

A Practical Way To Maintain A Transitive Reduction

Michael L. Case Robert K. Brayton Alan Mishchenko
Department of EECS, University of California, Berkeley
{casem, brayton, alanmi}@eecs.berkeley.edu

Abstract—Several authors have studied methods to construct the transitive reduction of a directed graph, but little work has been done on how to maintain it. We are motivated by a real-world application which uses a transitively reduced graph at its core and must maintain the transitive reduction over a sequence of graph operations. This paper presents an efficient method to maintain the transitive reduction of a possibly cyclic directed graph under the following operations:

Vertex Addition: This involves adding a new vertex along with an application specified set of new edges.

Edge Removal: This involves removing an arbitrary edge in a transitive reduction and possibly insertion of additional edges to preserve implicit transitivity relationships

We present overviews of how to efficiently perform these operations without storing the original graph or its transitive closure, and we give proofs of correctness and the time complexities.

I. INTRODUCTION

Often one encounters a graph where the constituents satisfy a transitivity relationship. That is, if $a \rightarrow b$ and $b \rightarrow c$ then necessarily $a \rightarrow c$. If we have such a special graph then rather than storing the graph we could choose to store one of two related graphs: the transitive reduction and the transitive closure.

Definition 1 (Transitive Reduction): Given a directed graph G , the transitive reduction $r(G)$ is a directed graph satisfying the following properties:

- 1) The vertex set of G is equal to the vertex set of $r(G)$.
- 2) There is a directed path from vertex u to vertex v in G if and only if there is a directed path from u to v in $r(G)$.
- 3) There is no graph with fewer edges than $r(G)$ satisfying the above conditions.

Definition 2 (Transitive Closure): Given a directed graph G , the transitive closure $c(G)$ is a directed graph satisfying the following properties:

- 1) The vertex set of G is equal to the vertex set of $c(G)$.
- 2) If there is a directed path from vertex u to vertex v in G then there is an edge from u to v in $c(G)$.

We can interpret these definitions in the following way: the transitive reduction is the smallest graph preserving all original vertex reachability relationships. It has the property that it is the way to store the graph that requires the least amount of memory. The transitive closure in a sense is a representation that takes considerably more storage, but it has the property that checking if there is a directed path from u to v in the original graph is equivalent to testing for the existence of a single edge in the closure.

We are motivated by an application where it is critical to minimize storage space, and so we are interested in investigating the transitive reduction. Several authors have studied how to build the transitive reduction, and this is most often done by first constructing the transitive closure. In our motivating application it is not feasible to construct the closure due to the constrained storage space. Additionally, our graph evolves over time, and we wish to avoid re-computation of the transitive reduction after every graph transformation. Ways to maintain the transitive reduction across graph transformations have been studied, but prior to now the maintaining the reduction meant that one also had to maintain the transitive closure.

We present algorithms to perform operations on the transitive reduction of a possibly cyclic directed graph and preserve the reduction property. Specifically we are interested in maintaining the transitive reduction under the following:

Vertex Addition: This involves adding a new vertex along with an application specified set of new edges.

Edge Removal: This involves removing an arbitrary edge in a transitive reduction and possibly insertion of additional edges to preserve implicit transitivity relationships

These algorithms do not require the explicit construction of the transitive closure and so take much less memory to implement than previous solutions. We have implemented this theory and applied it towards our motivating application, and we witnessed dramatic performance improvements.

II. RELATED WORK

Aho et. al. [2] originally defined the transitive reduction. They show that the transitive reduction of an acyclic graph is unique, and they show constructively how to find it.

Several authors [3], [4] have showed how to construct the transitive reduction more efficiently. Some authors [5], [6] prefer to address a related problem called the minimum equivalent graph problem. In this problem, one attempts to find something akin to a transitive reduction where the edge set is constrained to be a subset of the original edge set. These papers provide motivation for our solution.

