Simpler Linear-Time Modular Decomposition via Recursive Factorizing Permutations

Marc Tedder[1], Derek Corneil[1]*, Michel Habib[2], and Christophe Paul[3]**

[1] Department of Computer Science, University of Toronto
{mtedder,dgc}@cs.toronto.edu
[2] LIAFA and the University of Paris 7 - Denis Diderot
habib@liafa.jussieu.fr
[3] CNRS - LIRMM, Univ. Montpellier II France (part of this research was conducted while on sabbatical in the School of Computer Science at the University of McGill)
christophe.paul@lirmm.fr

Abstract. Modular decomposition is fundamental for many important problems in algorithmic graph theory including transitive orientation, the recognition of several classes of graphs, and certain combinatorial optimization problems. Accordingly, there has been a drive towards a practical, linear-time algorithm for the problem. This paper posits such an algorithm; we present a linear-time modular decomposition algorithm that proceeds in four straightforward steps. This is achieved by introducing the notion of factorizing permutations to an earlier recursive approach. The only data structure used is an ordered list of trees, and each of the four steps amounts to simple traversals of these trees. Previous algorithms were either exceedingly complicated or resorted to impractical data-structures.

1 Introduction

A natural operation to perform on a graph G is to take one of its vertices, say v, and replace it with another graph G', making v's neighbours universal to the vertices of G'. Modular decomposition is interested in the inverse operation: finding a set of vertices sharing the same neighbours outside the set – that is, finding a *module* – and contracting this module into a single vertex. A graph's modules form a partitive family [2], and as such, define a decomposition scheme for the graph with an associated decomposition tree composed of the graph's *strong modules* – those that don't overlap other modules. To compute this *modular decomposition tree* is to compute the *modular decomposition* (and vice versa), and with its succinct representation of a graph's structure, its computation is often a first-step in many algorithms. Indeed, since Gallai first noticed its importance to comparability graphs [12], modular decomposition has been established as a

* Marc Tedder and Derek Corneil wish to thank NSERC for partially funding this research.

** Michel Habib and Christophe Paul were supported by the French ANR project ANR-O6-BLAN-0148-01, *Graph Decomposition and Algorithms* (GRAAL).

fundamental tool in algorithmic graph theory. All efficient transitive orientation algorithms make essential use of modular decomposition (e.g., [17]). It is frequently employed in recognizing different families of graphs, including interval graphs [18], permutation graphs [23], and cographs [3]. Furthermore, restricted versions of many combinatorial optimization problems can be efficiently solved using modular decomposition (e.g., [8]). While the papers [18–20] provide older surveys of its numerous applications, new uses continue to be found, such as in the areas of graph drawing [22] and bioinformatics [11].

Not surprisingly, the problem of computing the modular decomposition has received considerable attention. Much like planarity testing and interval graph recognition, the importance of the problem has bent efforts toward a simple and efficient solution. The first polynomial-time algorithm appeared in the early 1970's and ran in time $O(n^4)$ [5]. Incremental improvements were made over the years – [14, 21], for example – culminating in 1994 with the first linear-time algorithms, developed independently by McConnell and Spinrad [16], and Cournier and Habib [4]. These are unfortunately so complex as to be viewed primarily as theoretical contributions. Subsequent algorithms have not been much better.

The attempts made in [17] and [7] are illustrative. Both adopt an approach pioneered by Ehrenfeucht et. al. [9], later improved upon by Dahlhaus [6]. The idea is to pick an arbitrary vertex, say x, and recursively compute the modular decomposition tree for its neighbourhood, $N(x)$, and its non-neighbourhood, which we denote $\overline{N(x)}$. Any strong module not containing x must be a module of either $G[N(x)]$ or $G[\overline{N(x)}]$, and therefore can be extracted from their recursively computed modular decomposition trees. Once extracted, these can then be used to compute the strong modules containing x. The two types of modules are then assembled to form the tree. While conceptually simple, identifying the strong modules containing x and then constructing the tree has proven difficult to perform in linear-time. In [17] they settle for an $O(n + m \log n)$ implementation, while [7] must use conceptually difficult tricks, a careful charging argument, and the challenging Gabow-Tarjan [10] version of union-find.

