In J. of Algorithms, vol. 12, 1991, pp. 110-125

TRANSITIVE COMPACTION IN PARALLEL VIA BRANCHINGS

Phillip Gibbons[*] *Richard Karp*[*,†]
Computer Science Division, University of California, Berkeley, CA
Vijaya Ramachandran[*,‡]
Coordinated Science Lab., University of Illinois, Urbana, IL
Danny Soroker[*]
IBM Almaden Research Center, San Jose, CA
Robert Tarjan[§]
Computer Science Dept., Princeton University, Princeton, NJ
and AT&T Bell Labs., Murray Hill, NJ

July 15, 1988

ABSTRACT

We study the following problem: given a strongly connected digraph, find a minimal strongly connected spanning subgraph of it. Our main result is a parallel algorithm for this problem, which runs in polylog parallel time and uses $O(n^3)$ processors on a PRAM. Our algorithm is simple and the major tool it uses is computing a minimum-weight branching with zero-one weights. We also present sequential algorithms for the problem that run in time $O(m + n \cdot \log n)$.

---

[*] Supported in part by the International Computer Science Institute, Berkeley, California.
[†] Also supported by NSF grant CCR-8411954.
[‡] Also supported by Joint Services Electronics Program under N00014-84-C-0149.
[§] Supported in part by NSF grant DCR-8605962 and ONR contract N00014-87-K-0467.

1. Introduction

The transitive compaction problem for strongly connected digraphs is: given a strongly connected digraph G, find a minimal strongly connected spanning subgraph of it, i.e., a strongly connected spanning subgraph for which the removal of any arc destroys strong connectivity. We are looking for a minimal subgraph because the problem of finding a *minimum* subgraph with the same transitive closure is NP-hard [GJ].

There is an obvious sequential algorithm for solving this problem: scan the arcs one by one; at step i test if the i-th arc can be removed without destroying strong connectivity. If so, remove it and update the digraph. This algorithm has complexity $O((n+m)^2)$, where n is the number of vertices of the input graph and m is the number of arcs. A simple modification is to initially reduce the number of arcs to at most $2n-2$ by taking the union of a forward and an inverse branching (defined below). This reduces the running time to $O(n^2)$.

The problem studied here is reminiscent of the well-studied problem of finding a maximal independent set of vertices in a graph, for which several parallel algorithms have appeared in the literature ([KW],[Lu],[ABI],[GS]). Two common features are that there is a simple sequential algorithm for it that seems hard to parallelize and that the related optimization problem (minimum vs. minimal) is NP-hard.

We can define the following independence relation on the arcs of a strongly connected digraph, G: a set of arcs is independent if it can be removed without destroying strong connectivity of G. Using this definition, finding a transitive compaction of G is equivalent to removing a maximal independent sets of arcs from G. A property that sets our problem apart from the maximal independent set problem is that in our case independence of a set is not guaranteed when every pair of elements in it is independent.

Our problem can be expressed as the determination of a maximal independent set in an independence system as defined by Karp, Upfal and Wigderson ([KUW]). The problem computed by a "rank oracle" in this case is NP-hard, but an "independence oracle" is easy to compute in NC. Following the method described in [KUW] this automatically yields a *randomized* parallel algorithm that uses a polynomial number of processors and runs in time $O(\sqrt{n} \cdot \log^c n)$ (for some constant c).

In this paper we present parallel and sequential algorithms for this problem. Our first parallel algorithm runs in time $O(\log^5 n)$ and uses $O(n^3)$ processors on a CREW PRAM. We then present an improved implementation of one of the steps in the algorithm that leads to a parallel algorithm that runs in $O(\log^4 n)$ time with the same processor bound. Both of these algorithms can be speeded up by a $\log n$ factor if we use a CRCW PRAM; we assume here the COMMON concurrent-write model in which all processors participating in a concurrent write must write the same value [KR]. The processor bound of $O(n^3)$ represents the number of processors needed to multiply two n by n matrices in $O(\log n)$ time on a CREW PRAM by the straightforward parallel matrix multiplication algorithm. It is possible that the processor bound can be improved by using sophisticated techniques for multiplying n by n matrices (see e.g., [CW]); we do not elaborate on this.

The major tool that our algorithms use is computing a minimum-weight branching with zero-one weights. Central to our algorithms is a proof that two suitable applications of this tool are guaranteed to reduce by half the number of arcs still to be removed. We also present two sequential algorithms for the problem, each of which runs in time $O(m + n \cdot \log n)$. This is an improvement over the straightforward algorithm mentioned above.

The transitive compaction problem is, in some sense, a dual of the minimum strong augmentation problem - add a minimum set of arcs to a digraph to make it strongly connected. A linear time sequential algorithm was given for this problem by Eswaran and Tarjan ([ET]), and a parallel algorithm running in $O(\log n)$ time with $O(n^3)$ processors on a CRCW PRAM was given by Soroker ([So]).