A fundamental part of finding a transitive reduction is to identify strongly connected components. Traditionally people use variants of Tarjan's algorithm [7] to find the components of a graph. Several [8], [9], [10] have addressed ways to speed up the original algorithm. Some [11] have studied a method to efficiently maintain the closure under edge deletion. This is related to our work, but we study maintaining not only a component but an entire transitive reduction under edge deletion.

La Poutre et. al. [12] studied a way to maintain the transitive reduction of a graph under edge insertion and deletion. Unfortunately their method requires one to store and simultaneously update the transitive closure as well. This makes their method impractical.

III. AN EXAMPLE TRANSITIVE REDUCTION

At this point it is useful to introduce an example of a transitive reduction of a cyclic directed graph. This example will show the general shape of a transitive reduction as well as define key terminology.

If one wanted to build a transitive reduction for the top graph in Figure 1, one would go through the following steps:

- 1) Identify the strongly connected components as $\{A, B, C\}$ and $\{D, E, F\}$. Remove all edges inside of these components and then join the components by two simple cycles.

From this point forward we treat each component as a single vertex. We designate one vertex in each component as the *representative*, and we ignore

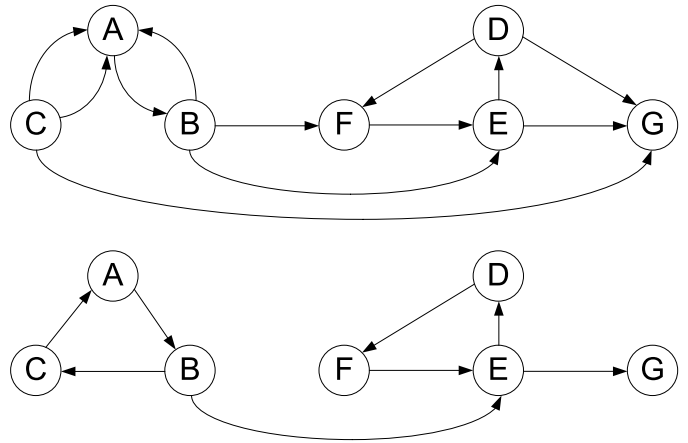


Fig. 1. An example transitive reduction.

the other component vertices. In this case, let B and E be the representatives. Ignoring the non-representative component vertices, the remaining graph is referred to as the *condensed graph* and is acyclic.

- 2) For each edge $u \rightarrow v$ in the condensed graph, test to see if there is an alternate path from u to v that does not involve this edge. If such a path exists then the path is called a *reducing path* and $u \rightarrow v$ is called a *redundant edge*. Remove it.

After the above two steps, the bottom graph in Figure 1 is obtained. It is a transitive reduction.

There are two important observations to make about the transitive reduction:

- After finding the strongly connected component, one could choose an arbitrary vertex ordering in constructing the joining simple cycle. However, one can say that the number of edges in all such cycles is unique. Once the condensed graph is formed, we know it is acyclic and Aho has shown that any acyclic graph has a unique transitive reduction. Therefore we can conclude that the transitive reduction of a cyclic directed graph is unique in cardinality.
- A transitive reduction is composed of simple cycles joined onto a sparse acyclic graph. This shape is used in the algorithms to follow.

IV. MOTIVATING APPLICATION

Our motivating application comes from the field of logic synthesis. In this field, we have a design for a digital computer chip that we wish to optimize. This design comes to us as a graph of logical components, and by manipulating this graph we can optimize the

chip to use fewer logical components, use faster logical components, or use components in a way so as to use less power.

Most often the chip contains some amount of memory. This memory stores the state of the system implemented by the chip design, and the system can be thought of as progressing through a finite number of states stored in this memory. Such a system is called a *finite state machine*.

As the system progresses through its states, it may not be able to reach every state representable by the memory. If we know which states are unreachable then we are free to change the behavior of the system on these unreachable states. This modification may simplify the design where it was unnecessarily complicated. Unfortunately, this simplification requires us to discover the set of states that the system can reach, and currently no scalable algorithms exist to find this set of states for large designs.

