invariant, as does adding on a state along the path to the root, which is done when $k' \geq k$. Merging states preserves the jump list invariant trivially because we throw the jump list away, and marking states live preserves the jump list invariant trivially since the jump list is only maintained and used for unknown states. Finally, marking states as closed initializes the jump list correctly assuming that the successor is calculated correctly. □

A.5 Definition of Symbolic Derivatives (Section 5)

The formal definition of a symbolic derivative is as follows where the operations $\bigoplus$, $\mathbin{\&\!\!\&}$, $\overset{\frown}{\ }$ and $\odot$ are here for the purposes of this paper and w.l.o.g. *normalizing* variants of `|`, `&`, `~` and $\cdot$, by distributing the operations into if-then-elses and build a single binary tree as a nested if-then-else as a result (see [50] for details):

$$\begin{array}{ccc} \delta(\varepsilon) = \delta(\bot) = \bot & \delta(\varphi) = (\varphi\,\mathbf{?}\,\varepsilon\,\mathbf{:}\,\bot) & \delta(R*) = \delta(R) \odot R* \\ \delta(R \mid R') = \delta(R) \bigoplus \delta(R') & \delta(R \,\&\, R') = \delta(R) \mathbin{\&\!\!\&} \delta(R') & \delta(\sim R) = \overset{\frown}{\ }\delta(R) \\ \multicolumn{3}{c}{\delta(R \cdot R') = \mathbf{if}\ \mathit{Nullable}(R)\ \mathbf{then}\ (\delta(R) \odot R') \bigoplus \delta(R')\ \mathbf{else}\ \delta(R) \odot R'} \end{array}$$

If c is a term of type *character* then $\delta_c(R)$ is obtained from $\delta(R)$ by instantiating all internal nodes (conditions of the if-then-elses) φ by tests $c \in \varphi$. This means that in the context of SMT, $\delta_c(R)$ is a term of type *regex*, while $\delta(R)$ represents the lambda-term $\lambda c.\delta_c(R)$ that is constructed *independently* of c.