# Online Algorithms To Maintain A Transitive Reduction

CS 294-8 Class Project, Fall 2006

Michael L. Case
Department of EECS, University of California, Berkeley
casem@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 an approximation to the transitive reduction of a possibly cyclic directed graph under edge addition and removal. 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

In any directed graph, if one is only interested in preserving the reachability information, then there is an amount of flexibility in choosing the set of edges to represent . Take for example the simple graph $a \rightarrow b \rightarrow c$. Clearly, $a$ can reach $c$ because there is a path from $a$ to $c$. Note that in this graph the presence of the edge $a \rightarrow c$ is entirely optional. It doesn't add any new reachability information to the graph, and depending on the application, the implementer may or may not choose to include this edge.

If the graph contains all optional edges, it is known as a transitive closure:

*Definition 1 (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 $c(G)$.

The transitive closure has a maximal edge set, and this can dramatically increase the resources required to store it. However, note that checking for the existence of a path between two vertices is equivalent to checking for the existence of a single edge. For this reason, the transitive closure is often used in database applications where it is referred to as the minimum query time representation.

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 2 (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 then $r(G)$ satisfying the above conditions.

As can be expected, because the set of edges is minimized, the storage requirements are also minimized. There is an induced graph sparsity that can increase the time required to check for the existence of a path between two vertices. In this sense, the transitive reduction can be thought of as the maximum query time representation. Therefore in both memory and runtime, the transitive reduction is the exact opposite of the transitive closure. Unfortunately, while the transitive closure has been extensively studied, the transitive reduction has been virtually ignored in the literature.

We are motivated by an application in the field of logic synthesis. Our application utilizes a transitive reduction to maintain internal data-structures, benefiting from the reduced memory requirements. Also, in this tool a large amount of work must be done in processing the facts that the edge set represents, and 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 approximate the transitive reduction. This approximation 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 finally some

experimental results to empirically verify the theoretic results.

## II. 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 $a(s) \Rightarrow b(s)$ 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 a set of implications $I$ in a design, then we build a Boolean function $R(s)$ over the system state such that

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

Any state $s$ for which $R(s) = 1$ is necessarily reachable, and so $R(s)$ 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 to compute.

The process described above involves proving logical implications. A set of implications can be thought of as forming a directed 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*.

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

## III. 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.

## IV. Approximating the Transitive Reduction

Due to problems that arise in the online maintenance of a true transitive reduction, we find it necessary to approximate the transitive reduction in this graph.

Consider the example of a graph and its transitive reduction shown in figure 1. The reduction can be built by first greedily removing redundant edges. This eliminates the need for $C \to G$, $B \to F$, and $D \to G$. For all of these edges, there exists an alternate path joining

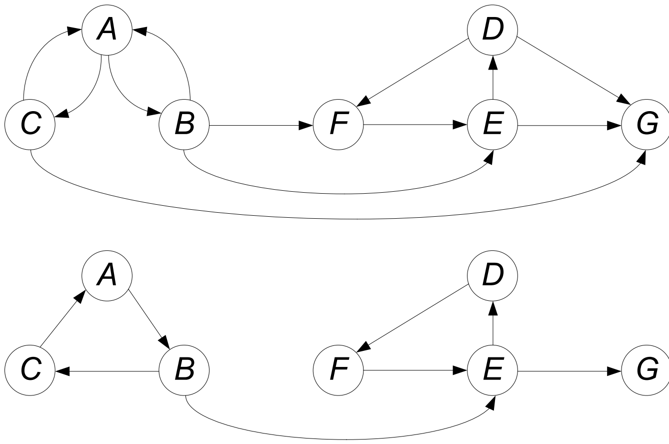


Fig. 1. An example transitive reduction.

the pair of vertices, and this path makes the directed edge redundant. The next step in creating the transitive reduction is to identify all strongly connected components and then replace any existing interconnections with a simple cycle. This simple cycle maintains the connectedness of the component with the minimum number of edges. In the figure note that the strongly connected component $\{A, B, C\}$ has been this processed.

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 1. 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 to refine this component, and without external help we cannot proceed.