In a related work [1], we find an over-approximation to the set of reachable states by proving logical implications of the form $a \Rightarrow b$ over pairs of logical functions a and b present in the design's logic network. We prove by induction that the implication holds in every reachable state. If we succeed in proving a set of implications I in a design, then we build a Boolean function R over the state of the system s such that

$$R(s) = \bigwedge_{(a \Rightarrow b) \in I} a \Rightarrow b$$

Here a and b being two logical functions in the network are functions over the system's current state. Any state s for which $R(s) = 1$ is necessarily reachable, and so R provides an over-approximation to the reachable state set. In practice, this approximation is close enough to the actual reachable state set to enable optimization, and it is fast enough to be applied to large designs.

The process described above involves proving logical implications. A set of implications can be thought of as forming a graph. The vertex set of this graph is the set of logic functions (Boolean functions) available from the original design. For each logical implication between logic functions in the design, we insert an edge between corresponding graph vertices. Call the resulting graph an *implication graph*.

Logical implications obey transitivity, so it makes sense to talk about the transitive closure and transitive reduction of the implication graph. If n is the number of logic functions in the original design then there could potentially be $O(n^2)$ edges in the implication graph.

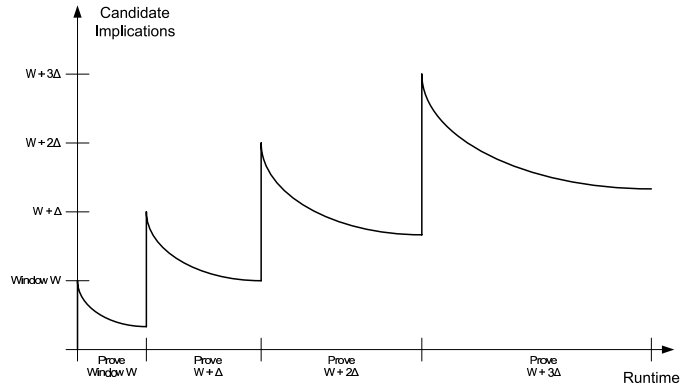


Fig. 2. The hierarchical proof technique.

Also, each implication must be proved by induction on the state space, and we wish to reduce the number of required proofs as much as possible. All information conveyed by the implication graph is also conveyed by its transitive reduction, and so we are motivated to always store and process the transitive reduction of the implication graph.

The inductive proof technique operates by inputting a candidate set of implications and then iteratively pruning it until all the implications satisfy the inductive hypothesis. When the set of implications reaches a fixed point we know that we have succeeded in proving that the set of implications holds in every reachable state. Until this fixed point is reached, we know nothing of the correctness of the implications. To quickly provide the user with meaningful results, a hierarchical proof technique is used. This technique is illustrated in Figure 2. We initially find a set of candidate implications W , and we remove implications from this set until we reach a fixed point. We then add Δ to this set and begin the proof anew. In this way, we reach a usable fixed point early, and our set of usable implications increases over time as we reach new fixed points.

The hierarchical proof approach results in two distinct operations being performed on the implication graph:

Vertex Addition: As the candidate implication set grows, we add new logic functions as new vertices to our implication graph. We then add any new implications associated with these logic functions. So by vertex addition here we mean addition of a new vertex plus a set of edges either going to or coming from the new vertex.

Edge Removal: When we discover that an implication does not satisfy our inductive hypothesis, we remove it from the implication graph.

V. MAINTAINING THE TRANSITIVE REDUCTION

A. Assumptions

Before describing our algorithms for vertex addition and edge deletion on a transitive reduction, we must first make clear our assumptions. This work was done as a way to solve a problem in logic synthesis, and as such there was information available that somewhat simplifies the general problem.

1) *The Vertex Set Represents Logical Functions:* The first assumption is that the vertex set of the implication graph represents logical functions. As such, every vertex has a dual vertex that represents the negation of its logic function. For example, suppose a vertex represents the logic function A . There is another vertex in the graph representing $\neg A$.

Every logic implication $A \Rightarrow B$ has the corresponding contrapositive $\neg B \Rightarrow \neg A$. This means that each edge in the graph also has a dual edge. We exploit this duality in places to save effort. Essentially, at times we only compute a graph transformation on half of the graph and obtain the transformation on the second half by symmetry.

