Online Verification of Commutativity

Aditi Kabra
Carnegie Mellon University
Pittsburgh, PA, USA
akabra@cs.cmu.edu

Dietrich Geisler
Cornell University
Ithaca, NY, USA
dag368@cornell.edu

Adrian Sampson
Cornell University
Ithaca, NY, USA
asampson@cs.cornell.edu

Abstract

Systems of transformations arise in many programming systems, such as in graphs of implicit type conversion functions. It is important to ensure that these diagrams commute: that composing any path of transformations from the same source to the same destination yields the same result. However, a straightforward approach to verifying commutativity must contend with cycles, and even so it runs in exponential time. Previous work has shown how to verify commutativity in the special case of acyclic diagrams in $O(|V|^4|E|^2)$ time, but this is a *batch* algorithm: the entire diagram must be known ahead of time. We present an *online* algorithm that efficiently verifies that a commutative diagram remains commutative when adding a new edge. The new incremental algorithm runs in $O(|V|^2(|E| + |V|))$ time. For the case when checking the equality of paths is expensive, we also present an optimization that runs in $O(|V|^4)$ time but reduces to the minimum possible number of equality checks. We implement the algorithms and compare them to batch baselines, and we demonstrate their practical application in the compiler of a domain-specific language for geometry types. To study the algorithms' scalability to large diagrams, we apply them to discover discrepancies in currency conversion graphs.

CCS Concepts: • **Theory of computation** → **Dynamic graph algorithms**; **Program verification**.

Keywords: commuting diagrams, automatic type conversion, online verification

ACM Reference Format:
Aditi Kabra, Dietrich Geisler, and Adrian Sampson. 2020. Online Verification of Commutativity. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Tools for Automatic Program Analysis (TAPAS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3427764.3428322

1 Introduction

Many systems use diagrams: graphs where nodes are domains and edges are transformation functions. A type system with coercions, for example, corresponds to a graph whose nodes are types and whose edges are coercions. Figure 1 illustrates an example in a simple language with units-of-measure types [4]. In such a system, an important correctness criterion is that the diagram *commutes*: when traversing the graph from any start node to any end node, applying every transformation along the path to any input value, the result is the same output value *independent of the path chosen between the two nodes*. With our coercion example, it is a problem if casting to a supposedly equivalent type as an intermediate step resulted in a different answer than a direct cast. Specifically, given a variable `x` of type `meters`, applying the cast `(wugs) x` can be done in two ways: either `(wugs)(feet) x` or `(wugs)(miles) x`. Which path is taken depends on the compiler; we would like the choice of paths to be semantically equivalent so the compiler is free to make a choice.

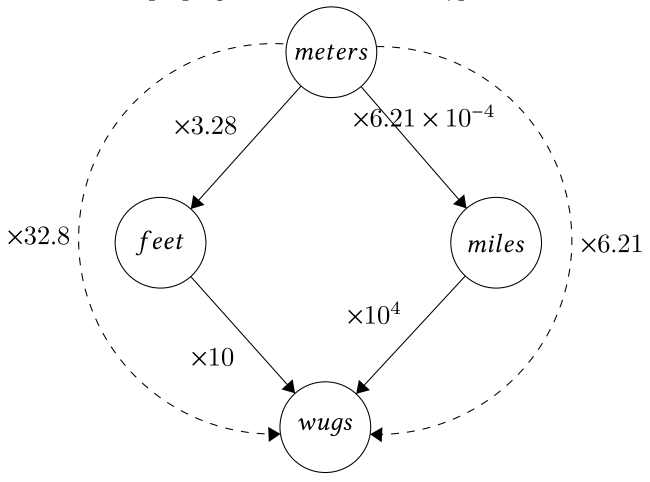
This paper is about efficiently checking commutativity in diagrams that arise in real systems. We assume a simple equivalence checker for individual transformation functions: in our type system example, for instance, it is possible to check transformation equivalence by comparing the conversion factors. Our aim is to analyze the graph of transformations and minimize the number of times we need to perform an equivalence check. Since diagrams may change over time in real systems, as new conversions are added, and verifying the entire system from scratch may be computationally expensive, we want an online method that only checks the impact of new edges. In Figure 1, for example, a run-time system can catch the point where the programmer adds a bad conversion definition by verifying each new conversion edge as it is created.

Efficient commutativity checking is not trivial. The presence of cycles implies a potentially infinite number of paths.. Further, naïvely checking if all path pairs that begin and end at the same node in a given diagram commute could require a number of function equality checks that grow as factorial in the number of nodes, because a path consists of an ordering of nodes. Previous work [5] has identified an $O(|E|^2|V|^4)$ algorithm to verify that a complete acyclic diagram commutes; however, it addresses neither online addition nor cyclic diagrams.

For verifying commutativity over online addition, we identify two key insights. First, when a new edge is added, only

```
var x : meters = 1;
define foot:
    1 meter = 3.28 feet;
define miles:
    1 meter = 0.000621 miles;
define wugs:
    1 mile = 10000 wugs;
    1 foot = 10 wugs;
var y : wugs = (wugs) x;
```

(a) A sample program with user defined type conversion.



(b) Diagram for the type conversions in the program.

Figure 1. In this sample program, the user implicitly defines two ways to cast variable a from meters to the new unit wugs. The definitions are different, and a compiler performing implicit conversion would not know which to choose.

one path per source and sink pair needs to be checked against the existing commutative diagram. Because the diagram commutes, all the paths between a given source and sink are equal and a representative to check against can arbitrarily be chosen. This leads to an $O(|V|^2(|E| + |V|))$ algorithm to verify a diagram remains commutative over the course of online addition, assuming an oracle to check the equality of functions. The algorithm makes an *asymptotically* optimal number of calls to the oracle.