Factorizing permutations were introduced by Capelle and Habib [1] as a means of avoiding the difficulty just described. A *factorizing permutation* is a permutation of a graph's vertices in which the strong modules appear consecutively. If such a permutation can be computed, then the algorithm of [1] can be used to derive the tree in linear-time. This indirect approach was used in [15] to get an $O(n + m \log n)$ algorithm, and while linear-time was claimed in [13], the paper contains an error which kills the algorithm's simplicity.

In this paper we introduce the notion of factorizing permutations to the recursive framework described above to produce a straightforward linear-time modular decomposition algorithm. The two approaches turn out to be complementary. From the recursively computed trees a factorizing permutation is easily constructed using a refinement technique generalizing traditional partition refinement from sets to trees. We then show how this factorizing permutation makes it easy to identify the strong modules containing x and assemble the tree. We manage to maintain the conceptual simplicity of both, facilitating the proof of the algorithm's correctness and running time. Moreover, our algorithm avoids sophisticated data structures, using only an ordered list of trees.

1.1 Preliminaries

All graphs in this paper are simple and undirected. Connected components will simply be referred to as *components*, while the connected components of the complement will be referred to as *co-components*. We will talk often of an *ordered list of trees*, which will sometimes be referred to as an *ordered forest*. The leaves within this forest will always correspond to the vertices of the graph in question. When we refer to an ordering of these vertices it is with respect to one implicitly defined by the ordered forest; in particular, any pre-ordering of the leaves of each tree, processed from left to right. Thus, the leaves descendent from any one node will always appear consecutively. Note that sometimes a set of vertices will be referred to as a "tree". We do this to streamline the exposition; our intent will become clear.

A *module* is a set of vertices all of whom share the same neighbourhood outside the set. The modular decomposition tree will occasionally be referred to as the *MD tree*. The MD tree can be recursively defined as follows: the root of the tree corresponds to the entire graph; if the graph is disconnected, the root is called *parallel* and its children are the MD trees of its components; if the graph's complement is disconnected, the root is called *series* and its children are the MD trees of the co-components; in all other cases the root is called *prime*[4], and its children are the MD trees of the graph's maximal modules. Recall that the nodes in this tree are the graph's *strong modules*, which are those that don't *overlap* others.

2 Overview of the Algorithm

2.1 Recursion

The algorithm begins by selecting an arbitrary vertex, x, called the *pivot*, and placing its neighbourhood to its left and its non-neighbourhood to its right, giving us the ordered list of trees, $N(x), x, \overline{N(x)}$. Next, the modular decomposition tree for $G[N(x)]$ is recursively computed. As this occurs, with new pivots being selected, the neighbours of these pivots in $\overline{N(x)}$ are "pulled forward" so that afterwards we have the ordered list of trees, $T(N(x)), x, \overline{N_A(x)}, \overline{N_N(x)}$, where $T(N(x))$ is the modular decomposition tree for $G[N(x)]$, and $\overline{N_A(x)}$ is the subset of $\overline{N(x)}$ with at least one neighbour in $N(x)$, and $\overline{N_N(x)}$ is the subset of $\overline{N(x)}$ without neighbours in $N(x)$. The algorithm then recursively computes the modular decomposition tree for $\overline{N_A(x)}$, pulling its neighbours in $\overline{N_N(x)}$ forward in a similar fashion. And so on. Eventually we arrive at the following ordered list of trees:

[4] This definition of prime differs somewhat from that which normally appears in the literature.

$$\underbrace{T(N_0)}_{N(x)}, x, \underbrace{T(N_1), \ldots, T(N_k)}_{\overline{N(x)}}, \tag{1}$$

where the N_i's correspond to the distance layers in a breadth-first-search begun from x, and the $T(N_i)$'s are their modular decomposition trees. We will sometimes refer to the N_i's as *layers*.

The rest of this paper assumes that the graph is connected and thus each vertex in N_i has an edge to N_{i-1} (or x in the case of N_0). When the graph is disconnected, the layers up to N_{k-1} along with x form one of its connected components. In this case the algorithm builds the MD tree for this component as described below, then unifies the result with $T(N_k)$ under a common root labeled parallel. This adds a constant amount of work to each stage. Each stage is defined by a pivot, and vertices are only pivots once, so this work is consistent with linear-time.