2) *There Exists a Way To Query the Transitive Closure:* The second assumption in this work is that there exists a way to query the transitive closure. Essentially, our logic synthesis framework provides us with a function $\text{fastTest}(a \Rightarrow b)$ which quickly tests whether $a \Rightarrow b$. It has one interesting property that guarantees the correctness of the following algorithms: it is consistent with transitivity. By this we mean that if $\text{fastTest}(a \Rightarrow b) = 1$ and $\text{fastTest}(b \Rightarrow c) = 1$ then $\text{fastTest}(a \Rightarrow c) = 1$.

$\text{fastTest}()$ works by simulating the design for a short time period, but because it is always consistent with transitivity one could interpret $\text{fastTest}()$ as a way to query the transitive closure. However, do not confuse this with actually storing and continually updating the transitive closure as other works have done. At no time do we explicitly work with the closure, but we do leverage this querying mechanism in our algorithms.

B. Vertex Addition Algorithm

Suppose we wish to add a new vertex v into an existing graph, and we also wish to add edges $s \rightarrow v$ and $v \rightarrow d$ for all s and d such that $\text{fastTest}(s \Rightarrow v) = 1$ and $\text{fastTest}(v \Rightarrow d) = 1$. This section describes such an algorithm. Please refer to Figure 3 for an example of this technique.

Also note that in this context, addition of vertex v also implies addition of vertex $\neg v$. A vertex and its dual are considered a pair in this work.

- 1) For both v and $\neg v$ do the following:
 - a) For each existing vertex u in the condensed graph such that u is not colored, if $\text{fastTest}(v \Rightarrow u)$ then add the edge $v \rightarrow u$. Color all descendants of u , not including u .
 - b) For each u such that we just added $v \rightarrow u$, if u is colored then it means that u exists on an alternate path from v , and this alternate path is a reducing path. Delete $v \rightarrow u$.
- 2) For all edges from v and $\neg v$, add the contrapositive of the corresponding implication to the graph.
- 3) For both v and $\neg v$ do the following:
 - a) Color the ancestors of v blue. Color the descendants red.
 - b) If there exists a condensed graph node u that is both blue and red then $v \Rightarrow u$ and $u \Rightarrow v$. This implies that v is part of u 's strongly connected component. Delete all edges to v and from v , and insert v into the strongly connected component (a simple cycle) starting at u .
 - c) If there exists a blue node b and a red node r in the condensed graph such that there is an edge $b \rightarrow r$ then this edge is redundant because it has a reducing path through v . Delete $b \rightarrow r$.

1) *Proof of Correctness:* This will be included in the final draft of this paper.

2) *Time Complexity:* This will be included in the final draft of this paper.

C. Removal of An Edge In a Component

The procedure to remove an edge in a transitive reduction differs depending on whether that edge is in a strongly connected component's simple cycle or if the edge is between two vertices in the condensed graph. We first present the case where the edge is inside of a component. Throughout this discussion the reader may refer to Figure 4 for an illustration of the procedure.

Suppose an edge is being removed that is inside a component with representative v in the condensed graph. We then do the following:

- 1) Set
$$\text{toComponent} = \{u \mid \exists u \rightarrow v \text{ and } u \text{ is a condensed vertex}\}$$

$$\text{fromComponent} = \{u \mid \exists v \rightarrow u \text{ and } u \text{ is a condensed vertex}\}$$
- 2) Remove all edges associated with any vertex in the component.