Second, there is a single rule that places a partial, transitive ordering on paths indicating the amount of information they contain about other paths. This insight yields a greedy $O(|V|^4)$ optimization step that results in the number of oracle calls being exactly minimal. The optimization is critical when equality checking is expensive.

We evaluate our algorithms against random graphs and use them in two case studies. First, we use our algorithm in the domain specific geometry type language *Gator* [2] to ensure that user defined transformations between spaces stay consistent. Second, we use our algorithm to identify inefficiencies in a currency conversion graph. We empirically compare our solution to three baseline implementations: a naïve cycle-sensitive all-pairs check, a check for all path pairs that involve the new edge, and an algorithm suggested by previous work to solve the batch version of the problem for acyclic diagrams. Our proposed algorithms run orders of magnitude faster than the baseline implementations.

2 Formal Problem Setup and Terminology

We start by formalizing the notion of a diagram, drawing terminology from the previous acyclic work by Murota [5].

Notation. We start with a directed graph $G = (V, E)$, where V corresponds to sets of elements and edges (u, v) in E correspond to functions that maps elements of u to elements in v. These functions form a semigroup $\mathcal{F}$, where multiplication is function composition. A semigroup consists of a set and an associative binary operation, which we use to capture function composition. The correspondence between edges and functions is stored as a mapping $f : E \longrightarrow F$, where f maps each edge to the function it represents.

A path is a sequence of edges. The edge-to-function mapping f can be naturally extended to paths: if path $p{=}e_1; \cdots ; e_n$ then $f(p) = f(e_1); f(e_2); \cdots ; f(e_n)$. We write $\partial(p)^+$ for p's start node, $\partial(p)^-$ for its end node, and $\partial(p)$ to denote the pair $(\partial(p)^+, \partial(p)^-)$.

A pair of paths p_1 and p_2 is said to *parallel* iff their terminal nodes are the same, i.e., $\partial(p_1) = \partial(p_2)$. ∂, ∂^+ and ∂^- are extended to apply to parallel pairs. For parallel pair $\phi = (p_1, p_2)$, $\partial(\phi) = \partial(p_1) = \partial(p_2) = (\partial(\phi)^+, \partial(\phi)^-)$.

Let $\mathcal{R}_{all}$ be the set of all parallel pairs of paths in a given diagram. The diagram commutes iff $\forall(p_1, p_2) \in \mathcal{R}_{all}$, $f(p_1) = f(p_2)$; that is, the composition of maps along any path connecting any pair u to v is independent of path choice.

Problems. The Online addition problem, given a commuting diagram and a new edge, returns whether the diagram commutes. Checking function equality is a domain specific, potentially hard problem, dependent on the nature of the graph. For example, in our case study in graphics programming (see Section 5.1), edges are matrices and nodes are vector spaces, so function composition uses matrix multiplication and equivalence checking simply compares matrix values. We therefore assume some oracle for checking transformation function equivalence that will vary by domain. We therefore collapse the Online addition problem to the Verification set problem; we solve the latter and assume an oracle with the results to produce the former. The latter, when given a diagram and a new edge, returns the set of parallel pairs of paths, such that if and only if the members in each pair have function equivalence, then the new graph must commute. The output to the Online addition problem can then be obtained as whether function equivalence checking for all pairs succeeds.

The algorithms in this paper assume that the function equivalence oracle is reflexive, symmetric, and transitive.

3 Baseline Algorithms

To examine the efficacy of our proposed solution to the Verification set problem, we compare it to some potential alternatives. Specifically, we examine a naïve factorial algorithm, a slightly less naïve factorial algorithm which we identify to be a two-flip tolerant path search, and Murota's previous batch solution [5].

3.1 Naïve Baseline Algorithm

Our first goal is to develop a baseline (exponential) algorithm that can reason about cycles without producing an infinite set of paths. This algorithm will first pare the structure of the graph down to remove cycles, extract the pairs of paths in the graph, and finally reason about each pair to check commutativity. This results in two components C and Q: the cycle verification pairs and acyclic parallel pairs, respectively.

We start with the set of all parallel pairs in the diagram. We pare it down to be finite by handling cycles: using a procedure like Johnson's algorithm [3], we find all simple cycles in the diagram. We create a cycle verification set, C, and verify for each cycle that a single traversal is equal to the identity function by adding $(v \longrightarrow v, 1)$ for each node v in the cycle to C. Here, $v \longrightarrow v$ is a simple cycle starting and ending at v, 1 is the identity function, and these must be verified to be equal to each other.

We then create a set $\mathcal{P}$ of all the paths in the diagram with no cycles, and filter the set $\mathcal{P} \times \mathcal{P}$, excluding pairs where the paths begin or end on different nodes, or are identical, to get the set of all cycle-free parallel pairs Q. After verifying C, it is sufficient to verify only Q (as opposed to all pairs) because cycles must now be the identity, so for any pair in the set of all parallel paths, any instance of a cycle can be removed to obtain an equivalent pair with shorter, cycle-free paths.

If the shorter pair has equal paths then the paths in the original pair must also be equal to each other. It is therefore safe to remove all pairs of paths with cycles, leaving only parallel pairs where neither path has a cycle. $\mathcal{P}$ is finite, bounded by $2^{|V|}$, as a path without cycles is an ordering on nodes, each node occurring at most once. $|\mathcal{P} \times \mathcal{P}|$, and consequently, $|Q|$, are also finite, bounded by $2^{2|V|}$. Thus the algorithm terminates and returns a finite (if large) set.

3.2 Baseline Incremental Algorithm

For an incremental algorithm, we explore how the addition of an edge can change the baseline to looking at a subset of the graph rather than every pair of paths. In achieving this, this second baseline essentially refines the results of the naïve; a similar structure, but with a substantially reduced set of paths to examine.