Our problem extends naturally to general digraphs: given a digraph G, find a minimal spanning subgraph of it whose transitive closure is the same as that of G. A sequential algorithm for this problem in the case that G is acyclic is given in [AGU] and can be parallelized in a straightforward manner. Combining it with our algorithms we obtain parallel algorithms (with the same complexities as stated above) for the transitive compaction problem on general digraphs. We point out that these parallel algorithms are good with respect to the state of the art, since the problem solved is at least as hard as testing reachability from one vertex to another in a digraph, and the best NC algorithm currently known for this requires on the order of $M(n)$ processors, where $M(n)$ is the number of processors needed to multiply two n by n Boolean matrices in $O(\log n)$ time.

We note that the name "transitive reduction" was given to a problem similar to transitive compaction by Aho, Garey and Ullman ([AGU]). Given a digraph G, they ask for a digraph with a minimum number of arcs (not necessarily a subgraph of G) whose transitive closure is the same as that of G. When G is acyclic, the transitive compaction and transitive reduction of G are the same.

Definitions

Let G be a strongly connected digraph. A *forward (inverse) branching* rooted at x is a spanning tree of G in which x has in-degree (out-degree) zero and all other vertices have in-degree (out-degree) one. A *branching* is either a forward or an inverse branching. Throughout this paper the root, x, will be some (arbitrarily) fixed vertex of the input digraph, and the set of all branchings will be taken to be only those rooted at x.

An arc, e, is $\underline{G-redundant}$ (or simply $\underline{redundant}$ when the graph is clear) if $G-\{e\}$ is strongly connected. Arc e is $\underline{G-essential}$ (or $\underline{essential}$) if it is not redundant. Let H be a subgraph of G. Let $\underline{r_G(H)}$ denote the number of G-redundant arcs in H. When $H = G$ we will use the shorthand $\underline{r(G)}$.

An $\underline{H-philic}$ ($\underline{H-phobic}$) branching in G is one that has the greatest (smallest) number of arcs in common with H over all branchings (rooted at x) in G.

Our model of parallel computation is the Parallel Random Access Machine (PRAM), which consists of a collection of independent processing elements communicating through a shared memory. For a survey on the PRAM model and PRAM algorithms see [KR].

2. The Transitive Compaction Algorithm

Our basic algorithm is based solely on computing philic and phobic branchings. The following lemma explains how these branchings are computed:

Lemma 0: An H-philic (H-phobic) branching can be computed by a minimum-weight branching computation with zero-one weights.

Proof: Assign weight 0 (1) to every arc in H and weight 1 (0) to all other arcs. **[]**

Such a minimum-weight branching can be computed in time $O(\log^2 n)$ using $O(n^3)$ processors on a CRCW PRAM by Lovasz's method ([Lo]). On a CREW PRAM, this algorithm runs in $O(\log^3 n)$ time.

Proposition 1: An arc of G is essential if and only if it is the unique arc crossing some directed cut of G.

Proposition 2: The union of a forward branching and an inverse branching of G is a strongly connected spanning subgraph of G.

Proposition 3: Let G' be a strongly connected spanning subgraph of G. Then e is G'-redundant only if it is G-redundant.

Lemma 1: Let F be a forward branching in G and let I be an F-philic inverse branching in G. Let $G' = F \cup I$. Then the arcs of $I - F$ are all G'-essential.

Proof: Let $e \in I - F$. Assume $G' - \{e\}$ contains some inverse branching, I'. Then I' has one more arc in common with F than I does (since all branchings have the same number of arcs). But this contradicts the fact that I is F-philic. Thus $G' - \{e\}$ contains no inverse branching and is therefore not strongly connected. **[]**

A *cut leaving S* is the set of arcs extending from S to $V(G) - S$ in a digraph, G, and its cardinality is denoted by $\delta_G(S)$.

Theorem 1 (Edmonds' Branching Theorem ([Ed])):

Let

$$k = \min \{ \delta_G(S) \mid x \in S \, , \, S \neq V(G) \}.$$

Then G contains k arc-disjoint forward branchings (rooted at x).

Lemma 2: For every strongly connected digraph, G, there exists a forward branching, F, of G such that $r_G(F) \leq \frac{1}{2} r(G)$.

Proof: Let G' be obtained from G by duplicating all essential arcs. Let S be a proper subset of $V(G)$ containing x. We claim that $\delta_{G'}(S) \geq 2$. This is because the cut leaving S must contain at least one duplicated essential arc of G or at least two redundant arcs (by proposition 1). Therefore, by theorem 1, there are two arc-disjoint forward branchings in G' (each corresponding to a branching in G), one of which must contain at most half of the (unduplicated) G-redundant arcs.[]

Theorem 2: Let R be the set of redundant arcs in G. Let F be an R-phobic forward branching and let I be an F-philic inverse branching. Let $G' = F \cup I$. Then $r(G') \leq \frac{1}{2} r(G)$.