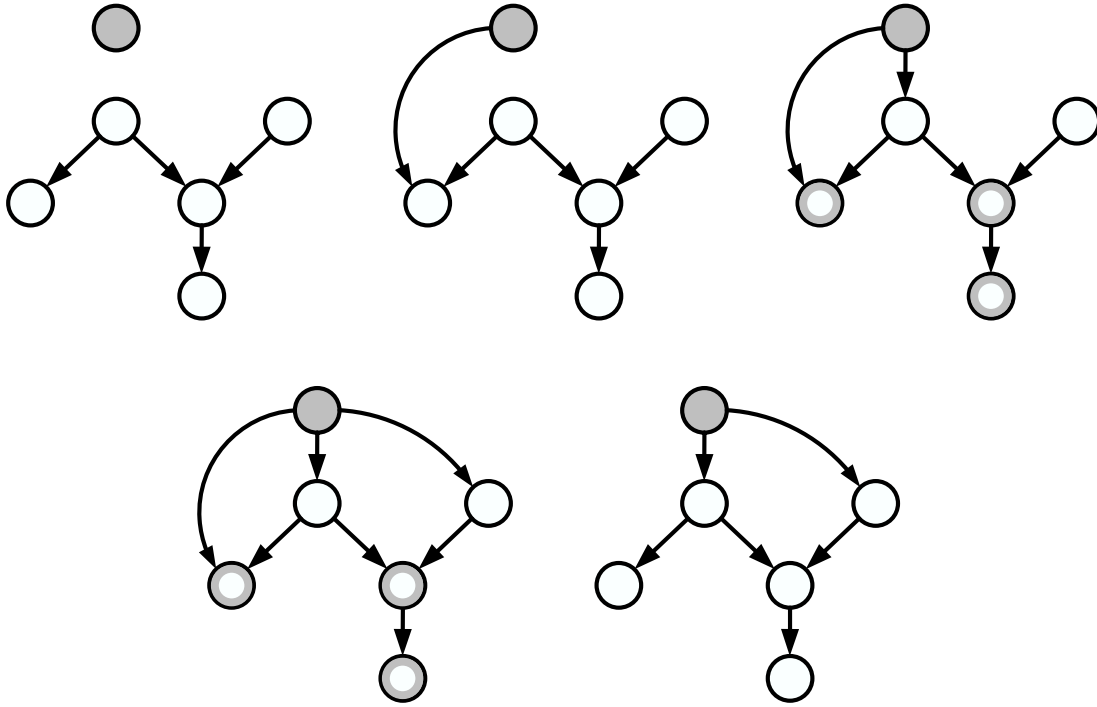


Fig. 3. Vertex addition example. We introduce the gray vertex to the graph, and iteratively add edges to non-colored vertices. At each step, we color the descendants of the recipient vertex. We find that the edge to the left-most vertex is redundant because that vertex becomes colored, and so we remove it and are left with the graph in the lower right.