The solution to this problem is to relax the constraint that components be joined by simple cycles. This gives us a graph that we will refer to as the pseudo transitive reduction (PTR):

*Definition 3 (Pseudo Transitive Reduction (PTR)):* Given a directed graph $G$, the pseudo transitive reduction $ptr(G)$ is a directed graph satisfying the following properties:

1) The vertex set of $G$ is equal to the vertex set of $r(G)$.
2) $\forall$ edges $u \rightarrow v$, $\nexists$ an alternate simple path $u \rightarrow v$.

(Note that if the graph is acyclic then the PTR is the same as the transitive reduction.)

## V. Maintaining the Pseudo Transitive Reduction

In this section we develop online algorithms to maintain a PTR under edge addition and removal.

### A. Edge Addition

Consider the addition of an edge to a PTR. This edge can introduce any number of paths in the graph, and these paths may make some of the edges redundant. Therefore any edge addition algorithm must identify and remove these now-redundant edges.

Algorithm 1 takes takes redundant edges into account to maintain the PTR under edge addition.

**Algorithm 1** Edge addition algorithm.

```
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
```

An example application of algorithm 1 is shown in figure 2. The left-most graph is a PTR to which we will add edge $C \rightarrow D$. To help us 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 a quick check verifies that for both of these edges, there exists an alternate simple path through $C \rightarrow D$. These makes these two edges redundant, and they are removed to make the output graph a PTR. Note that because we are maintaining a PTR, adding one edge caused the total number of edges to decrease by 1. The reachability information content of the graph increased, however.

*Theorem 1 (Correctness of the addition algorithm.):* If the input to algorithm 1 is a PTR then the output is also a PTR.

*Proof:* Let $a \rightarrow b$ be the edge added by algorithm 1, and let the input and output graphs be given by $I$ and $O$ respectively. We assume $I$ is a PTR and now proceed to show that $O$ is a PTR.

Consider $a \rightarrow b \in O$. If $I$ had a path from $a$ to $b$ then $O = I$ and so is a PTR. Therefore assume $I$ had no such path. Algorithm 1 adds only one edge ($a \rightarrow b$), and so is

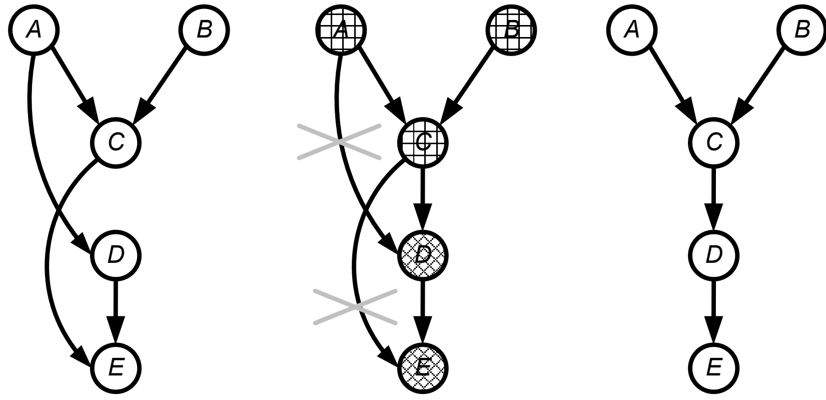


Fig. 2. Example edge addition.

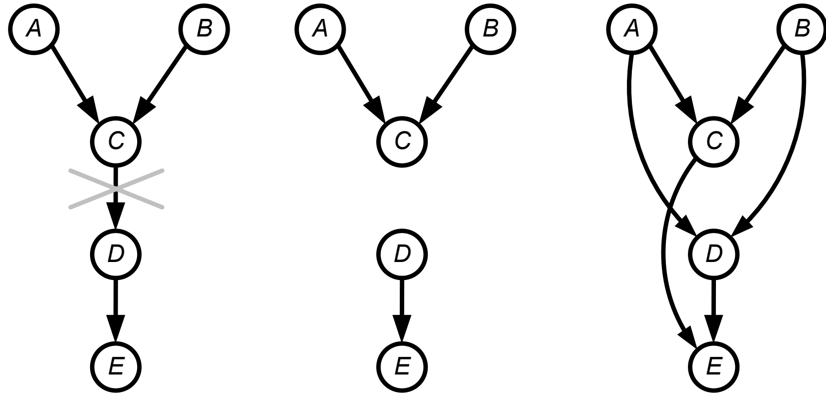


Fig. 3. Example edge removal.

incapable of forming an alternate simple path from $a$ to $b$.