Proof: First note that by proposition 2, G' is strongly connected. By lemma 2 and proposition 3, $r_{G'}(F) \leq r_G(F) \leq \frac{1}{2} r(G)$. By lemma 1, $r(G') = r_{G'}(F)$. Therefore $r(G') \leq \frac{1}{2} r(G)$. **[]**

It is an immediate consequence of theorem 2 that the following *NC* algorithm gives a transitive compaction of G:

Repeat

(1) $R \leftarrow$ set of redundant arcs in G

(2) $F \leftarrow R$-phobic forward branching in G

(3) $I \leftarrow F$-philic inverse branching in G

(4) $G \leftarrow F \cup I$

until $R = \phi$

(5) output G (it is a transitive compaction of the input digraph)

By Theorem 2 the repeat loop runs $O(\log n)$ times, where n is the number of vertices in G. Steps (2) and (3) are implemented with Lovasz's minimum-weight branching algorithm (lemma 0). The straightforward implementation of step (1) is to perform a strong connectivity test (transitive closure) with each vertex of the graph deleted in turn, which requires $n \cdot M(n)$ processors. In the next section we shall show how to perform this step more efficiently.

3. Efficient Classification of Arcs

In this section we give parallel algorithms to classify the arcs of G as essential or redundant in poly-log time using only $O(n^3)$ processors. In section 3.1 we provide a simple polylog time parallel algorithm using $O(n^3)$ processors. In section 3.2 we provide a faster algorithm using tree contraction [MR].

3.1. Finding Redundant Arcs Using Minimum Weight Branchings

Let $\underline{E_f}$ ($\underline{E_i}$) be the set of essential arcs contained in all forward (inverse) branchings. It follows from proposition 2 that:

Proposition 4: An arc is essential if it is either in E_f or in E_i (or both).

Lemma 3: Let H be a set of arcs containing E_f and let F be an H-phobic forward branching in G. Then $|(F \cap H) - E_f| \leq \frac{1}{2} |H - E_f|$.

Proof: Let G' be obtained from G by duplicating all the arcs in E_f. As in lemma 2, there exist two arc-disjoint forward branchings in G' (corresponding to branchings in G), one of which contains at most half the arcs of $H - E_f$. **[]**

Therefore E_f (and similarly E_i) can be computed by the following algorithm:

(1) $H \leftarrow G$

repeat steps (2) and (3) $\lceil lg\ m \rceil$ times

(2) $F \leftarrow H$-phobic forward branching in G

(3) $H \leftarrow H \bigcap F$

(4) output H (this is the set E_f)

This algorithm requires $\log n$ applications of Lovasz's minimum weight branching algorithm, which runs in $O(\log^2 n)$ parallel time on a CRCW PRAM with $O(n^3)$ processors. Thus we can use this algorithm to find all redundant arcs in $O(\log^3 n)$ parallel time on a CRCW PRAM with $O(n^3)$ processors. This in turn leads to a transitive compaction algorithm that runs in $O(\log^4 n)$ parallel time on a CRCW PRAM with $O(n^3)$ processors.

3.2. Finding Redundant Arcs Using Tree Contraction

Let r be a fixed root of a directed graph $G = (V, E)$. We call arc (v, w) an *out − bridge* if (v, w) is on every path from r to w, and an *in-bridge* if (v, w) is on every path from v to r. Let O be the set of out-bridges of G, and I the set of in-bridges of G. Then the set of redundant arcs is the set $E - (I \bigcup O)$.

Let B be a forward branching rooted at r. Then every out-bridge of G lies in B. We can view B as a rooted directed tree $B = (V, E', r)$. For a vertex v in $V - \{r\}$, we denote by $parent(v)$, the parent of v in B. A vertex v is *active* if there is a path from r to v that avoids arc $(parent(v), v)$. Similarly, a non-tree arc (w, v) is *active* if it lies on a path from r to v that avoids arc $(parent(v), v)$.

Lemma 4: Let $B = (V, E', r)$ be a forward branching in a directed graph G. A tree arc $e = (parent(v), v)$ in B is an out-bridge of G if and only if v is not active.

Proof: If e is an out-bridge of G then every path from r to v passes through e. Thus v cannot be active. Conversely, if e is not an out-bridge, then there exists a path from r to v that avoids e and hence v must be active.[]

We now give an algorithm to identify all active vertices, and hence all out-bridges, using *tree contraction* [MR]. An analogous computation on an inverse branching rooted at r gives the in-bridges, from which we can compute the redundant arcs in G.

We shall use a variant of tree contraction proposed in [Ra] in which the basic operation is *shrink,* which we now define. A *leaf chain* in a rooted tree $T = (V, E, r)$ is a path $< v_1, \cdots, v_l >$ such that each $v_i, i > 1$ has exactly one incoming arc and one outgoing arc in T, v_1 has either no incoming arc or more than one outgoing arc in T, and v_l is a leaf in T. We will call v_1 the *root,* and v_l the *leaf* of the leaf chain. Note that every leaf in T is part of a leaf chain, possibly a degenerate one (if $l = 2$).

The shrink operation applied to a rooted tree $T = (V, E, r)$ removes all vertices in each leaf chain in T except the root of the leaf chain. It can be shown that $O(\log n)$ applications of the shrink operation suffice to reduce any n-node tree to a single node [Ra].