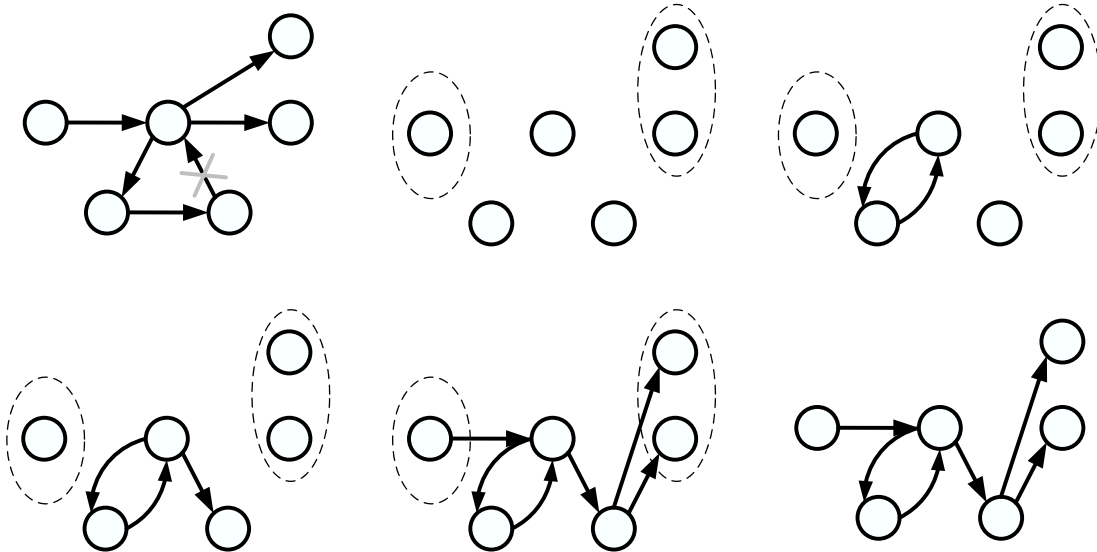


Fig. 4. We remove an edge in a component. First we identify the sets toComponent and fromComponent. We delete all edges associated with the component, and re-discover the new refined components contained within. We then add edges between the new components and to/from the outside world.

- 3) Use Tarjan’s algorithm in conjunction with the `fastTest()` function to re-construct the strongly connected components in the recently disconnected subgraph. Here we assume that `fastTest()` has been updated with new information about the design’s implications, and so this procedure will find more than one component in this new subgraph. We should find that the old component has been partitioned into 2 or more new components.
- 4) Choose vertices in the new components to represent these components in the condensed graph. For each pair of representative vertices (a, b) :
 - a) Use depth first search (DFS) to check for a directed path from a to b and from b to a . Note: because they are not in the same component we cannot find both directed paths.
 - b) If there is no path from a to b then add edge $a \rightarrow b$. Similarly, if there is no path from b to a then add an edge $b \rightarrow a$.

There is now an acyclic condensed sub-graph where the old component once was.

- 5) For each node a with no children in the condensed sub-graph and for each $b \in \text{fromComponent}$, add an edge $a \rightarrow b$. Note that we did not call `fastTest()` here. If `fastTest()` would have rejected this edge then the application will remove this new edge at some later time. To simplify the algorithm, we may be conservatively adding too many edges.
- 6) For each node a with no parents in the condensed sub-graph and for each $b \in \text{toComponent}$, add an edge $b \rightarrow a$.
 - 1) *Proof of Correctness*: This will be included in the final draft of this paper.
 - 2) *Time Complexity*: This will be included in the final draft of this paper.

D. Removal of An Edge In a the Condensed Graph

The final algorithm applies when an edge is being removed between two vertices in the condensed graph. Please refer to Figure 5 for an example of this procedure.

Suppose u and v are vertices in the condensed graph, and the edge $u \rightarrow v$ is being removed. In this case, we do the following:

- 1) For each parent p of u :
 - a) Do a DFS search to test for the existence of a directed path from p to v . If there is no path, add $p \rightarrow v$. Note that again we did not call `fastTest()` here. We are conservatively building more reachability information into

the graph than is necessary, and we assume that the application will remove any unnecessary edges later.

- 2) For each child c of v :
 - a) Do a DFS search to test for the existence of a directed path from u to c . If there is no path, add $u \rightarrow c$.
 - 1) *Proof of Correctness*: This will be included in the final draft of this paper.
 - 2) *Time Complexity*: This will be included in the final draft of this paper.

VI. IMPLEMENTATION NOTES

The motivating application along with all the above transitive reduction algorithms were implemented in C++ using the logic synthesis system ABC [14]. The application was run on a number of benchmarks intended to simulate real-world chip designs, and we found that on average the transitive reduction was able to reduce the derived implication graph to less than 5% of its original size. This reduction not only reduced the memory requirements but also dramatically reduced the number of inductive proofs necessary.

The end result was a dramatic increase in the speed of the application and a dramatic decrease in the memory requirements. There is a runtime cost required to always maintain the transitive reduction, but in this application the main workhorse is a Boolean satisfiability (SAT) solver [15]. SAT solving is NP-complete, and so the cost of keeping the transitive reduction was negligible. We found that even with the new overhead, the simplifications to our SAT problem caused the application to run an order of magnitude faster.

VII. CONCLUSION

In this work we have proposed algorithms to maintain a transitive reduction of a possibly cyclic directed graph under vertex addition and edge deletion. The approach is novel in that it does not require the transitive closure to be explicitly computed, stored, or maintained.