2.2 Refinement

We wish to transform the above ordered list of trees into a factorizing permutation that will simplify the construction of the modular decomposition tree. We begin doing so by refining the trees using the edges active at this stage:

Definition 1. *An edge becomes* active *when one of its endpoints is pivot or if its endpoints reside in different layers.*

The refinement procedure, which generalizes traditional partition refinement from sets to trees, is detailed in section 3.1. In it we process each vertex in turn and use its incident active edges to refine the trees other than its own. What results is not a factorizing permutation but something very close.

To see why, first consider a strong module not containing x, say M. Notice that for some N_i, we have $M \subseteq N_i$, with M also a module in $G[N_i]$. A theorem of [19] says that either M is a strong module in $G[N_i]$, and thus an internal node in $T(N_i)$, or it corresponds to the union of siblings in $T(N_i)$. In the former case, refinement leaves the node corresponding to M unaffected. In the latter case, refinement groups the siblings under a new internal node inserted into $T(N_i)$ in their former location, in a sense "splitting" their former parent in two. Thus:

Lemma 1 (Proved in section 3.1). *The strong modules not containing x appear consecutively after refinement.*

We are not so fortunate for strong modules containing x, although refinement does get them close to appearing consecutively. As described above, refinement groups siblings under new internal nodes. When these new nodes are at depth-1, however, refinement deletes their parent, making that new node a root of its own tree in our ordered list, effectively splitting the siblings' old tree in two. The intuition here comes from the fact that the (co)-components of the layers correspond either to the nodes at depth-0 or depth-1 in the $T(N_i)$'s, combined with the special role played by these (co)-components:

Proposition 1. *If C is a co-component of $G[N_0]$ and M' is a strong module containing x, then either $C \subset M'$ or $C \cap M' = \emptyset$. Similarly for C a component of $G[N_i], i > 0$.*

During refinement the module will be contained within an interval of trees:

Lemma 2 (Proved in section 3.1). *Let $T_k, \ldots, T_1, x, T'_1, \ldots, T'_\ell$ be the ordered forest at some point during refinement, and let M' be a strong module containing x. Then there are trees T_i and T'_j (called the* bounding trees *for M') such that,*

(i) $M' \supset T_{i-1} \cup \cdots \cup T_1 \cup \{x\} \cup T'_1 \cup \cdots \cup T'_{j-1}$, *and*
(ii) $M' \subseteq T_i \cup \cdots \cup T_1 \cup \{x\} \cup T'_1 \cup \cdots \cup T'_j$.

The interval bounding the module becomes more and more precise as trees are split. The next stage in the algorithm makes the interval exact.

2.3 Promotion

When siblings are grouped under a new node during refinement it happens because a vertex in a different tree is adjacent to them but not their other siblings. The siblings' former parent cannot therefore correspond to a module; this is also true of all their ancestors. Refinement accounts for this by marking these nodes for deletion. We show in section 3.1 that when refinement has finished, the nodes without marked children will correspond to the strong modules not containing x. Promotion is the process of deleting all other nodes, – internal nodes are "promoted" upward as their ancestors are deleted – leaving only the strong modules not containing x.

The real benefit of promotion however is that it gives us the desired factorizing permutation. The strong modules not containing x end up consecutive as explained above. But now the strong modules containing x will also be consecutive: as nodes are deleted, the portion of the bounding trees that is in the module will be placed next to the other trees in the module (see lemma 2).

Lemma 3 (Proved in section 3.2). *The ordered forest that results from promotion provides a factorizing permutation.*

2.4 Assembly

In fact, promotion gives us much more than a factorizing permutation: we have an ordered list of trees whose nodes (excepting x) correspond to the strong modules not containing x; moreover, each of these strong modules is itself properly decomposed (their parts were originally in their respective $T(N_i)$'s, and neither refinement nor promotion changes this). What remains, then, is to identify the strong modules containing x, determine the trees in our list constituting them, then use this information to assemble the modular decomposition tree. This was the bottleneck encountered by the previous recursive algorithms. Our factorizing permutation makes it easy.

With a factorizing permutation we know the strong modules containing x are nested: $[\cdots[\cdots[\cdots x\cdots]\cdots]\cdots]$. Since our ordered forest consists of the strong modules not containing x, no tree in it overlaps these brackets. So to build the MD tree, it suffices to insert the brackets between the trees in our list: once this is done, a node is made for each pair of brackets and a "spine" for the MD tree is built; to this we merely affix the trees in our list according to the placement of the brackets. We show how to insert the brackets in section 3.3.