We now develop an algorithm *Shrink(P)* for identifying out-bridges for the case when the forward branching is a simple path. We shall then use this to find the out-bridges in leaf chains while implementing the shrink operation in a tree contraction algorithm to find out-bridges in G given an arbitrary forward branching.

The input to algorithm Shrink(P) is a directed graph $P = (V, E)$ consisting of a directed path $p = < 1, 2, \cdots, t >$, together with a collection of *forward arcs* of the form $(i, j), i < j$, and a collection of *back arcs* of the form $(i, j), i > j$. The algorithm Shrink(P) will identify all active vertices, thereby giving the out-bridges in p. Note that P is allowed to have two arcs of the form $(i, i+1)$, one of which is a forward arc and the other lies in p. We will need this when we apply algorithm Shrink(P) to the general problem of finding out-bridges in a graph with an arbitrary forward branching.

We now make a series of observations.

Observation 1: Every forward arc is active.

Let $p(u)$ be the subgraph of G induced by vertices u through t. For each vertex v in $p(u)$, let $v \rightarrow u$ if u is reachable from v in $p(u)$. Let $reach(u)$ be the set of vertices v in $p(u)$ with $v \rightarrow u$.

Observation 2: $Reach(u)$ is a single interval of the form $[u, u']$. Further a vertex $v \neq u$ is in $reach(u)$ if and only if there exists a sequence of back arcs $b_i = (u_i, v_i), i = 1, \cdots, k$ such that $v_1 = u$, $u_k \geq v$, and $u_i \geq v_{i+1}, i = 1, \cdots, k-1$.

Lemma 5: A vertex u is active if and only if there is a forward arc (k, l) with $k < u$ and l in $reach(u)$.

Proof: Let u be an active vertex. Then there is a path q from the root to u that avoids arc $(u-1, u)$. This in turn implies that q must contain a forward arc $f = (k, l)$ with $k < u$, $l \geq u$ and with u reachable from l using only arcs in $p(u)$. Hence l must be in $reach(u)$.

Conversely suppose there is a forward arc $f = (k, l)$ with $k < u$ and l in $reach(u)$. Hence there is a path q from l to u using only arcs in $p(u)$. Then the path consisting of arcs in p from the root to k, followed by arc f and then the path q is a path from 1 to u that avoids arc $(u-1, u)$. Hence u must be an active vertex.[]

Observations 1 and 2 and Lemma 5 together give us the following algorithm to find all out-bridges when the forward branching is a simple path.

Shrink(P);

1. Find $reach(u)$ for each vertex u as follows:

   a) For each back arc $b = (i, j)$ find a back arc $next(b) = (i' j')$ with j' in $[j, i]$ and maximum i'. If $i' \le i$ then set $next(b) = \phi$.

   b) Form an auxiliary graph with a vertex for each back arc b and an arc from b to $next(b)$, if $next(b)$ exists. This auxiliary graph is a forest of trees.

   c) For each vertex u, pick some back arc $b = (v, u)$ incident on u, and find the root b' of the tree it belongs to. Let b' be the back arc (x, y). Set $reach(u) = [u, x]$.

   If there is no back arc incident on u set $reach(u) = [u, u]$.

2. For each vertex u, find a forward arc $f = (k, l)$ with l in $reach(u)$ and with minimum k. If $k < u$ mark u as active.

3. For each vertex u that is not active, mark $(p(u), u)$ as an out-bridge.

We now show how to implement each of the steps in the algorithm efficiently in parallel. Step 3 can be implemented trivially in constant time with t processors. The following method implements step 2 in $O(\log t)$ time with a number of processors linear in the size of P: Initially we determine, for each vertex u, the forward arc (v, u) with minimum v (if such an arc exists). It is straightforward to compute this in $O(\log t)$ time with a linear number of processors. Then by a doubling computation we compute, for each interval $[u, u + 2^j], 1 \le j \le \lceil \log t \rceil, 1 \le u \le t - 2^j$, the forward arc (v, x) with minimum v such that x is in the interval $[u, u + 2^j]$. This computation can be done in $O(\log t)$ time with a linear number of processors on a CREW PRAM. Any interval $[i, j], 1 \le i < j \le t$ can be written as the overlapping union of two of the previously computed intervals, and hence each vertex can now find a forward arc as required in step 2 in constant time.

Step 1 can be implemented in $O(\log t)$ time with a linear number of processors on a CREW PRAM as follows. Step 1a can be performed in a manner analogous to step 2. Step 1b can be implemented in constant time with a linear number of processors. Step 1c can be implemented by pointer jumping in $O(\log t)$ time with a linear number of processors. Thus we have a parallel algorithm for Shrink(P) that runs in $O(\log t)$ time with a linear number of processors on a CREW PRAM.

