Maintaining A Minimum Equivalent Graph In The Presence of Graph Connectivity Changes

Michael L. Case and Robert K. Brayton

Department of EECS, University of California, Berkeley
{casem, brayton}@eecs.berkeley.edu

Abstract. Several authors have studied methods to construct a minimum equivalent graph, but little work has been done on how to maintain it. We are motivated by a real-world application which uses a directed graph at its core and depends on the edge set of that graph being small.
This paper presents an efficient method to maintain a minimum equivalent graph under edge addition and removal. We present methods to perform these operations without storing the original graph or its transitive closure, and we give proofs of correctness and the time complexities. We explore an application where trading of reduced edge count for the overhead of maintaining the graph dramatically improves overall performance.

1 Introduction

In any directed graph, if one is only interested in preserving reachability information, then there is flexibility in choosing the set of edges to represent. For example, in the simple graph $a \rightarrow b \rightarrow c$, a can reach c because there is a path from a to c. In this graph the presence of the edge $a \rightarrow c$ is optional. Since it does not add any new reachability information to the graph, depending on the application an implementer may choose to exclude it.

If the graph contains all optional edges, it is known as a transitive closure. The transitive closure has the maximal edge set, and this can dramatically increase the resources required to store it. However, checking for the existence of a path between two vertices is as simple as checking for the existence of a single edge.

Alternately, we may wish to minimize the number of edges in the graph by excluding all optional edges. This gives rise to the transitive reduction.

Definition 1 (Transitive Reduction). *The transitive reduction $r(G)$ of a directed graph G is a directed graph satisfying the following properties:*

1. *The vertex sets of G and $r(G)$ are identical.*
2. *There is a directed path from vertex u to vertex v in G if and only if there also one in $r(G)$.*

3. There is no graph with fewer edges than $r(G)$ satisfying the above conditions.

Since the set of edges is minimized, storage requirements are also minimized. However, the induced graph sparsity can increase the time required to check for the existence of a path between two vertices. In both memory and runtime, the transitive reduction has characteristics opposite the transitive closure. Unfortunately, while the transitive closure has been extensively studied, the transitive reduction has received less attention in the literature.

We are motivated by an application in the field of logic synthesis. It utilizes a conservative approximation to a transitive reduction to maintain internal data-structures, benefiting from the reduced memory requirements. Also, since a large amount of work must be done in processing the facts that the edge set represents, by reducing the number of edges the performance of the logic synthesis tool is dramatically improved.

To support this application, new algorithms have been developed to efficiently maintain a conservative approximation of the transitive reduction. It is maintained in an online manner through any number of edge addition and removal operations. This paper gives the algorithms, their complexities, proofs of correctness, and some experimental results to empirically verify the theoretic results.

2 Motivating Application

In logic synthesis, a design for a digital computer chip is given which is to be optimized. It is given as a graph of logical components, and by manipulating this graph the chip can be optimized to use fewer components, faster components, and/or ones that use less power.

The chip usually contains some amount of memory which stores the state of the system implemented by the chip design. 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. Knowledge of which states are unreachable can be used to change the behavior of the system on these unreachable states thereby simplifying the design. Unfortunately, this simplification requires knowledge of the set of states that the system can reach, and currently no scalable algorithms exist to find this set 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 $a(s) \Rightarrow b(s)$ over pairs of logical functions $a(s)$ and $b(s)$ present in the design's logic network. Here

s is the state of the machine. We prove by induction that the implication holds in every reachable state. If we succeed in proving this for a set of implications I in a design, then the Boolean function $R(s)$ defined as

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

provides an over-approximation to the set of reachable states. In practice, this approximation is close enough to the actual reachable state set to enable optimization, and it is fast to compute.

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

It is vitally important to reduce this "implication graph" as much as possible. The number of candidate implications that might be attempted to be proved can be excessively large, and to fit the resulting graph into memory requires transitive reduction. More importantly, each implication in the graph must be proved, and this proof can be a very expensive operation. Using the transitive reduction reduces the number of required proofs, dramatically improving the performance of the tool.

3 Related Work

Aho et. al. [2] originally defined transitive reduction, and showed that is unique for an acyclic graph. Also a method for constructing it was given.