3 Details and Correctness

3.1 Refinement

The refinement process described in the overview is given by algorithm 1; note that it requires algorithm 2 which appears afterwards.

Algorithm 1: Refinement of the ordered list of trees in (1) by the active edges

```
foreach vertex $v$ do
    Let $\alpha(v)$ be its incident active edges;
    Refine the list of trees using $\alpha(v)$ according to algorithm 2, such that:
    if $v$ is to $x$'s left or $v$ is to $x$'s right and refines a tree to $x$'s left then
        refine using left splits according to algorithm 2, and when a node is
        marked, mark it with "left";
    else
        refine using right splits according to algorithm 2, and when a node is
        marked, mark it with "right";
    end
end
```

Below we sketch the proof of lemmas 1 and 2. For the former we actually prove something slightly stronger from which lemma 1 follows immediately:

Lemma 4. *The nodes in the ordered list of trees resulting from refinement that do not have marked children correspond exactly to the strong modules containing x.*

Proof. **[Sketch]** Let M be a strong module not containing x. As stated in the overview, M must be entirely contained in some N_i, and it must be a module of $G[N_i]$. A theorem of [19] guarantees that M is either a node in $T(N_i)$ or the union of children, say $c_1, \ldots, c_k$, of a series or parallel node in $T(N_i)$. Appealing to algorithm 1, we see that in the former case it remains a node throughout refinement and none of its children are ever marked, since each vertex outside $T(N_i)$ is either universal to, or isolated from, the node. Algorithm 1 also makes clear that in the latter case the children will remain siblings throughout refinement, and will not be marked at any time, since, again, each refining vertex

Algorithm 2: Refinement (using either "left" or "right" splits) of an ordered list of trees by the set X

```
Let $T_1, \dots, T_k$ be the maximal subtrees in the forest whose leaves are all in $X$;
Let $P_1, \dots, P_\ell$ be the set of parents of the roots of the $T_i$'s;
foreach $P_i$ do
    Let $A$ be the set of $P_i$'s children amongst the $T_j$'s, and $B$ its remaining
    children;
    Let $T_a$ either be the single tree in $A$ or the tree formed by unifying the trees
    in $A$ under a common root, and define $T_b$ symmetrically;
    When unifying under a common root, assign $P_i$'s label to this root;
    if $P_i$ is a root then
        Replace $P_i$ in the forest with either $T_a, T_b$ (for a left split) or $T_b, T_a$ (for
        a right split)
    else
        Replace the children of $P_i$ with $T_a$ and $T_b$;
    end
    Mark the roots of $T_a$ and $T_b$ and all their ancestors;
    Mark the children of the prime nodes marked above;
end
```

is either universal to them or isolated from them. So for contradiction, assume that after refinement the c_i's have a sibling c different from them. Inspecting algorithm 1, we see that c must have been a sibling of the c_i's in $T(N_i)$, and that c and the c_i's must have the same set of neighbours outside N_i. Hence, $c \cup c_1$ is a module overlapping $c_1 \cup \cdots \cup c_k$, contradicting the latter being strong.

For the converse, consider a node N without any marked children, and suppose N was formed from the refinement of $T(N_i)$. Clearly, the vertices of N have the same neighbours outside $T(N_i)$. By algorithm 1, if N is prime, it existed in $T(N_i)$ and so has the same neighbours within $T(N_i)$. This is also true when N is not prime, since its children must have been children of the same non-prime node in $T(N_i)$. Hence, each node with unmarked children is a module. If the node existed in $T(N_i)$ then it is clearly strong. If it is new, a simple case analysis shows that no other module can overlap it, since two overlapping modules must be a module themselves.

Proof of lemma 2. [**Sketch**] Recall the statement of the lemma, and the bounding trees T_i and T'_j. We prove this by induction on the number of vertices refining. Prior to refinement we have the ordered list of trees $T(N_0), x, T(N_1), \dots, T(N_k)$. It is easy to show that if $M' \cap N_i \neq \emptyset$ for some $i > 1$, then $M' = V$. Thus, the lemma holds prior to refinement since $T(N_0)$ and either $T(N_1)$ or $T(N_k)$ can be taken as the bounding trees. So suppose there are such bounding trees T_i and T'_j after some number of vertices have refined; now consider what happens after the next vertex refines. Clearly we need only focus on T_i and T'_j; we'll argue the case for T_i, with the case for T'_j being similar.