Like before, we start by creating a cycle verification set C', but includes only the simple cycles that pass through the new edge. Then, instead of Q, the set of all non-cyclic parallel pairs, the algorithm obtains its subset Q' consisting of all non-cyclic parallel pairs such that exactly one path in each pair passes through the new edge. To this end, the algorithm performs a *two-flip tolerant path search* whose output is passed into a *path extraction algorithm*; this search finds all parallel pairs for which only one path includes the new edge. The result of the path extraction algorithm to get the final output $Q' \cup C'$.

This narrowing can be done because the original diagram commutes. Pairs where both paths do not involve the new edge would remain equal (this would apply to cycles too; cycles that do not pass through the new edge must be the identity). Also, pairs where both paths involved the new edge would have to be equal. To see why this is true, each path could be thought of as consisting of the composition of three segments. For path pair p, and new edge from node S to node T, the first segment extends from $\partial(p)^+$ to S, the second, the new edge (S, T) itself, and the third, from T to $\partial(p)^-$. The new edge could only appear once because cycles have already been dealt with so only pairs where the path includes the new edge once need be checked. The first segment of both pairs would have to be equal because they existed as parallel pairs in the original diagram, and similarly the third segment would also have to be equal. The second segment, consisting of the same edge, would also have to be equal because the equivalence oracle is reflexive. A composition of these three equal components would be then be equal, since the oracle would preserve transitivity of equality. We are left only with parallel pairs where exactly one of the paths passes through the new edge.

To resolve this algorithm fully, we will need to define the specifics of the two-flip tolerant path search and how to narrow down the results of this search into an actual set of paths to verify.

Two flip tolerant path search. We use a "two-flip tolerant" path search from the source (S) to the sink (T) of the new edge to identify the pairs of paths where exactly one path includes the new edge.

In a normal directed graph path search, only forward edges, i.e., edges that go outward from the current node while executing the search are considered. A *two flip path* consists of up to three phases: in the first phase, only backward edges—pointing inward to the source of the search—are accepted. In the second phase, only forward edges are accepted, and in the third phase, again only backward edges are accepted. For a two flip tolerant path p, let $t_1(p)$ map to the first phase, $t_2(p)$, to the second, and $t_3(p)$, to the third. The node between the first two phases we refer to as the *first flipping point*, which has both edges pointing outward; similarly, we refer to the node between the latter two phases as the *second flipping point*, at which both edges point inwards.

We present the idea diagrammatically in Figure 2. Squiggly arrows represent path phases (these are the composition of zero or more edges, not a single edge). The new edge is

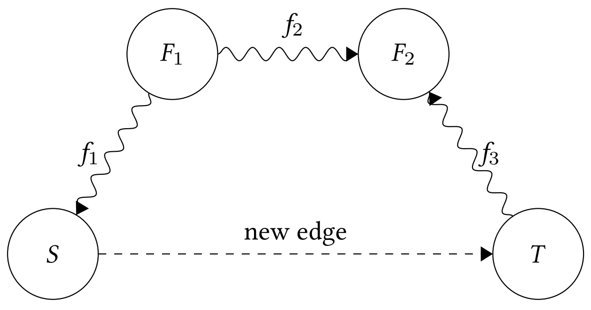


Figure 2. Two flip tolerant path.

represented with a dashed arrow. Here, $f_1; f_2; f_3$ is a two flip path, and $f_1; (S,T); f_3$ is a new path created because of the addition of (S,T) that forms a parallel pair with f_2.

The two flip tolerant path search returns the set of all paths between a given source and sink that have up to two flips (paths with zero or one flip are also accepted).

Path extraction algorithm. Next, the *path extraction algorithm* then transforms the output of the two flip path search into the verification set, $Q' \cup C'$. Given a set of two flip tolerant paths from the new edge source to sink, the algorithm outputs a set of pairs to verify.

Let the new edge added to the diagram be (S,T) and the input set of paths, $\mathcal{P}$. The algorithm processes every two flip tolerant path p in $\mathcal{P}$ case-wise to obtain pairs to add to the output set.

- In the case where p has two flips, $t_1(p); (S,T); t_3(p)$ and $t_2(p)$ form a parallel pair.
- When p has only the first flip (which is to say, the third phase of the path is missing), the parallel paths are $t_1(p); (S,T)$ and $t_2(p)$.
- Similarly when only the second flipping point is present (so that there is no first phase), then the parallel pair is $(S,T); t_3(p)$ and $t_2(p)$.
- Finally when no flipping points are present, there are two possibilities: Either p is a path from S to T, in which case the parallel paths are simply the edge (S,T) and p, or p is a path from T to S. In this case, we have found a cycle, $p; (S,T)$, to be paired with the identity function. Like with the naïve algorithm, for every node v in the cycle, we add the pair $(v \longrightarrow v, 1)$, where $v \longrightarrow v$ is the cycle $p; (S,T)$ written to start and end at v.

Resolving the Incremental Algorithm. We conclude our discussion of this incremental algorithm by proving that the result is the same as if we were running the naïve baseline algorithm. This in turn shows that we have found a more efficient algorithm to achieve the same result of providing a set of paths which can be used to check commutativity.

Theorem 3.1. *Perform the two-flip tolerant path search from the source to sink node of the edge that is to be added followed, and on the output, apply the path extraction algorithm. The result is the set $O = Q' \cup C'$ of new parallel pairs with exactly one path passing through the new edge and neither paths containing any cycles, and the set of simple cycles passing through the new edge.*

Proof. Every element in the output of the path extraction algorithm was by construction an element of O. Every cycle in C' can be expressed as $(S,T); p$, and corresponds to the input two flip tolerant path p.