Now consider all $u \rightarrow v \in O$ such that $u \neq a$ and $v \neq b$. Because algorithm 1 adds only $a \rightarrow b$, we must have $u \rightarrow v \in I$. Because $I$ is a PTR, there is no alternate simple path from $u$ to $v$. Suppose algorithm 1 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 algorithm 1 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 a PTR. ■

We now proceed to analyze the time complexity of algorithm 1. Suppose the input PTR 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 the final section in lines 6-10, we must for each edge do a path existence check and possibly an edge removal. This dominates all other steps with complexity $O(e \cdot ((v+e)+1)) = O(ev + e^2)$. Finally, we see that the complexity of algorithm 1 is $O(ev + e^2)$.

### B. Edge Removal

Removal of an edge from a PTR is conceptually slightly harder than edge addition, but the algorithm ends up being simpler. Note that in a PTR a single edge may lie on multiple paths in the graph. These paths represent reachability information, and the removal of an edge must not disturb this reachability. Therefore, edge removal involves identification of these disturbed paths and the addition of edges to preserve the paths in the absence of the now removed edge.

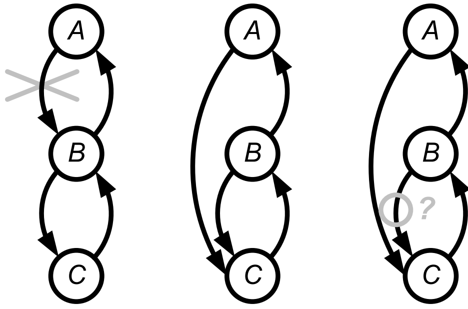


Fig. 4. Edge removal can make other edges redundant.

Algorithm 2 effectively removes an edge without disturbing any other paths in the PTR.

**Algorithm 2** Edge removal algorithm.

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

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

The use of algorithm 1 as a subroutine to algorithm 2 may at first seem unnecessary, but it is vitally important in order to maintain the PTR. Consider figure 4 as an example. Removal of edge $A \to B$ causes algorithm 2 to add the edge $A \to C$. This edge makes $B \to C$ redundant. Unless we use algorithm 1 to add edge $A \to C$, we will not detect this redundant edge and the output will not be a PTR.

*Theorem 2 (Correctness of the removal algorithm.):* If the input to algorithm 2 is a PTR then the output is also a PTR.

*Proof:* Algorithm 2 can be subdivided into two parts: removal of $a \to b$ and calls to algorithm 1.

Since the input to the first part is a PTR, and because removal of an edge cannot introduce any paths in the graph, the output of the first part is also a PTR.

The input to the second part is a PTR, and by theorem 1 we know the output is also a PTR. ∎

Consider now the time complexity of algorithm 2. Again, let the input PTR 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 1 possibly $v$ times for a complexity of $O(v \cdot (ev + e^2)) = O(ev^2 + e^2v)$. From this we can see that the total complexity of our edge removal algorithm is $O(ev^2 + e^2v)$.

## VI. Experimental Results

The graph algorithms described in this paper were implemented in a C++ library. This library was tested by itself and 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]. The PTR library is incorporated into this tool as the implication graph manager. The benefits that the PTR provides in this application are studied below as well.

### A. Graph Theoretic Results

In this first set of results, the PTR library is run by itself in a synthetic environment. A small driver application was developed that simply generates a sequence of random edge additions and removals. This sequence is 200,000 operations long in order to make all average measurements more accurate. In addition, the number of vertices in the graph is varied to expose the sensitivity of the underlying algorithms to this parameter. The performance is compared against a normal graph package that does no reduction whatsoever.

Figure 5 shows the average number of edges present in the graph. It shows this for both the reduced (PTR) graph library and the normal library, and it shows this over a range of vertex set sizes. From this graph we can see that the PTR has dramatically fewer edges, and the difference grows with the vertex set size. This dramatic difference was the original motivation for studying the transitive reduction.

Figure 6 shows the runtime performance of the addition algorithm. The code was profiled, and the total amount of time taken in all addition operations was recorded. This was normalized to arrive at the average amount of time taken to do 1,000,000 edge additions [1].

[1] All tests were run on my laptop, a 1.6 GHz Pentium-M.