We now incorporate the Shrink algorithm in the following tree contraction algorithm that finds the out-bridges in an arbitrary forward branching of a directed graph G rooted at r. The algorithm constructs a sequence of pairs (G_k, T_k), where G_k is a digraph and T_k is a forward branching; G_1 is the input digraph and T_1 is a forward branching of G rooted at a fixed vertex r. Iteration k identifies the leaf chains of T_k, determines the out-bridges of G_k within those leaf chains, deletes all the vertices of the leaf chains except their roots, and then performs a transitive closure computation and adds appropriate arcs to ensure that the out-bridges in G_{k+1} are precisely the out-bridges of G not yet identified.

Outbridges(G=(V,E,r),T);

Input: A directed graph $G = (V, E)$ with a forward branching T rooted at r; $|V| = n$.

Repeat

1. *Find out-bridges in the leaf chains of T:*

For each leaf chain l in T pardo

Let t be the root of l and t' the leaf of l. Let L' be the subgraph of G induced by vertices in l.

a) Form L from L' by introducing a forward arc (t, y) for each non-tree arc (x, y) in G with y in $V(l) - \{t\}$ and x not in $V(l)$.

b) Apply *Shrink(L)* to find the out-bridges in L and label these as out-bridges of G.

2. *Remove leaf chains from T:*

a) Form the graph H with vertex set V and arc set the arcs in all leaf chains and all non-tree arcs of G.

b) Form M, the adjacency matrix of H, and form the transitive closure M^* of M.

c) For each vertex v, determine, using M^*, the set of vertices from which v is reachable in H. For each such vertex w, introduce an arc (w, v) in G.

d) For each vertex t that is the head of some leaf chain, delete all incoming non-tree arcs to proper descendants of t. Collapse all of these proper descendants into t. Delete any self-loops in this graph.

until $T = \phi$

Generalizing our earlier notation for the case when the forward branching is a simple path, we now let $p(u)$ be the subgraph of G induced by those vertices that lie in the subtree of T rooted at u. For each vertex v in $p(u)$, let $v \rightarrow u$ if u is reachable from v in $p(u)$. Let $reach(u)$ be the set of vertices v in $p(u)$ with $v \rightarrow u$.

The following lemma is a straightforward generalization of Observation 2 and Lemma 5 (here a vertex v is a *descendant* of a vertex u if $u = v$ or if there is a directed path from u to v in T; otherwise v is a *non-descendant* of u).

Lemma 6: A vertex u in G is active if and only if there is an arc (x, y) with x a non-descendant of u and with y in $reach(u)$.

Let G be a directed graph with a forward branching T rooted at r, and let v be a vertex in G. An *active path* to v is a path p from r to v consisting of an initial path p' using tree arcs from r to a non-descendant x of v followed by an intermediate path consisting of a single non-tree arc a from x to a descendant y of v followed by a final path p'' from y to v using only arcs connecting descendants of v.

Observation 3: Vertex v is active if and only if there is an active path to v.

We now prove some lemmas that will allow us to establish the correctness of algorithm Outbridges. As before let G_i and T_i be the graph and forward branching present at the start of the ith iteration of the repeat loop in the algorithm; hence G_1 and T_1 are the input graph together with its forward branching, and G_k and T_k are the current graph and forward branching at the start of the kth iteration. Similarly let H_i be the graph H of step 2a of algorithm Outbridges constructed in the ith iteration of the repeat loop.

We first note that Observation 2 remains valid in each G_k when u is a vertex in a leaf chain of T_k. We state this in the following observation.

Observation 4: Let u be a vertex in a leaf chain l of forward branching T, where for convenience we assume that the vertices in the leaf chain are numbered from 1 to s, with 1 the root of the leaf chain and s the leaf of the leaf chain. Then $reach(u)$ is a single interval of the form $[u, u']$. Further, a vertex $v \neq u$ is in $reach(u)$ if and only if there exists a sequence of back arcs $b_i = (u_i, v_i), i = 1, \cdots, k$ in L (where L is the subgraph of G induced by vertices in l) such that $v_1 = u$, $u_k \geq v$, and $u_i \geq v_{i+1}, i = 1, \cdots, k-1$.

Lemma 7: For each $k \geq 1$, algorithm Outbridges correctly finds the out-bridges in the leaf chains of G_k.

Proof: By Observation 4, for a vertex u in a leaf chain l of T_k, $reach(u)$ in G_k is the same as $reach(u)$ in the subgraph of G_k induced by l. Hence the reach value of each vertex in the leaf chain is correctly computed in the Shrink computation of step 1b in algorithm Outbridges.

By Lemma 6, a vertex u in a leaf chain is active if and only if there is an arc $e = (x, y)$ in G_k with x a non-descendant of u and with y in $reach(u)$. Such an arc e is either a forward arc in the leaf chain or is an arc with x not in the leaf chain and y in the leaf chain. The former case is the same as that used in the Shrink algorithm. In the latter case, (x, y) will cause any vertex u in the leaf chain with y in $reach(u)$ to be active. Hence for the purpose of the Shrink algorithm this is equivalent to having an arc from the root, t, of the leaf chain to y. Thus the computation in steps 1a and 1b of algorithm Outbridges correctly finds the outbridges in the leaf chains of G_k.[]

Lemma 8 Let $e = (u, v)$ be an out-bridge in G_k, $k > 1$. Then e is an out-bridge in G_{k-1}.