Now, if T_i is not split we are done, so assume T_i is split and replaced by the trees T_A, T_B in order. Let v be the vertex doing the refining and observe that v is universal to the leaves of T_A and not universal to the leaves of T_B; additionally, we must have $v \in N_1$. If $v \in M'$ as well, then v is universal to the portion of T_i outside M' and hence we take T_A as the new left-bounding tree. If $v \notin M'$, then it is isolated from the portion of T_i in M', and so we take T_B as the new left-bounding tree in this case.

3.2 Promotion

The promotion process is given by algorithm 3. Below we sketch the proof of lemma 3. The key here is that refinement uses two types of marks, "left" and "right", with promotion handling each differently.

Algorithm 3: The promotion algorithm

```
while there is a root $r$ with a child $c$ both marked by "left" do
    Remove from $r$ the subtree rooted at $c$ and place it just before $r$;
end
while there is a root $r$ with a child $c$ both marked by "right" do
    Remove from $r$ the subtree rooted at $c$ and place it just after $r$;
end
Delete all marked roots in the forest with one child, replacing them with that child;
Delete all marked roots in the forest with no children;
Remove all marks;
```

Proof of lemma 3. [**Sketch**] By lemma 4 and inspection of algorithm 3, we see that the strong modules not containing x will appear consecutively after promotion.

Let M be a strong module containing x. Let T_i and T'_j be the bounding trees provided by lemma 2. It suffices to show that promotion deletes nodes in such a way as to place the portions of T_i and T'_j that are in M next to the other vertices in M. We'll focus on T'_j, with the case for T_i following similarly.

In the proof of lemma 2 we observed that if $M \cap N_i \neq \emptyset$ for some $i > 1$, then $M = V$. As such, we'll assume T'_j is composed of vertices in N_1. If T'_j only contains vertices in M, then clearly we are done since promotion does not rearrange trees in our ordered list. So assume T'_j contains some vertices in M and some outside M. By proposition 1, this means it contains vertices in at least two different components of $G[N_1]$, say C and C' with $C \subset M$ and $C' \cap M = \emptyset$. Now, C and C' were siblings at depth-1 in $T(N_1)$, and by assumption, some portion of each remains in the same tree after refinement. Appealing to algorithm 1, we see that this is only possible if all vertices in C and C' remain in the same tree after refinement; that is, C and C' must still be siblings after refinement, which means they remained siblings throughout refinement.

If both C and C' share the same neighbours outside N_1, then $C \cup C'$ is a module overlapping M, contradicting M being strong. It follows that at least one of C and C' is marked "left" or "right" (or both). We now consider the cases:

Case 1: Assume C' is marked by "left". This means a vertex in N_0 is adjacent to some but not all vertices in C'; let v be the first such vertex. Note that $v \notin M$ if it is adjacent to some of C'; thus, v is universal to C. But we remarked above that C and C' had the same parents throughout refinement; so at the time v refined, it would have split C away from C', contradicting their being siblings afterwards. This case is therefore impossible.

Case 2: Assume C' is marked by "right". Observe that no vertex in C can be adjacent to a vertex in N_i, $i > 1$, since such vertices are outside M and not adjacent to x. Thus C cannot be marked by "right". Thus, promotion places the vertices of C' to the right of those in C.

Case 3: Assume C' is not marked by a split. Then C must be marked by a split, as argued above, and as seen in case 2, it must be a left-split that marks it. Thus, promotion places the vertices of C to the left of those of C'.

In all cases, promotion puts C to the left of C'. Since C and C' were chosen arbitrarily, we can conclude that the vertices of M appear consecutively.