We are motivated by a real-world application in logic synthesis, and while this application allows us to make some application-specific simplifications to our algorithm, we do not believe these simplifications to be prohibitive to a more general application of this theory.

REFERENCES

- [1] M.L. Case and R.K. Brayton and A. Mishchenko, “Inductively Finding a Reachable State Space Over-Approximation,” submitted to *International Workshop on Logic Synthesis, IWLS*, 2006

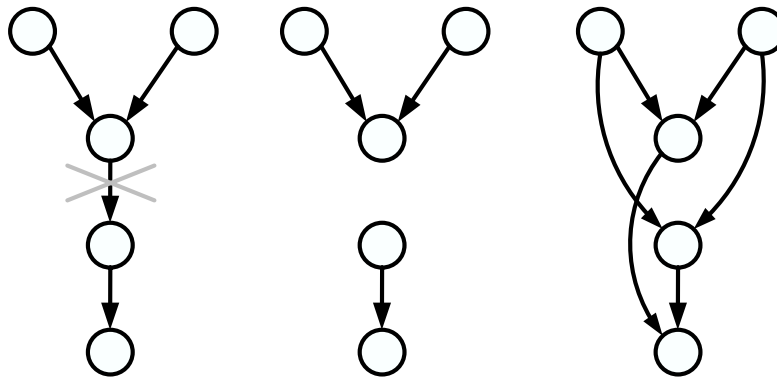


Fig. 5. Edge removal in the condensed graph. The edge $u \rightarrow v$ is removed and edges are inserted from all parents of u to v and from u to all children of v .

- [2] A.V. Aho and M.R. Garey and J.D. Ullman, "The transitive reduction of a directed graph," in *SIAM J. Comput.*, 1972, Pages 131-137
- [3] D. Gries and A.J. Martin and J.L. van de Snepscheut and J.T. Udding, "An algorithm for transitive reduction of an acyclic graph," in *Sci. Comput. Program.*, 1989. Pages 151-155
- [4] P. Chang and L.J. Henschen, "Parallel transitive closure and transitive reduction algorithms," in *Databases, Parallel Architectures and Their Applications. PARBASE-90, International Conference on*, 1990. Pages 152-154
- [5] S. Khuller and B. Raghavachari and N. Young, "Approximating the Minimum Equivalent Digraph," in *SIAM Journal of Computing*, 1995. Pages 859-872
- [6] K. Simon, "Finding a minimal transitive reduction in a strongly connected digraph within linear time," in *WG '89: Proceedings of the fifteenth international workshop on Graph-theoretic concepts in computer science*, 1990. Pages 245-259
- [7] R.E. Tarjan, "Depth-first search and linear graph algorithms," in *SIAM Journal on Computing*, 1972.
- [8] E. Nuutila and E. Soisalon-Soininen, "On finding the strongly connected components in a directed graph," in *Information Processing Letters*, 1994
- [9] R. Bloem and H.N. Gabow and F. Somenzi, "An Algorithm for Strongly Connected Component Analysis in $n \log n$ Symbolic Steps," in *Formal Methods in Computer Aided Design*, Springer-Verlag, 1994.
- [10] S. Khuller and B. Raghavachari and N. Young, "On strongly connected digraphs with bounded cycle length," in *Disc. Applied Math.*, 1996.
- [11] M.R. Henzinger and J.A. Telle, "Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning," in *Scandinavian Workshop on Algorithm Theory*, 1996. Pages 16-27
- [12] J.A. La Poutre and J. van Leeuwen, "Maintenance of transitive closures and transitive reductions of graphs," in *Proceedings of the International Workshop WG '87 on Graph-theoretic concepts in computer science*, 1988. Pages 106-120
- [13] J. van Leeuwen, "Graph algorithms," in *Handbook of theoretical computer science, vol. A: Algorithms and Complexity.*, MIT Press, Cambridge, MA, 1990. Pages 525-631
- [14] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [15] Niklas Een, Niklas Sorensson, MiniSat. [http://www.cs.chalmers.se/Cs/Research /FormalMethods/MiniSat/Main.html](http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html)