Proof: First note that if e is an out-bridge in G_k, then e lies in T_k. Hence e lies in T_{k-1}, since every tree arc in T_k is present as a tree arc in T_{k-1}.

Suppose e is not an out-bridge in G_{k-1}. Hence v is an active vertex in G_{k-1}. Let p be an active path to v in G_{k-1}, and let p consist of an initial tree path p' to a vertex x that is a non-descendant of v, followed by a non-tree arc $a = (x, y)$, where y is a descendant of v, followed by a final path p'' from y to v using only arcs connecting vertices that are descendants of v. We now establish that there must be an active path to v in G_k, contradicting the assumption that e is an out-bridge of G_k, and thereby establishing the lemma.

If p contains no vertex in $G_{k-1} - G_k$ then p is an active path to v in G_k as well. If p contains some vertices in $G_{k-1} - G_k$ then consider the last vertex z on p such that z is in $G_{k-1} - G_k$.

Case 1: z is a non-descendant of v. Then z must be x and all vertices in p'' lie in G_k. Let t be the root of the leaf chain of G_{k-1} to which z belongs. Then by step 2d of algorithm Outbridges, z is collapsed into t and hence the path in G_k consisting of the tree path to t, followed by non-tree arc (t, y), followed by path p'' is an active path to v in G_k.

Case 2: z is a descendant of v. Let $b = (z, a)$ be the outgoing arc from z in p, and let t' be the root of the leaf chain in G_{k-1} to which z belongs. Hence t' is a descendant of a and z is a proper descendant of t'. Let p''' be the portion of p'' from a to v. The path p''' is a path in G_k as well.

Case 2a: The vertex z is reachable from some non-descendant w of v in H_k. Then an arc (w, z) is introduced in step 2c of the algorithm. If w is in G_k then the path from r to w followed by arc (w, a) followed by path p''' is an active path to v in G_k. If w is in $G_{k-1} - G_k$ then the analysis of Case 1 gives an active path to v in G_k.

Case 2b: The vertex z is not reachable from any non-descendant of v in H_k. Now consider p''. This is a path of the form $< u_{1,1}, \cdots, u_{1,k_1}, v_{1,1}, \cdots, v_{1,l_1}, \cdots, u_{c,1}, \cdots, u_{c,k_c}, v_{c,1}, \cdots, v_{c,l_c} >$, where the $u_{i,j}$ are in $G_{k-1} - G_k$ and the $v_{i,j}$ are in G_k, and if y is in G_k the initial sequence of $u_{1,j}$'s is empty. All of the $u_{i,j}$ and $v_{i,j}$ are descendants of v. Each $v_{i,1}$ is reachable from $v_{i-1,l_{i-1}}, i > 1$ in H_k. Hence by step 2c of algorithm Outbridges, there is an arc from $v_{i-1,l_{i-1}}$ to $v_{i,1}, i > 1$ in G_k. The vertex $v_{1,1}$ has an incoming arc from x in G_k. The remaining arcs in p'' remain in G_k. Hence there is a path from x to v in G_k that contains only vertices that are descendants of v in G_k. Hence v is an active vertex in G_k.[]

Lemma 9: Let $e = (u, v)$ be a tree arc in G_k, $k > 1$ that is not an out-bridge in G_k. Then e is not an out-bridge in G_{k-1}.

Proof: Since e is not an out-bridge in G_k there is an active path p to v in G_k. Consider any arc $f = (x, y)$ in p that is not present in G_{k-1}. If f was introduced in step 2c of algorithm Outbridges then there is a path from x to y in G_{k-1} that avoids all tree arcs in G_k and hence arc e. If f was introduced in step 2d then there is a path from a descendant of x to y in G_{k-1} that avoids all tree arcs in G_k. Hence there is a path from x to y in G_{k-1} that avoids arc e. Hence from p we can obtain an active path p' to v in G_{k-1}. Thus e is not an out-bridge in G_{k-1}.[]

Lemma 10: Algorithm Outbridges correctly finds the out-bridges of G.

Proof: We show that at the start of each iteration of the repeat loop,

1) The out-bridges identified so far are exactly the out-bridges in the portion of the input graph G that has been collapsed by the algorithm.

2) An arc e in the current graph G is an out-bridge in this graph if and only if it is an out-bridge in the original input graph.

The proof is by induction on k, the number of iterations of the repeat loop.

Base: $k = 1$. The claim is vacuously true since no out-bridges have been identified and the input graph is the same as the current graph.

Induction step: Assume that the two claims are true until the start of iteration $k - 1$ and now consider the start of iteration k. Claim 1) follows by the induction hypothesis and Lemma 7. Claim 2) follows by the induction hypothesis and Lemmas 8 and 9.[]

Finally we note that algorithm Outbridges runs in $O(\log^2 n)$ with $O(n^3)$ processors on a CRCW PRAM. To see the processor and time bounds let us analyze the time complexity of each iteration of the repeat loop. By the previous analysis for the time complexity of algorithm Shrink, step 1 runs in $O(\log n)$ time with $O(n^2)$ processors on a CREW PRAM. Steps 2a, 2c, and 2d run in $O(\log n)$ time with $O(n^2)$ processors on a CREW PRAM. Step 2b runs in $O(\log n)$ time with $O(n^3)$ processors on a CRCW PRAM, and is the most expensive step in the repeat loop. Since the repeat loop is executed $O(\log n)$ times we obtain the stated time and processor bounds for algorithm Outbridges. On a CREW PRAM this algorithm runs in $O(\log^3 n)$ time with $M(n)$ processors.