It remains to show that every new parallel pair p in Q' corresponds to a two flip tolerant path. Let $\partial(p)^+ = F_1$ and $\partial(p)^- = F_2$. Only one path passes through (S,T). Let it be called p_1, and the other path, p_2. The two flip tolerant path from S to T can be constructed as follows: phase 1 is the segment of p_1 from F_1 to S, phase 2 is p_2, and phase 3 is the segment of p_1 from T to F_2. Effectively, F_1 corresponds to the first flipping point, and F_2, to the second. It is possible that some of F_1, F_2, S and T coincide (e.g., p starts at S, i.e., $F_1 = S$), in which case the corresponding segments between the coinciding nodes can be considered the identity; the resultant path simply has fewer than two flips. □

Analysis. An upper bound on the number of pairs that this algorithm returns is $O(|V|^2 2^{|V|})$, since two flip tolerant paths are an ordering on nodes, each node appearing at most once, followed by a selection of the flip points. In practice, the algorithm significantly outperforms the naïve batch baseline because it looks only at parallel pairs that involve the new edge, which is usually a small subset of all parallel pairs. Empirical results are presented in Section 6.

3.3 Optimal Batch Solution

Murota's main result [5] solves the batch version of VERIFICATION SET: given an acyclic diagram, it returns the minimal set of equality checks that succeed if and only if the diagram commutes. Murota describes an algorithm to find the ($|V|^2|E|$ bounded) minimal set of pairs that need be checked.

The approach in this algorithm, at a high level, is to define a function that takes in a subset of pairs and returns the subset of pairs whose equivalence is implied by the equivalence of the pairs in the input set. Then the algorithm greedily eliminates redundancies until a minimal set is reached.

A bilinking is defined to be a parallel pair that is disjoint but for their terminal nodes. The set of all bilinkings is $\mathcal{R}_0$. In an acyclic diagram, if all bilinkings are equal, all parallel pairs must also be equal since any given pair can be expressed as a composition of bilinkings.

Define $r_1 > r_2$ for bilinkings $r_1 = \{p_1, q_1\}$, $r_2 = \{p_2, q_2)$ $\in \mathcal{R}_0$, if there exists a path p such that $\partial(p) = \partial(r_1)$ and p contains p_2. Define $\langle\rangle$ as: $\langle r \rangle = \{s \in \mathcal{R}_0 | r > s\}$.

For bilinking s, let $F(s)$ be the vector in $\mathrm{GF}(2)^{|E|}$ (where GF(2) is the Galois field, that is, finite field of two elements) representing the edges present in s (the n^{th} dimension of $F(s)$ is 1 if the corresponding edge is in s, and 0 otherwise).

```
Result: Find a spanning set Rs = [$r_1$, ... ,$r_k$].
Graph existingGraph
$R_s$ ← {}
foreach node v in V do
    subgraph ←
      existingGraph.extractReachableSection(v)
    /* Get the portion of the graph that can
       be reached starting from v.          */
    tree ← createMinimumSpanningTree(subgraph)
    excludedEdges = edges in subgraph - edges in tree
    foreach edge e ∈ excludedEdges do
        firstPath = tree.findPath(source: e.source, sink:
          e.sink)
        $R_s$.addElement((firstPath, e))
    end
end
return $R_s$
```

Algorithm 1: Finding a spanning set of path pairs, as in section 3.3.

Let this function be extended to sets, so that for some set of bilinkings $\mathcal{S}$, $F(\mathcal{S}) = \{F(s) | s \in \mathcal{S}\}$. A notion of linear independence in this vector field exists.

For a set of bilinkings $\mathcal{R}$, the closure function cl is defined as: $cl(\mathcal{R}) = \{s \in \mathcal{R}_0 | s \text{ is linearly dependent on } F(\mathcal{R})\}$. The closure function on $\mathcal{R}$ captures all the pairs that can be made by made by composing or "gluing together" the bilinkings in $\mathcal{R}$. Using these two functions, we define the function σ on a set of bilinkings $\mathcal{R}$ as $\sigma(\mathcal{R}) = \{s \in \mathcal{R}_0 | s \in cl(\mathcal{R} \cap \langle s \rangle)\}$. This is the function used to capture all the pairs whose equivalence is implied by the equivalence of pairs in $\mathcal{R}$. We use σ to iteratively check if a given pair is redundant. We eliminate Bilinkings until we reach a minimum "spanning" subset.

Roughly, the algorithm proceeds by first efficiently finding a *spanning* set of bilinkings (a subset whose verification implies the verification of all bilinkings in the graph). It does this, starting at every node, by finding the reachable subsection of the graph, and a spanning tree for the subsection. From each edges in the reachable section that is not a part of the tree, it generates a bilinking using the edge and a path in the tree that is parallel to the edge (Algorithm 1).

With the spanning set thus initialized, it greedily tries to remove each pair from the spanning set if the set remains spanning even after removing the edge (Algorithm 2).

The proof of correctness can be found in Murota[5]. The number of checks returned by the algorithm is at worst $O(|V|^2|E|)$. The overall run time of an optimized implementation is $O(|V|^4|E|^2)$.

```
Result: Find a minimal spanning set of path pairs
          (bilinkings) R.
Function σ(input set S, spanning set Rs):
    output ← {}
    for bilinking ∈ Rs do
        smallerPairs ← allShorterPieces(bilinking)
        /* Get fragments that could build up
           to the bilinking. Corresponds to
           applying <> function.             */
        consideredPieces ← smallerPairs ∩ S
        /* Now see if bilinking can be built
           from these pieces.                */
        /* Linear independence is in GF(2) as
           in algorithm description.         */
        if linearlyDependent(consideredPieces,
          bilinking) then
            output.add(bilinking)
        end
    end
    return output
R ← Rs
for i=1 to K do
    if $r_i$ ∈ σ(R-$r_i$) then
        R ← R-$r_i$
    end
end
return R
```

Algorithm 2: Finding a minimal spanning set, as described in section 3.3.