3.3 Assembly

Recall from the overview that to assemble the tree it suffices to insert the brackets delineating the strong modules containing x. We can simplify this by viewing our ordered list of trees as an ordered list of (co-)components. The (co-)components of the layers appear at depth-0 and depth-1 in the $T(N_i)$'s and thus appear consecutively prior to refinement. Examining algorithms 1,2, and 3 we see that they will remain consecutive after promotion. Thus, our ordered list of trees can be seen as an ordered list of (co-)components: $C'_\kappa, \ldots, C'_1, x, C_1, \ldots, C_\lambda$, where the C'_i's correspond to the co-components of N_0 and the C_i's correspond to the components of the remaining layers. The process to insert the brackets is derived from the lemma below:

Lemma 5. *Let M be the smallest strong module containing x. Then M satisfies one of the following three conditions:*

(i) M is the maximally contiguous module containing x and no C'_i (in which case M is series);
(ii) M is the maximally contiguous module containing x and no C_i, and only C'_j's in N_1 with no edge to their right (in which case M is parallel);
(iii) M is the minimally contiguous module containing x and at least C_1 and C'_1 (in which case M is prime).

We can determine if a C_i has an edge to its right by checking each vertex's incident active edges. To help identify the modules required by the lemma we associate with each (co-)component a μ-value, defined as follows: for C'_i, let C_j be the component with smallest index such that C'_i is isolated from $C_\lambda, \ldots, C_j$, then if $j \neq 1$, $\mu(C'_i) = C_{j-1}$, otherwise $\mu(C'_i) = x$. The μ-values for the C_i's are defined symmetrically.

We apply the lemma by operating greedily. Let M be the smallest strong module containing x. We first check if M is a parallel module by comparing $\mu(C_1)$ with x; if they are the same, we then check that C_1 has no edge to its right. If both tests succeed then M is a parallel module and we maximally add consecutive C_i's that satisfy both tests. If either test fails then a series module is attempted in a symmetric manner.

Failing both a series and parallel module, we know M is prime and must include C_1 and C_1'. In this case M is formed by iteratively applying the following rule: when a C_i is included in M, so too must be $C_1', \ldots, \mu(C_i)$, and symmetrically for a C_i' added to M. When no new (co-)components can be added we know we have found M. In all cases, once a module is found, brackets are inserted, the module contracted, and the process begins again with the just identified module in the role of x.

4 Running Time and Implementation

4.1 Recursion

In order to effect the partitioning required of the recursion, we need to traverse the pivot's adjacency list in its entirety. However, each vertex is a pivot exactly once during the algorithm, so this is consistent with linear-time.

We will need to isolate the incident active edges of each vertex so that refinement, promotion, and assembly can be performed efficiently; this can be done during the recursion. Initially we assume all vertices are marked as *unvisited* and that each has associated with it an empty list denoted by α (which will be used to store the incident active edges). As pivots are chosen during the recursion they are marked as *visited*. When a pivot's adjacency list is traversed, the pivot is appended to the α-list of all its visited neighbours. Thus, after recursion the α-lists of each vertex in N_i will correspond to their incident active edges to N_{i+1}. The rest of their active edges can then be added by traversing the α-list of each vertex, and appending vertices to the other α-lists in the obvious way. At the end of each stage the α-lists must be cleared to satisfy our induction hypothesis. We can thus assume that the active edges at each stage can be isolated at the cost of work proportional to their number. Notice that each edge is active precisely once during the algorithm, so this effort is consistent with linear-time overall.

4.2 Refinement

A simple recursive marking procedure finds the maximal subtrees required by algorithm 2. All nodes in our trees have at least two children, so the sizes of these subtrees are linear in the number of their leaves, which is equal to the number of incident active edges of the vertex refining. Notice that each vertex has at least one incident active edge. Thus, finding these trees is proportional to the number of active edges at each stage and so is consistent with linear-time.

The children of a prime node need only be marked once, and the ancestors of a node need only be marked twice (once each for "left" and "right"). The time for this marking is therefore proportional to the size of our ordered forest, which is linear in the number of its leaves, which is linear in the number of active edges (since each leaf has at least one active edge), and hence consistent with linear-time overall.

4.3 Promotion

If we implement promotion in a depth-first manner, we see that it requires no more than a single traversal of our ordered forest, which as just observed, is consistent with linear-time.

4.4 Assembly

Identifying the (co-)components requires at most two traversals of the forest: one prior to refinement to mark them and one after promotion to retrieve them. Determining if a C'_i has an edge to its right needs only a traversal of each vertex's α-list. Computing the μ-values of the (co-)components can be accomplished by processing each vertex in order and traversing its α-list. All this work is therefore consistent with linear-time.

The placement of the brackets amounts to a single traversal of the list of (co-)components, each of which contains an active edge, and so is consistent with linear-time.