Whether we use a CREW model or a CRCW model the time and processor bounds for finding a minimum weight branching using the algorithm in [Lo] dominate the time and processor bounds of algorithm Outbridges. Hence we can find redundant arcs within the time and processor bounds for minimum weight branchings, and thus the parallel transitive compaction algorithm runs in $O(\log^3 n)$ parallel time with $O(n^3)$ processors on a CRCW PRAM and in $O(\log^4 n)$ parallel time with the same processor bound on a CREW PRAM.

4. Sequential Algorithms for Transitive Compaction

As in section 3.2, let r be a fixed root of a directed graph $G = (V, E)$, where $|V| = n$ and $|E| = m$. An algorithm for finding the in- and out-bridges is given in [Ta2]. This algorithm actually does more: It computes two forward branchings T_1 and T_2 having only the out-bridges in common, and two inverse branchings, T_3 and T_4, having only the in-bridges in common. This algorithm can be implemented to run in linear time by using linear-time algorithms for computing nearest common ancestors [HT] and maintaining disjoint sets [GT].

Let R be the set of redundant arcs, I the set of in-bridges and O the set of out-bridges. Hence $R = E - (I \cup O)$. The following algorithm finds a transitive compaction of G.

1. Pick a root vertex r in G. Find a forward branching B and an inverse branching B' in G and replace G by $B \cup B'$.

2. *Repeat*

   a) Construct two forward branchings T_1 and T_2 having only the out-bridges in common; identify the set of out-bridges as O.

   b) Construct two inverse branchings T_3 and T_4 having only the in-bridges in common; identify the set of in-bridges as I.

   c) Form the set of redundant arcs R as $R = E - (O \cup I)$.

   d) For $i = 1, 2, 3, 4$ form $S_i = T_i \cap R$.

   e) Choose T_i and T_j such that $1 \le i \le 2$, $3 \le j \le 4$ and $S_i \cup S_j$ has minimum cardinality among $S_1 \cup S_3, S_2 \cup S_3, S_1 \cup S_4, S_2 \cup S_4$.

   f) Replace G by $T_i \cup T_j$.

   until $R = \phi$.

The following claim establishes that the repeat loop is executed only $O(\log n)$ times.

Lemma 11: In step 2e of the algorithm the chosen S_i and S_j satisfy $|S_i \cup S_j| \le (3/4) \cdot |R|$.

Proof: For $i = 1, 2, 3, 4$, let F_i be the set of those arcs in T_i that are not present in any other T_j, and let $P_{1,3} = (S_1 \cap S_3) \cup F_1$, $P_{2,4} = (S_2 \cap S_4) \cup F_2$, $P_{2,3} = (S_2 \cap S_3) \cup F_3$ and $P_{1,4} = (S_1 \cap S_4) \cup F_4$. Note that $P_{1,3}, P_{2,4}, P_{2,3}$ and $P_{1,4}$ are disjoint. Let one, say $P_{1,3}$ be the one of maximum cardinality. Then we must have $|P_{2,4}| + |P_{2,3}| + |P_{1,4}| \le (3/4) \cdot |R|$. But $S_2 \cup S_4 \subseteq P_{2,4} \cup P_{2,3} \cup P_{1,4}$, which implies $|S_2 \cup S_4| \le (3/4) \cdot |R|$. Since we also have $S_1 \cup S_3 \subseteq P_{1,3} \cup P_{1,4} \cup P_{2,3}$, $S_1 \cup S_4 \subseteq P_{1,3} \cup P_{1,4} \cup P_{2,4}$ and $S_2 \cup S_3 \subseteq P_{2,3} \cup P_{2,4} \cup P_{1,3}$, we have $|S_1 \cup S_3| \le (3/4) \cdot |R|$ if $P_{2,4}$ is of maximum cardinality, $|S_1 \cup S_4| \le (3/4) \cdot |R|$ if $P_{2,3}$ is of maximum cardinality and $|S_2 \cup S_3| \le (3/4) \cdot |R|$ if $P_{1,4}$ is of maximum cardinality. Hence the chosen S_i and S_j in step 2e of the algorithm satisfy $|S_i \cup S_j| \le (3/4) \cdot |R|$.[]

Step 1 of the algorithm takes $O(n+m)$ time and renders G sparse ($O(n)$ arcs). As mentioned above, steps 2a and 2b can be implemented to run in $O(n)$ time using the algorithm in [Ta2], in conjunction with the algorithms in [HT] and [GT]. Each of steps 2c through f takes $O(n)$ time. Hence each execution of the repeat loop takes linear time. Since by Lemma 11 the repeat loop is executed $O(\log n)$ times, the entire transitive compaction algorithm runs in $O(m + n \log n)$ time.