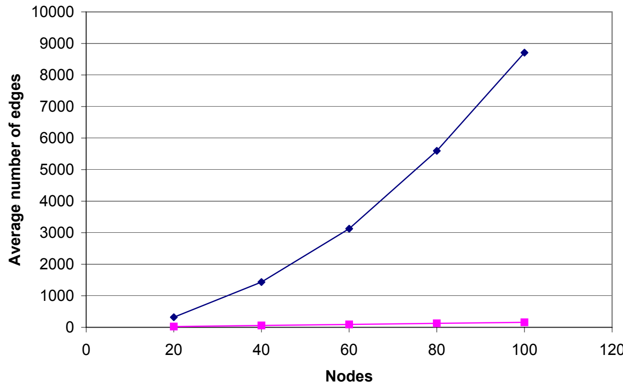


Fig. 5. Average number of edges.

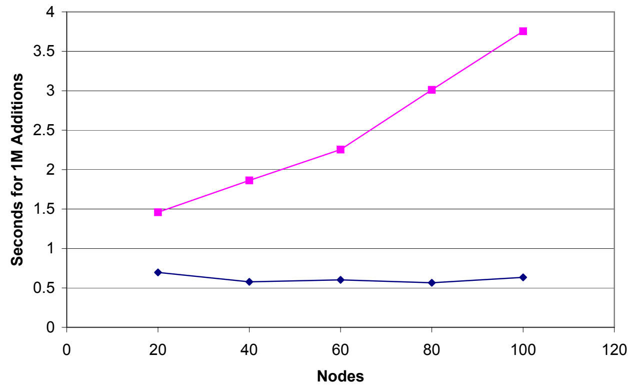


Fig. 6. Average edge addition time.

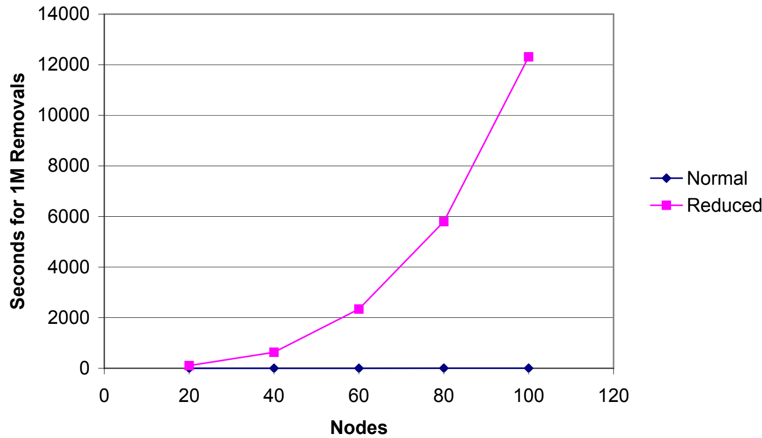


Fig. 7. Average edge removal time.

Recall that the edge addition algorithm has complexity $O(ev + e^2)$, and we can see from the figure that the runtime is nearly linear in the number of vertices. Also note that the normal graph library can add edges in nearly constant time. However, the difference between the two graph implementations is not very large.

The second algorithm, edge removal, is studied in figure 7. Again, the total runtime is averaged and normalized to 1,000,000 edge removal operations. Recall that we expect complexity $O(ev^2+e^2v)$, and we can see from the graph that the performance is roughly quadratic in the number of vertices. We can also see that the performance gap between the two graph packages is great, and so the cost of maintaining the PTR under edge removal is significant. Finding a more efficient edge removal algorithm could be the focus of further research.

These tests of the graph algorithms unfortunately do not tell the whole story. In these tests, a random sequence of edge additions and removals is generated. The driver application has no control over the number of edges present in the graph, and the complexity of our algorithms depends greatly on this parameter. In the following experiments we examine the performance of this algorithm in a real world application with vertex and edge numbers as they would naturally occur. This gives a much more informative picture of the true performance.