4 Solving the Online Addition Problem

We present a polynomial time solution to the VERIFICATION SET PROBLEM. As in the online baseline algorithm, we do not concern ourselves with parallel pairs where neither or both paths pass through the new edge.

The key observation allowing us to improve on the online baseline is a result of Theorem 4.1 (which we expand on later): for a given source and sink pair, only a single parallel pair needs to be verified. It is straightforward to see that, should our selected set of pairs and cycles passing through the new edge be verified commutative, the entire diagram must commute. Algorithm 3 uses this strategy of identifying a parallel pair with exactly one path through the edge for each (source, sink) pair.

The `try` block is executed at most $O(|V|^2)$ times, which is also the bound on the number of pairs verified. This bound is asymptotically tight, as can be seen in the case where the graph contains $2N$ nodes along S and T. Imagine dividing the nodes into two groups of N nodes each. Every node in group 1 has a forward edge to every node in group 2 and to S. T has a forward edge to every node in group 2. In this diagram, when adding edge (S, T), N^2 paths need to be verified which is polynomial in the total number of nodes, $2N + 2$.

```
Data: existing graph, new edge.
Result: Set of parallel pairs to verify.
Graph existingGraph; Edge newEdge;
parallelPairs ← {}
for src in existingGraph.Nodes do
    for snk in existingGraph do
        try:
            /* Use any standard path finding
               algorithm such as BFS to find a
               path in the existing graph from
               the specified source to sink.
               */
            Path pathWithNewEdge ← FindPath(
              sourceNode: src, sinkNode:
              newEdge.Source) +
            newEdge +
            FindPath( sourceNode: newEdge.Sink,
              sinkNode: snk)
            if src == snk then
                /* Assign the nullary path from
                   src to snk.                  */
                pathInOldGraph ← src
            end
            else
                Path pathInOldGraph ← FindPath(
                  sourceNode: src, sinkNode: snk)
            end
            parallelPairs.add((pathInOldGraph,
              pathWithNewEdge))
        catch PathFindingFailedException:
            /* No comparable pairs from node
               src to node snk that need to be
               checked                          */
            continue
        end
    end
end
return parallelPairs
```

Algorithm 3. Online polynomial-time algorithm to find parallel pair set.

If trying to optimize for path length (say, if composing functions is expensive) then "find any path" can be replaced with "find the shortest path."

An efficient implementation of the algorithm can run in $O(|V|^2(|V|+|E|))$ time, with space complexity not exceeding the asymptotic $O(|V|^2)$ bound on the output. In such an implementation, path finding from a given source node to all potential sink nodes could be done in a single $O(|V|+|E|)$ breadth first search.

4.1 Optimization Step

In the case where equality checks are very expensive, we begin by finding the minimal set of (source, sink) pairs such that checking for these pairs logically implies having checked the full diagram.

If Algorithm 3 were applied to the diagram shown in Figure 3 there would be redundancies in the output. It turns out that verifying $g_2 = f_2; n; h_2$ is sufficient to ensure the diagram still commutes on the addition of n.

$$\begin{tikzcd}
 & P_1 \arrow[r, "g_1"] \arrow[dl, "f_1"'] \arrow[dd, "l"] & P_2 & \\
S \arrow[rrr, dashed, "n"] & & & T \arrow[ul, "h_1"'] \arrow[dl, "h_2"] \\
 & Q_1 \arrow[ul, "f_2"] \arrow[r, "g_2"'] & Q_2 \arrow[uu, "m"] &
\end{tikzcd}$$

Figure 3. Reduction rule. Each arrow represents a path, where n is the new edge being added. While Algorithm 3 returns two pairs for verification, one from P_1 to P_2 and the other from Q_1 to Q_2, it actually suffices to just check a pair from Q_1 to Q_2 as demonstrated in theorem 4.1.

Theorem 4.1. *If parallel paths $g_2 = f_2; n; h_2$ then it must be that $g_1 = f_1; n; h_1$.*

Proof. We use the fact that f_1=$l; f_2$ and h_1=$h_2; m$.

$$g_2 = f_2; n; h_2 \Rightarrow l; g_2 = l; f_2; n; h_2$$
$$\Rightarrow l; g_2; m = l; f_2; n; h_2; m \Rightarrow g_1 = f_1; n; h_1$$

The proof holds if any of the paths used are the identity, e.g., if f_1 is the identity so S and P_1 are the same node. □

We conclude that verifying a comparable pair of paths with end points (P_1, P_2) implies the verification of all path pairs (Q_1, Q_2) such that Q_1 is a successor of P_1 and P_2 is a successor of Q_2. A successor S to node N is any node such that there exists a path from N to S. Nodes are also their own successors and predecessors. The rule effectively places an ordering on the informativeness of path pairs based on their terminal nodes.

Given that a set of path pairs are equal, suppose we attempt to derive the proposition that a different parallel pair of paths is equal with a step-by-step application of inference rules. Under the assumption that edges are generic functions, and no other information is available, $\mathcal{F}$ is a semi-group. The only inference rules allowed are composition (given that $f_1 = f_2$, it must be that $g; f_1 = g; f_2$) and replacement of one path by a different, equal path (given $f_1 = f_2$ and $g; f_1 = h; f_1$, it must be true that $g; f_1 = h; f_2$). Any permutation of the repeated application of these two rules results in the "reduction rule" already described; it is therefore the only rule that can be used to reduce the set of path pairs to check.