The algorithm of section 2 can also be implemented to run in $O(m + n \log n)$ time. This is because the minimum-weight branching algorithm of Edmonds [Ed2] can be implemented to run in linear time for 0-1 edge weights by using the algorithm in [GGST], with the heaps replaced by two buckets. As before, the redundant arcs can be found in linear time and hence each execution of the repeat loop takes linear time, leading to an $O(m + n \log n)$ time sequential algorithm for transitive compaction.

We have obtained sequential and parallel algorithms with similar complexities for analogous problems on undirected graphs, i.e., for finding a minimal bridge-connected spanning subgraph and a minimal biconnected spanning subgraph in an undirected graph, if such subgraphs exist. These results will appear in a companion paper.

We conclude by noting that it is conceivable that one (or both) of our sequential algorithms runs in linear time, since it is possible that the repeat loop needs to be executed only a constant number of times. We leave this question for further investigation. For the same reason it is possible that our parallel algorithms run faster than the stated time bounds by an $O(\log n)$ factor.

References

[ABI] Alon, N., Babai, L. and Itai, A. , "A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem", *J. of Algorithms,* 7, pp. 567-583, 1986.

[AGU] Aho, A.V., Garey, M.R. and Ullman, J.D. , "The Transitive Reduction of a Directed Graph", *SIAM J. Comput.,* 1, pp. 131-137, 1972.

[CW] Coppersmith, D. and Winograd, S., "Matrix multiplication via arithmetic progressions," *Proc. 19th Ann. ACM Symp. on Theory of Computing,* pp. 1-6, 1987.

[Ed] Edmonds, J. , "Edge-Disjoint Branchings", in *Combinatorial Algorithms,* Algorithmic Press, pp. 91-96, 1973.

[Ed2] Edmonds, J., "Optimum branchings," *J. of Res. of the Nat. Bureau of Standards,* 71B, 1967, pp. 233-240.

[ET] Eswaran, K.P. and Tarjan, R.E. , "Augmentation Problems", *SIAM J. Comput.,* 5, pp. 653-665, 1976.

[GGST] Gabow, H.N., Galil, Z., Spencer T. and Tarjan, R.E., "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica,* 6, pp. 106-122, 1986.

[GJ] Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, San Fransisco, CA, 1979.

[GT] Gabow, H.N. and Tarjan, R.E., "A linear-time algorithm for a special case of disjoint set union," *Journal of Comput. Sys. Sci.,* 30, pp. 209-221, 1985.

[GS] Goldberg, M. and Spencer, T., "A New Parallel Algorithm for the Maximal Independent Set Problem", *Proc. 28th IEEE Symp. on Foundations of Computer Science,* pp. 161-165, 1987.

[HT] Harel, D. and Tarjan, R.E., "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.,* 13, pp. 338-355, 1984.

[KR] Karp, R.M. and Ramachandran, V. "Parallel algorithms for shared memory machines," *Handbook of Theoretical Computer Science,* J. van Leeuwen, ed., North Holland, to appear; also Report No. UCB/CSD 88/408, Computer Science Div., Univ. of California, Berkeley, CA, 1988.

[KUW] Karp, R.M. , Upfal, E. and Wigderson, A. , "Are Search and Decision Problems Computationally Equivalent?", *Proc. 17th ACM Symp. on Theory of Computing,* pp. 464-475, 1985.

[KW] Karp, R.M. and Wigderson, A. , "A Fast Parallel Algorithm for the Maximal Independent Set Problem", *JACM,* pp. 762-773, 1985.

[Lo] Lovasz, L. "Computing Ears and Branchings in Parallel", *Proc. 26th IEEE Symp. on Foundations of Computer Science,* pp. 464-467, 1985.

[Lu] Luby, M. "A Simple Parallel Algorithm for the Maximal Independent Set Problem", *Proc. 17th ACM Symp. on Theory of Computing,* pp. 1-10, 1985.

[MR] Miller, G.L. and Reif, J.H., "Parallel tree contraction and its applications," *Proc. 26th Ann. Symp. on Foundations of Comp. Sci.,* pp. 478-489, 1985.

[Ra] Ramachandran, V. "Fast parallel algorithms for reducible flow graphs," *Concurrent Computations: Algorithms, Architecture and Technology,* S.K.Tewksbury, B.W.Dickinson and S.C.Schwartz, ed., Plenum Press, New York, NY, 1988, in press.

[So] Soroker, D. , "Fast Parallel Strong Orientation of Mixed Graphs and Related Augmentation Problems", *J. of Algorithms,* 9, pp. 205-223, 1988.

[Ta] Tarjan, R.E. , "Finding Optimum Branchings", *Networks,* 7, pp. 25-35, 1977.

[Ta2] Tarjan, R.E., "Edge-disjoint spanning trees and depth-first search," *Acta Informatica,* 7, pp. 171-185, 1976.