## B. Practical Results

We now return to our motivating logic synthesis application. The two graph libraries, PTR and normal, were incorporated into the implication proving tool and 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 run the tool on 15 circuit designs: 10 small academic designs and 5 circuits ob-

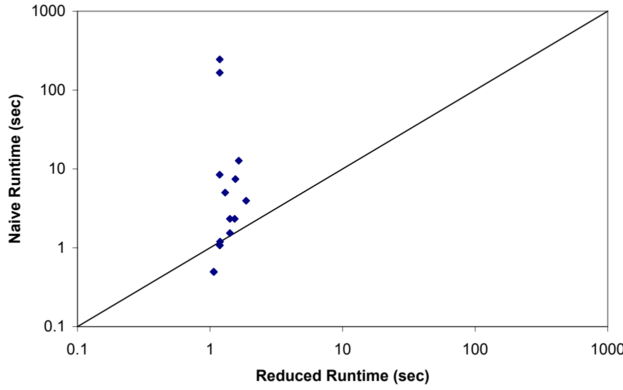


Fig. 8. Runtime comparison.

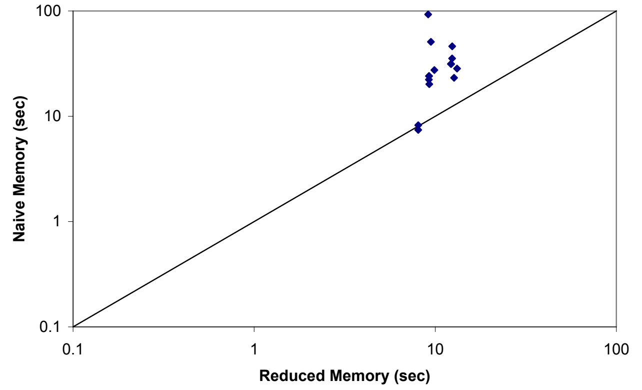


Fig. 9. Memory comparison.

tained from industry. The tool was run once with each graph library, and the total runtime and memory was recorded.

In the graphs recording these results, the x-axis gives the measured quantities for the PTR package, and the y-axis gives the quantities for the normal package. Each point is a single design as measured under both implementations. If the point appears above the diagonal, it means that the PTR implementation was better than the normal implementation, and the performance gap corresponds to the distance above (or below) the diagonal.

Figure 8 shows the runtimes of the logic synthesis tool. Note that the runtime of the tool with the PTR package is roughly constant over these 15 small designs, and in almost all cases the tool with the PTR package is much faster than the tool with the normal package. This figure shows the performance gap, but it doesn't tell all. In running this experiment, I found it necessary to terminate the flow of the tool early in order to force speedy execution. Without this early termination, the tool with the normal graph package failed to run in less than 10 minutes (on 13 designs) while the tool with the PTR package took roughly the same amount of time as shown. This shows that the performance gap between the two implementations is huge, and it grows as my tool is allowed to run for longer. Even with this early termination, the size of the vertex and edge sets should be representative and these results valid.

It is interesting to explore why the tool should run faster when using the PTR package. We know that the PTR package introduces polynomial-time overhead necessary to maintain the PTR. However, for each edge in the graph, the tool must prove the implication the edge represents, and this proof can be exponential in complexity. Using the PTR package allows us to trade exponential complexity for polynomial, and this enables very dramatic performance improvements. In fact, without these improvements, this logic synthesis tool is impractical, and the PTR is what enables it to be run at all.

Figure 9 shows the peak memory allocated by the logic synthesis tool. Again, this is recorded for both graph packages over the 15 circuit designs. Again we see that the memory used by the tool with the PTR package is roughly constant, and in nearly all cases this beats the memory used with the normal package. It is also interesting to note that the performance gap is not always large. Because we focus on small designs, the memory consumed in the graph is not a large portion of the peak memory consumption. This tends to reduce the gap shown in this figure. With larger designs, we have observed that without the PTR package we cannot fit the application into our memory space. Again we see that the logic synthesis tool is impractical without the PTR.

## VII. Conclusion

This work contributes several graph theoretic results. The PTR was defined, and algorithms were given to maintain it in an online manner under edge addition and removal operations. Proofs of correctness as well as complexities for these algorithms were given.

These algorithms were implemented and studied in an experimental setting. This verifies the theoretic results. Also, the algorithms were used in a practical application, and this shows how the PTR is be beneficial in a real-world application.

The PTR was shown to be vitally important in this logic synthesis application. Without the PTR, the motivating application is impractical. Therefore, not only

does this work embody an advance in graph theory, but it also enables previously unexplored avenues in logic synthesis.

## 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] Niklas Een, Niklas Sorensson, MiniSat. http://www.cs.chalmers.se/Cs/Research /FormalMethods/MiniSat/Main.html