That is to say, if verifying a comparable pair of paths with end points (P_1, P_2) implies the verification of a pair with endpoints (Q_1, Q_2), then it must be that Q_1 is a successor of P_1 and P_2 is a successor of Q_2.

```
Data: Existing graph, new edge.
Result: Set of parallel pairs to verify.
Graph existingGraph
Edge (S, T)
predecessors ← predecessors of S in existingGraph
successors ← successors of T in existingGraph
Graph terminalPairGraph ← empty
for $q \in$ successors do
    for $p \in$ predecessors do
        terminalPairGraph.addNode((q, p))
        for predecessor $\in$ predecessors of q in existingGraph do
            for successor $\in$ predecessors of p in existingGraph do
                terminalPairGraph.addEdge((predecessor, successor))
            end
        end
    end
end
verificationSet ← {}
while terminalPairGraph.nodes not empty do
    currentNode ← terminalPairGraph.node
        // an arbitrarily chosen node of terminalPairGraph
    while currentNode has predecessors do
        currentNode ← predecessorOfCurrentNode
        // an arbitrarily chosen predecessor of current Node
    end
    verificationSet.add(currentNode)
    terminalPairGraph.removeAllSuccessors(currentNode)
end
return verificationSet
```

Algorithm 4: Minimal set finding algorithm.

Using this information it is possible to choose a minimal subset of path pairs to verify, as in Algorithm 4. To summarize this algorithm conceptually, we start by constructing a graph with a node for each possible (source, sink) pair in the graph: each node then represents a possible choice for parallel pair endpoint pairs. Edges are drawn from node (P_1, P_2) to (Q_1, Q_2) if Q_1 is a successor of P_1 and P_2 is a successor of Q_2. We greedily search for the smallest set of nodes from which the entire graph would be reachable. The idea is to look for "roots" in the graph that have to be included in the ultimate verification set because they have no predecessor in the graph and cannot be verified "through" the verification of some other pair. Then all the successors whose verification is implied by the roots are eliminated.

At the end of the greedy graph reduction we are left with the unique set of root nodes. The only way to reduce the set of parallel pairs is to apply the reduction rule of theorem 4.1, but all the ways in which the rule is applicable was already captured in the edges of the graph. The leftover set has no edges and no scope for further reduction.

Also, the verification of the parallel pairs returned in the algorithm implies that the output of the previous algorithm must commute and that the entire diagram must commute.

The run time of the first step is $O(|V|^4)$, and that of the second step is $O(|V|)$, so that the overall bound is $O(|V|^4)$. Space complexity remains $O(|V|^2)$.

5 Case Studies

To demonstrate our algorithms applied to a real world situation, we search for inconsistencies in diagrams of geometry transformations, and in a diagram of the exchange rate between currencies. Each of these applications use commutative diagrams, and the commutative nature of each is necessary to reason about some form of correctness. We explore these examples with the intent of showing that the algorithms discussed apply to realistic settings and potentially identify real-world examples of incorrect behavior.

5.1 Gator

Gator is a domain specific language designed around geometry types, which are used to describe properties and transformations of geometric objects [2]. A key feature of Gator is `in` expressions, which insert code to automatically transform between two geometry types. For example, given a point `p` represented in 2-dimensional Cartesian coordinates (which has type `cart2`), we can transform this point into polar coordinates using the expression `p in polar`. These `in` expressions create a structure of commutative diagrams, allowing use as introduced in Section 1.

Specifically, Gator introduces transformations between *reference frames*, which are the geometry equivalent of transforming between linear algebra basis vectors. Each edge on our transformation graph is thus a matrix, with composition of edges as matrix multiplication and an oracle checking matrix equality (up to a rounding error ϵ).

There are several examples of reasonably complicated transformation graphs that we can pick from. Gator includes graphics examples as part of its examples package, all of which are in the Gator paper; for this evaluation, we looked at the *phong*, *reflection*, and *shadow map* examples.

We implemented a system for interfacing between the optimal set path checker (Algorithm 4) in the open-source implementation of Gator. The system was tested with intentional bugs, of which it found them all, although no "real" bugs were found. The graphs used were of size 5 or less; for graphs of this size, the checker was able to run in real time with no noticeable loss of frames. Since the program is running at 60 frames per second, the checker was running at a rate faster than .01 seconds.

5.2 Currency Graph

We imagine a units-of-measure type system as being an interesting application of concurrency graphs; however, to make this more interesting and scale nicely to large graphs with existing data, we focus on the specific unit of currencies. Consider a diagram with nodes as currencies and a directed edge being the conversion rate from its source node's currency to its sink node's currency. Since the exchange rate of money from any given base currency to a target currency can be expected to be the same regardless of which intermediate currency transformations are used, this diagram should commute.

Using a web API[1] for currency data, we built the fully connected diagram of exchange rates between 32 currencies on a given day. To ensure that it indeed commuted, we started with an empty diagram, and added in edges one by one. Before the addition of each edge, we used the algorithms (Algorithm 3 and Algorithm 4, the online polynomial and online minimal set algorithms, respectively) to ensure the addition of a new edge did not introduce inconsistencies in the existing diagram. If a new edge was problematic, the algorithms returned an example inconsistent pair that would arise from the addition of the edge. The pair would consist of two currency transformation sequences with the same source currency and ultimate destination currency, but with different effective exchange rates values, as computed by taking the product of all the exchange rates encountered through the chain.