More efficient constructions, [3] [4], exist. Some authors, [5] [6], prefer to address a related problem called the minimum equivalent graph problem where one attempts to find a graph related to a transitive reduction in which the edge set is constrained to be a subset of the original edge set. These papers provide motivation for our solution.

Identification of strongly connected components (SCCs) is key to existing transitive reduction methods, and the SCC has been extensively studied in [7] [8] [9] [10] [11].

La Poutre et. al. [12] studied a way to maintain the transitive reduction of a graph under edge insertion and deletion, similar to our work. Unfortunately their method requires one to store and simultaneously update the transitive closure as well, making the method impractical. The methods presented here are free of such requirements.

4 Approximating the Transitive Reduction

Due to problems that arise in the online maintenance of a true transitive reduction, we choose to approximate the transitive reduction.

Consider the directed graph G and its transitive reduction $r(G)$ shown in Figures 1 and 2. The reduction can be built by first greedily removing redundant edges. This eliminates the need for $C \rightarrow G$, $B \rightarrow F$, and $D \rightarrow G$. For all of these edges, there exists an alternate path joining the pair of vertices, and this path makes the directed edge redundant. The next step in creating the transitive reduction is to identify all SCCs and replace each with a simple cycle. This simple cycle maintains the connectedness of the component with the minimum number of edges. In the figure the SCC $\{A, B, C\}$ has been thus processed.

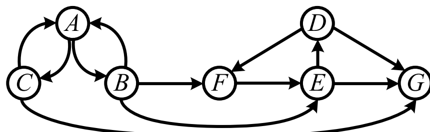

Fig. 1. An example graph G.

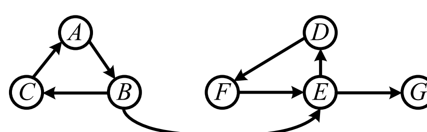

Fig. 2. Corresponding $r(G)$.

The transitive reduction is simple to build, but difficult to maintain. Suppose that we now wish to remove the edge $A \rightarrow B$ in the reduction shown in Figure 2. In the resulting graph, A and B should be placed in separate components, but it is unclear to which component C should be assigned. There is not sufficient information in $r(G)$ to refine this component.

The solution to this problem is to relax the constraint that components be joined by simple cycles. This leads us to explore an approximation called the minimum equivalent graph.

Definition 2 (Minimum Equivalent Graph (MEG)). *Given a directed graph G, the MEG $meg(G)$ is a transitive reduction satisfying the following:*[1]

1. $edges(meg(G)) \subseteq edges(G)$.
2. $\forall$ *edges* $u \rightarrow v$ *in* $meg(G)$, $\nexists$ *an alternate simple path* $u \rightarrow v$ *in* $meg(G)$.

5 Maintaining the Minimum Equivalent Graph

In this section we develop online algorithms to incrementally maintain an MEG under edge addition and removal.

[1] If G is acyclic then the $meg(G) = r(G)$.

5.1 Edge Addition

Consider the addition of an edge to an MEG. This edge can introduce any number of new paths which may make other edges redundant. Any edge addition algorithm must identify and remove these now-redundant edges.

Algorithm 3 takes redundant edges into account to maintain the MEG under edge addition.

```
1: // a→b := Edge to be added
2: if ∄ path from a to b then
3:    Add edge a→b
4:    Color ancestors of a red
5:    Color descendants of b blue
6:    for all edges from a red r
             to a blue b do
7:       if ∃ simple path r→b
             through a→b then
8:          Remove r→b
9:       end if
10:   end for
11: end if
```

Algo. 3. Edge addition algorithm.

```
1: // a→b := Edge to be removed
2: Remove edge a→b
3: for all parents p of a do
4:    Add p→b with algo. 3
5: end for
6: for all children c of b do
7:    Add c→c with algo. 3
8: end for
```

Algo. 4. Edge removal algorithm.

An example application of this algorithm is shown in Figure 5. The left-most graph is an MEG to which we will add edge $C \rightarrow D$. To identify the edges that are made redundant, we color the ancestors of C and the descendants of D. The edges $A \rightarrow D$ and $C \rightarrow E$ are between colored vertices, and both of these edges have alternate simple path through $C \rightarrow D$. The two edges are redundant, and they are removed to make the output MEG. Because we are maintaining an MEG, adding one edge caused the total number of edges to decrease by 1 while the reachability information content increased.