The final assembly of the tree can be done merely by traversing our ordered forest, and is therefore consistent with linear-time.

5 Conclusion

Given the fundamental relationship between modular decomposition and transitive orientation, the natural question to ask is whether the ideas here can be applied to the latter problem. In fact, the authors are confident they can. Modular decomposition for directed graphs should also be amenable to this approach. Code for the algorithm can be found at `www.cs.toronto.edu/~mtedder`; the webpage also provides a detailed example of the algorithm's execution.

References

1. C. Capelle, M. Habib, and F. de Montgolfier. Graph decompositions and factorizing permutations. *Discrete Mathematics and Theoretical Computer Science*, 5:55–70, 2002.
2. M. Chein, M. Habib, and M.C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37:35–50, 1981.
3. D.G. Corneil, Y. Perl, and L.K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal of Computing*, 14:926–934, 1985.
4. A. Cournier and M. Habib. A new linear algorithm of modular decomposition. In *Trees in algebra and programming (CAAP)*, volume 787 of *Lecture Notes in Computer Science*, pages 68–84, 1994.

5. D.D. Cowan, L.O. James, and R.G. Stanton. Graph decomposition for undirected graphs. In *3rd S-E Conference on Combinatorics, Graph Theory and Computing, Utilitas Math*, pages 281–290, 1972.
6. E. Dahlhaus. Efficient parallel algorithms for cographs and distance hereditary graphs. *Discrete Applied Mathematics*, 57:29–54, 1995.
7. E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical algorithm for sequential modular decomposition algorithm. *Journal of Algorithms*, 41(2):360–387, 2001.
8. Celina M. H. de Figueiredo and Frédéric Maffray. Optimizing bull-free perfect graphs. *SIAM J. Discret. Math.*, 18(2):226–240, 2005.
9. A. Ehrenfeucht, H.N. Gabow, R.M. McConnell, and S.L. Sullivan. An $O(n^2)$ divide-and-conquer algorithm for the prime tree decomposition of two-structures and modular decomposition of graphs. *Journal of Algorithms*, 16:283–294, 1994.
10. Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, New York, NY, USA, 1983. ACM.
11. J. Gagneur, R. Krause, T. Bouwmeester, and G. Casari. Modular decomposition of protein-protein interaction networks. *Genome Biology*, 5(8):R57, 2004.
12. T. Gallai. Transitiv orientierbare graphen. *Acta Math. Acad. Sci. Hungar.*, 18:25–66, 1967.
13. M. Habib, F. de Montgolfier, and C. Paul. A simple linear-time modular decomposition algorithm for graphs, using order extension. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 3111 of *Lecture Notes in Computer Science*, pages 187–198, 2004.
14. M. Habib and M.C. Maurer. On the x-join decomposition of undirected graphs. *Discrete Applied Mathematics*, 1:201–207, 1979.
15. M. Habib, C. Paul, and L. Viennot. A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata. In *15th Symposium on Theoretical Aspect of Computer Science (STACS)*, volume 1373 of *Lecture Notes in Computer Science*, pages 25–38, 1998.
16. R.M. McConnell and J. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 536–545, 1994.
17. R.M. McConnell and J. Spinrad. Ordered vertex partitioning. *Discrete Mathematics and Theoretical Computer Science*, 4:45–60, 2000.
18. R.H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Orders*, pages 41–101. D. Reidel, Boston, 1985.
19. R.H. Möhring. Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions. *Annals of Operations Research*, 4:195–225, 1985.
20. R.H. Möhring and F.J. Radermacher. Substitution decomposition for discrete structures and connections with cominatorial optimization. *Annals of Discrete Mathematics*, 19:257–356, 1984.
21. J.H. Muller and J. Spinrad. Incremental modular decomposition. *Journal of the ACM*, 36(1):1–19, 1989.
22. C. Papadopoulos and C. Voglis. Drawing graphs using modular decomposition. In Patrick Healy and Nikola S. Nikolov, editors, *Graph Drawing, Limerick, Ireland, September 12-14, 2005*, pages pp. 343–354. Springer, 2006.
23. A. Pnueli, S. Even, and A. Lempel. Transitive orientation of graphs and identification of permutation graphs. *Canad. J. Math.*, 23:160–175, 1971.