We allowed an "error tolerance" so that differences reported would not be the trivial consequences of a floating point error. However, this relaxation of the equality oracle into imprecision meant that the mathematical reasoning that allow the algorithms to remove redundant path checks no longer applied. For instance, composing a new function with two approximately equal functions does not lead to equal results, so Theorem 4.1 fails with this approximate equality. When the algorithms reported no inconsistencies, it was still possible that the graph possessed inconsistencies above the given threshold and did not commute. Nonetheless, both algorithms were effective in catching inconsistencies. Algorithm 3 started finding inconsistencies at error tolerances to the order of 10^{-3}, and Algorithm 4, which makes more invalid redundant path check removals, at error tolerances to the order of 10^{-7}.

Averaging over evaluation for the first 30 days of 2020, building and verifying a diagram to completion (inclusive of the time required by network calls) took 243±19 seconds using Algorithm 4, and in 133±13 seconds with Algorithm 3.

[1]https://exchangeratesapi.io

Table 1. Computation time for 9-node graph of density 0.4, averaged over ten runs.

Algorithm	Average seconds of computation
Naïve baseline	0.77
Two Flip tolerant	0.075
Batch algorithm	7.55
Algorithm 3	0.0038
Algorithm 4	0.00086

Table 2. Output size for 9 node graph of density 0.4, averaged over ten runs.

Algorithm	Average number of output pairs
Naïve baseline	39754.9
Two Flip tolerant	748.9
Batch algorithm	23
Algorithm 3	78.3
Algorithm 4	1

For this large of a graph and data set, these times are reasonable and show these algorithms can be used in a realistic setting. Finding actual inconsistencies further shows the value of using these algorithms and commutative diagrams in the real world.

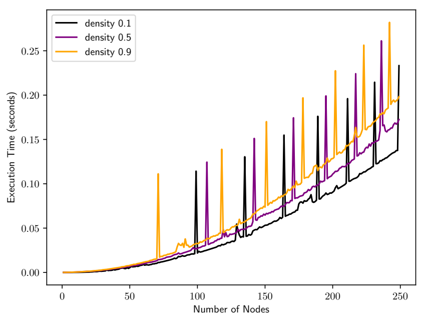
6 Evaluation

We compare performance of the following path checking algorithms: (1) the naïve baseline, (2) the less naïve two-flip baseline, (3) the batch baseline, (4) Algorithm 3, the non-minimal polynomial-time algorithm, and (5) Algorithm 4, the minimal set finding algorithm. The two metrics we evaluate are time for response and size of response set (smaller sets—tighter output results—would mean less calls to the oracle). We use randomly generated graphs of varying size: given a graph and a new edge, we time how long it takes for an algorithm to return the set of pairs that need to be verified. All computations were performed on a MacBook Pro 2015, 2.9 GHz dual-core Intel Core i5.

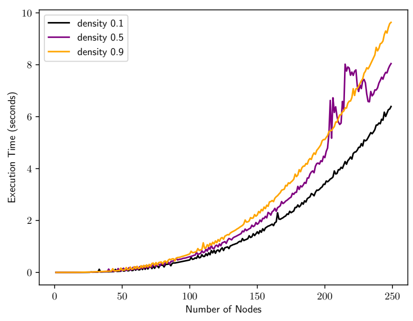
6.1 Comparison of Algorithm Time Cost

The average time taken by each algorithm over the course of 10 runs over randomly generated graphs with 9 nodes and 32 edges is listed in Table 1.

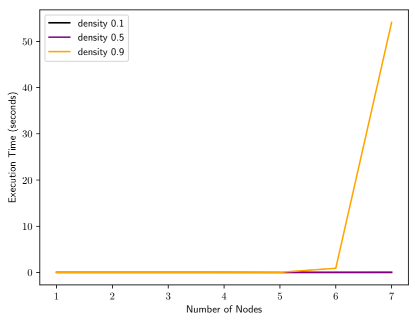
The naïve baseline performs poorly, taking well over a thousand seconds for even small graphs of 10 nodes. While the batch algorithm improves on this, it still does not scale very well, with computation for a graph with 14 nodes and 0.4 density taking hours. Our implementation does not memoize the construction of the vector and matrix representation of paths in GF(2); profiling indicates that this construction



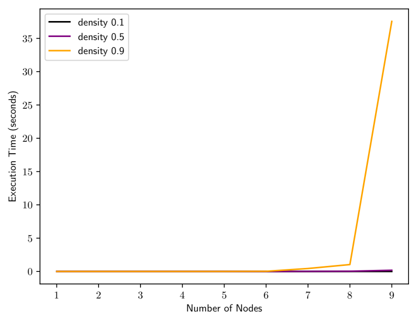
(a) Algorithm 4.



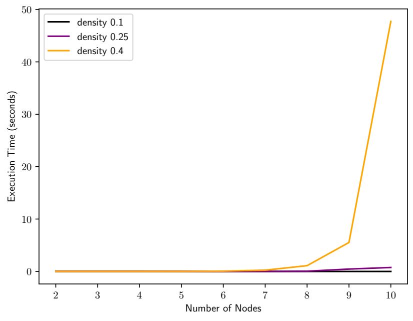
(b) Algorithm 3.



(c) Naive baseline.



(d) Two flip tolerant baseline.



(e) Batch algorithm baseline.

Figure 4. Running time for various algorithms as input graph size and density scale.

is a major factor in the high time cost for this algorithm. Algorithm 3 performs only slightly better than the batch algorithm. Surprisingly, the optimal set algorithm cuts time cost by several orders of magnitude, and runs in milliseconds for small graphs. All implementations are sensitive to density, performing better when density is low.

6.2 Scaling of Time with Input Size

Figure 4 shows that the algorithms' time scales with size, as expected. Both Algorithm 4 and Algorithm 3 exhibit graphs that are polynomial in appearance. The naïve baseline as well as the two flip tolerant baseline display quick growth. The batch algorithm also grows fast, though not as much as the online checking baselines.