Theorem 1 (Correctness of the addition algorithm). *If the input to Algorithm 3 is an MEG then the output is also an MEG.*

To analyze the time complexity of Algorithm 3, suppose the input MEG has v vertices and e edges. On line 2, path existence can be implemented as a depth-first search which has complexity $O(v+e)$. Adding the edge on line 3 can be done in constant time, if implemented properly. Coloring sets of vertices on lines 4-5 can be done with two more depth-first searches. In lines 6-10, a path existence check and possibly an edge removal must be done for each edge. This dominates all other steps with complexity $O(e \cdot ((v+e)+1)) = O(ev + e^2)$. Thus the complexity of Algorithm 3 is $O(ev + e^2)$.

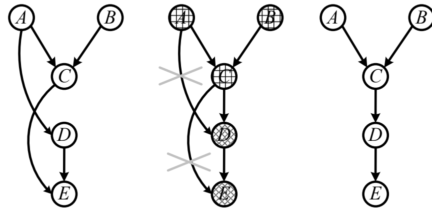

Fig. 5. Example edge addition.

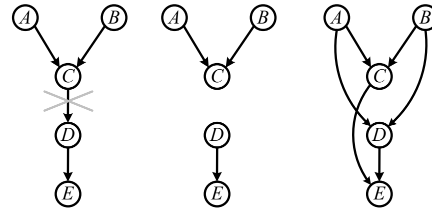

Fig. 6. Example edge removal.

5.2 Edge Removal

Removal of an edge from an MEG is conceptually harder and has higher complexity than edge addition, but the algorithm is simpler. In an MEG a single edge may lie on multiple paths in the graph. These paths represent reachability information, and the removal of an edge must not violate this reachability. Therefore, edge removal involves both the identification of these disturbed paths as well as the addition of edges to preserve the paths in the absence of the now removed edge.

Algorithm 4 effectively removes an edge without disturbing any other paths in the MEG.

An example application of this algorithm is shown in Figure 6. The left-most graph is an MEG from which we wish to remove the edge $C \rightarrow D$. Note that in this graph, A can reach $\{D, E\}$, but removal of $C \rightarrow D$ disturbs this. Several other reachability relationships are similarly disturbed. We can maintain the graph by adding the edges $A \rightarrow D$, $B \rightarrow D$, and $C \rightarrow E$, as described in lines 3-8 of the algorithm. Because we maintain an MEG, removal of an edge caused the total number of edges to increase by 2 while the total reachability information decreased.

The use of Algorithm 3 as a subroutine to Algorithm 4 may at first seem unnecessary, but it is vitally important in order to maintain the MEG. Consider Figure 7 as an example. Removal of edge $A \rightarrow B$ causes $A \rightarrow C$ to be added which makes $B \rightarrow C$ redundant. Unless we use Algorithm 3 for edge addition, we will not detect this redundant edge and the output will not be an MEG.

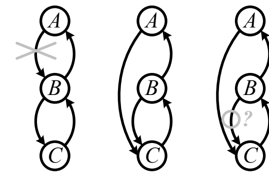

Fig. 7. Edge removal needs Algo. 3.

Theorem 2 (Correctness of the removal algorithm). *If the input to Algorithm 4 is an MEG then the output is also an MEG.*

Consider now the time complexity of Algorithm 4. Let the input MEG have v vertices and e edges. The edge removal on line 2 can be done in constant time. In lines 3-8 we call Algorithm 3 possibly v times for a complexity

of $O(v \cdot (ev + e^2)) = O(ev^2 + e^2 v)$. The total complexity of our edge removal algorithm is $O(ev^2 + e^2 v)$.

6 Experimental Results

The graph algorithms described in this paper were implemented in a C++ library which was highly tuned for maximum performance. The results are shown below.

In addition, the motivating logic synthesis application was also studied. This application is implemented as a plugin to the logic synthesis system ABC [14], and the MEG library is incorporated as the implication graph manager. The performance in this setting is studied as well.