We define density to be the ratio of the number of edges in the graph to the total possible number of edges (which is $|V|^2$, where $|V|$ is the number of nodes). Run time relates to the density of edges in the input graph. The degree of the effect differs with the algorithms, as Figure 4 shows. Generally, denser graphs entail longer computation time. For the batch algorithm we use lower densities since the input graph must be acyclic. This puts an upper bound on density that approaches 0.5 in large graphs.

6.3 Variance

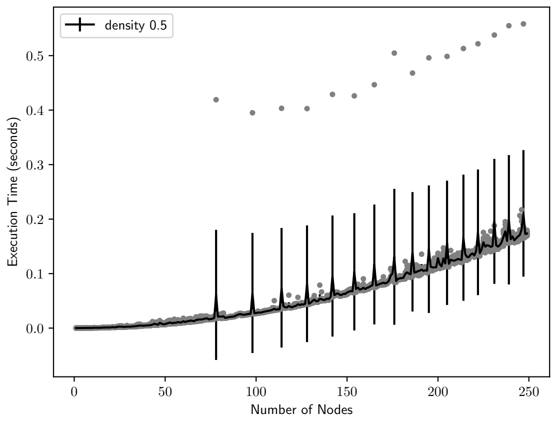
The periodic spikes in Figure 4a are striking. We plot the spread of results in Figure 5a to understand what is happening. Grey points are the results of evaluation on individual points, and error bars show standard deviation. The black curve traces the mean. We find Algorithm 4 has outliers about two standard deviation above the mean responsible for the spikes in the average. The outliers themselves follow a polynomial curve, appearing almost periodically. We have not yet identified the cause of the behavior. Figure 5 depicts the situation for Algorithm 3, where no such effect is observed.

6.4 Size of Output

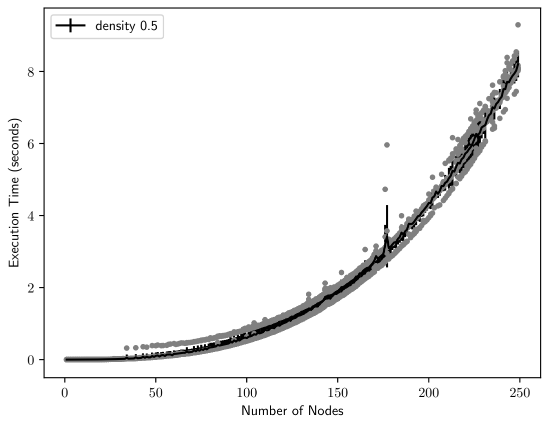
Output size is a metric of interest, should the equality checking oracle be expensive. Table 2, summarizes the number of output pairs that the algorithms returned on average over 10 runs, for graphs with 9 nodes and 32 edges. These results are essentially as expected, although it is interesting to note that Algorithm 3 produces around triple the number of pairs compared to the batch algorithm. Also note that Algorithm 4 produces the minimal number of paths, showing why it is the minimal set algorithm.

7 Related Work

Section 3.3 describes Murota's solution to efficiently finding the minimal set of path pairs that need to be compared to



(a) Algorithm 4.



(b) Algorithm 3.

Figure 5. Spreads of algorithm running times.

check if a given acyclic graph commutes [5]. We did not find any other work that solves the question of verifying that diagrams commute. However, the question of commuting does come up in programming languages with implicit type conversion. Gator [2], as described in Section 5.1, supports automatic type conversion between geometry types. The language implements some restrictions to eliminate obvious cases of non-commuting graphs, but does not verify that defined graphs commute, allowing scope for non-commuting graph definitions. Frink [1] is a language that supports automatic conversion between units and infinite precision floating point numbers. It does not appear to support the implicit definition of conversion between units but if extended to do so, would need to contend with the problem of commuting graphs. The same is true for F# which has support for units of measure [4] and Ada's GNAT compiler [6].

8 Conclusion

Being able to verify if diagrams commute allows a compiler to make deterministic automatic type conversions and can catch inconsistencies of definition in a program with user defined conversions. In this paper, we have presented verification algorithms that efficiently compute the set of paths that would equal to each other if and only if the diagram would still commute after addition of a new edge.

Integrating conversion consistency checks into widely used languages such as Scala could provide a lot of value to the program. Since Scala and several other languages provide automatic conversions between types, it seems important to ensure that the choice of which path to take (an apparently arbitrary choice) does not effect the behavior of the program. Having an algorithm to ensure the commutativity of the resulting diagrams can ensure that behavior is correct and help prevent semantic confusion or errors when using such features. More engineering work still remains to implement this feature in a language such as Scala, but this paper provides the algorithms necessary to explore a solution.

References

[1] Frink. http://frinklang.org/#Features. Accessed: 2020-08-10.

[2] Dietrich Geisler, Irene Yoon, Aditi Kabra, Horace He, Yinnon Sanders, and Adrian Sampson. Geometry types for graphics programming. In *OOPSLA*, 2020.

[3] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4:77–84, 1975.

[4] Andrew Kennedy. Types for units-of-measure: Theory and practice. In *Proceedings of the Third Summer School Conference on Central European Functional Programming School*, CEFP'09, page 268–305, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Kazuo Murota. Homotopy base of an acyclic graph—a combinatorial analysis of commutative diagrams by means of preordered matroid. *Discrete Appl. Math.*, 17(1–2):135–155, May 1987.

[6] Edmond Schonberg and Vincent Pucci. Implementation of a simple dimensionality checking system in ada 2012. In *Proceedings of the 2012 ACM Conference on High Integrity Language Technology*, HILT '12, page 35–42, New York, NY, USA, 2012. Association for Computing Machinery.