6.1 Graph Theoretic Results

In this first set of results, the MEG library is run by itself using a small driver application to generate a sequence of random edge additions and removals. This sequence is sufficiently long to provide reliable statistics, and the number of vertices in the graph is varied to expose the underlying sensitivity to this parameter. The performance is compared against a normal graph package which just updates the graph and does no reduction whatsoever.

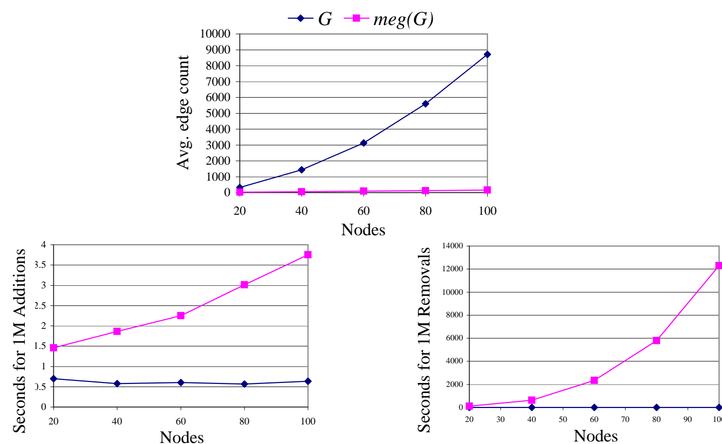

Fig. 8. Empirical Analysis of the Algorithm.

Figure 8 shows the average number of edges present in the graph. It shows this for both the version maintaining $meg(G)$ and that maintaining G, and it shows this over a range of vertex set sizes. From this, we can see that the MEG has dramatically fewer edges, and the difference grows with the vertex set size. This was the original motivation for our study of the transitive reduction and MEG.

Figure 8 also shows the runtime performance of the addition algorithm by showing the normalized time for 1,000,000 edge additions[2]. Recall that the edge addition algorithm has complexity $O(ev + e^2)$, and we can see this linear dependence on v in the figure. For this algorithm, the difference between the two graph implementations is not very large.

The edge removal is studied in the last graph in Figure 8 where the normalized time for 1,000,000 edge removal operations is shown. Recall that we expect complexity $O(ev^2+e^2v)$, and the graph shows that the performance is roughly quadratic in the number of vertices. The performance gap between the two graph packages is great, and the cost of maintaining the MEG under edge removal is non-trivial. Finding a more efficient edge removal algorithm could be the focus of further research.

As expected, the tests show great savings in storage but increased runtime. We need to look at a particular application to see any benefit. In these tests, our driver application has no control over the number of edges present in its random graph, and the complexities of our algorithms depend greatly on this parameter. In the following experiments we examine the performance of the algorithms in a real application with vertex and edge numbers as they would occur naturally. This gives a much more informative picture of the true performance.

6.2 Practical Results

In our motivating logic synthesis application the two graph libraries, maintainers of $meg(G)$ and G respectively, were used as the implication graph manager. The code is structured such that the user may select which graph library to use.

Using this framework, we ran the tool on 15 circuit designs: 10 small academic designs and 5 obtained from industry. The tool was run using each graph library, and the total runtime and memory was recorded.

In the following graphs, the x-axis gives the measured quantities for the $meg(G)$ package, and the y-axis gives the quantities for the G package. Each point is a single design as measured under both implementations. If the point appears above the diagonal, the $meg(G)$ implementation was better than the G implementation, and the performance gap corresponds to the distance above (or below) the diagonal.

Figure 9 shows the runtimes of the logic synthesis tool. Note that the runtime of the tool with the $meg(G)$ package is roughly constant over these 15 designs, and in almost all cases the tool is made faster by using $meg(G)$. However, this figure is not a true indication of the performance gap because

[2] All tests were run on a 1.6 GHz Pentium-M laptop.

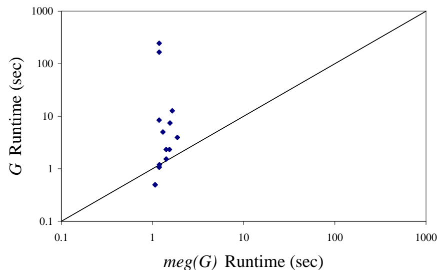

Fig. 9. Runtime comparison.

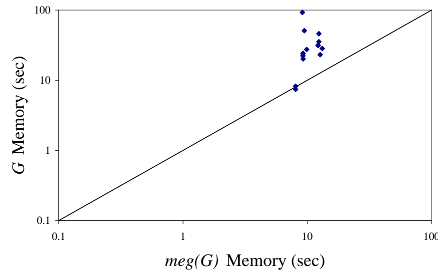

Fig. 10. Memory comparison.

in running the experiments, it was necessary to terminate the flow of the tool early. Without this early termination, the tool with G failed to run in less than 10 minutes (on 13 designs) while the tool with the $meg(G)$ package took roughly the same amount of time as shown. Thus the performance gap between the two implementations is very large and grows as the cutoff time limit is increased.

It is interesting to understand why the tool runs faster when using $meg(G)$. Although using this reduction introduces polynomial-time overhead in the maintenance routines, for each edge in the graph, the tool must prove the implication the edge represents. This proof can be exponential in complexity, and the reduced edge set of the MEG allows a trade-off between exponential and polynomial complexity. This enables the dramatic performance improvements. In fact, without these improvements, this logic synthesis tool is impractical, and the MEG is what enables it to be run at all.

Figure 10 shows the peak memory allocated by the logic synthesis tool. This is recorded for both graph packages over the 15 circuit designs. Again we see that the memory used by the tool with the MEG package is roughly constant, and in nearly all cases this beats the memory used with the normal package. The performance gap in memory is not always large because we focus on small designs where the implication graph typically does not dominate the memory consumption. With larger designs, we have observed that without the MEG package we cannot fit the application into our memory space, making the logic synthesis tool impractical.

7 Conclusion

This work contributes several graph theoretic results. Algorithms to maintain the MEG in an online manner under edge addition and removal were given. The correctness was proved, and the time complexities were derived.

These algorithms were implemented and studied in an experimental setting, verifying the theoretic results. In a practical application, the study shows how the MEG can be extremely beneficial.

Not only does this work improve performance of our application, but it also enables previously unexplored avenues in logic synthesis. Generally, in applications where each edge represents an additional amount of computation, the trade-off of fewer edges for increased time to maintain a graph is worthwhile and potentially enabling.

References

1. M.L. Case and R.K. Brayton and A. Mishchenko, "Inductively Finding a Reachable State Space Over-Approximation," *International Workshop on Logic Synthesis, IWLS*, 2006
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. N. Een, N. Sorensson, MiniSat. http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/

Appendix: Proofs of Correctness

Theorem 1 (Correctness of the addition algorithm). *If the input to Algorithm 3 is an MEG then the output is also an MEG.*

Proof. Let $a \rightarrow b$ be the edge added by Algorithm 3, and let the input and output graphs be given by I and O respectively. We assume I is an MEG and now proceed to show that O is an MEG.

Consider $a \rightarrow b \in O$. If I had a path from a to b then $O = I$ (by line 2) and so is an MEG. Therefore assume I had no such path. Algorithm 3 adds only one edge $(a \rightarrow b)$, and so is incapable of forming an alternate simple path from a to b.

Now consider all $u \rightarrow v \in O$ such that $u \neq a$ or $v \neq b$. Because Algorithm 3 adds only $a \rightarrow b$, we must have $u \rightarrow v \in I$. Because I is an MEG, there is no alternate simple path from u to v. Suppose algorithm 3 introduced such an alternate path. Then $a \rightarrow b$ would have to be on this path since all new paths go through this edge. In this case, $u \rightarrow v$ is removed in line 8 and so could not possibly be in O. By contradiction, $u \rightarrow v$ has no alternate simple path in O.

Because no edge in O has an alternate simple path, O is an MEG. □

Theorem 2 (Correctness of the removal algorithm). *If the input to Algorithm 4 is an MEG then the output is also an MEG.*

Proof. Algorithm 4 can be subdivided into two parts: removal of the specified edge and calls to Algorithm 3.

Since the input is an MEG, and because removal of an edge cannot introduce any paths in the graph, the graph after the edge removal is an MEG.

By Theorem 1 we know the graph after calls to is also a Algorithm 3 is also an MEG. □