

DECLINE AND FALL OF THE ICALP 2008 MODULAR DECOMPOSITION ALGORITHM

WILLIAM ATHERTON AND DMITRII V. PASECHNIK

ABSTRACT. We provide a counterexample to a crucial lemma in ICALP 2008 paper [7], invalidating the algorithm described there.

1. INTRODUCTION

Graph modular decomposition is an important technique in graph theory, and a well-studied algorithmic problem, with dozens of different algorithms published since the pioneering work [3], see e.g. [4]. One highly cited linear time algorithm has been presented at ICALP 2008 [7], also published as [6]. A Java implementation [5] has been made available by the first author of [loc.cit.]. An archival copy of this implementation can be found in [8]. As well, a number of attempts has been made to implement it in a different language, one of them by the first author of this note, as a part of an undergraduate project [1] supervised by the second author in Spring of 2023. At the testing phase of the implementation, done in SageMath [9], an example, see Sect. 2, has been found which produced an obviously incorrect output. The problem was traced back to a lemma in [6], which is invalidated by the example. As well, the implementation [5] run on this example produced basically the same incorrect output.

We contacted the authors of [6] in May 2023, who were quick to acknowledge the problem to us. In March 2024 they published a revision [2] of [6], without mentioning the problem in [loc.cit.] and in the associated with it implementation [5], and without mentioning our communication. The main purpose of this note is to publicise the problem in [6], [7], and thus to stop further waste of time stemming from attempts to implement the incorrect algorithm.

We quickly recall that for a graph G with the vertex set $V := V(G)$, a *module* is a subset of vertices $M \subseteq V$ s.t. any two $x, y \in M$ cannot be distinguished by any $v \in V \setminus M$, i.e. v is either simultaneously adjacent to x and y , or non-adjacent to x and y . Modules V and singleton modules are called *trivial*. Note that connected components of G and of the complement graph $\bar{G}$ are modules. A module M can be properly contained in another module $M' \neq M$, or they can *overlap*, in the sense that $M \cap M' \neq \emptyset$, but neither $M' \subset M$, nor $M \subset M'$. A module which does not overlap any module is called *strong*. A *modular decomposition tree* is a recursive partition (also known as a laminar set family) of V into strong modules; an example may be seen on Fig. 1. More details may be found in e.g. [4].

2. THE EXAMPLE

The counterexample graph G which was discovered is described here. Note that not only its isomorphism type matters here, but the particular vertex ordering—different orderings lead to different outcomes, for some of them the result is correct. We mark nodes with letters $a, b, \dots, i$, to adhere to the vertex labeling conventions of [5]. G is the join of two graphs: the 4-cycle with

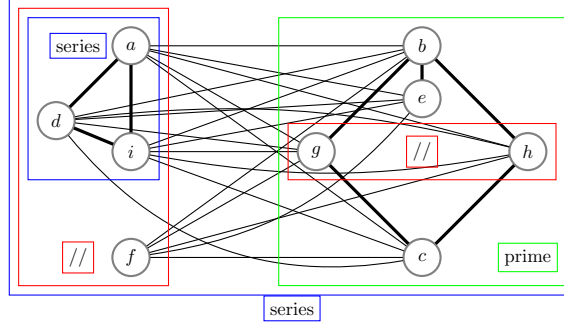


FIGURE 1. The modular decomposition of G , with "series", resp. "parallel" (abbreviated "//"), resp. "prime", nodes of the decomposition tree are show by blue, resp. red, resp. green boxes.

an extra edge attached at a vertex, and the disjoint union of a vertex and the 3-cycle. G and its modular decomposition is shown on Fig. 1.

The implementation [5] on this example produces

```
(SERIES, numChildren=5
  (label=e, neighbours:a,b,d,f,i), (label=b, neighbours:a,d,e,f,g,h,i),
  (label=c, neighbours:a,d,f,g,h,i),
  (PARALLEL, numChildren=2
    (label=g, neighbours:a,b,c,d,f,i), (label=h, neighbours:a,b,c,d,f,i)),
  (PARALLEL, numChildren=2
    (label=f, neighbours:b,c,e,g,h),
    (SERIES, numChildren=3
      (label=j, neighbours:a,b,c,d,e,g,h), (label=a, neighbours:b,c,d,e,g,h,i),
      (label=d, neighbours:a,b,c,e,g,h,i))))
```

which is obviously incorrect, there is no prime node! Or, alternatively, note that the top series node contains 5 children, i.e. the corresponding quotient graph is K_5 , and in particular there must be an edge between the singleton nodes b and c . The latter would only be possible if (b, c) was an edge in G —which is not the case.

Alternatively one could compute the decomposition of the complement $\overline{G}$ of G . Such decompositions must be "dual" to each other, in the sense that one must swap meanings of "series" and "parallel". Indeed, [5] works correctly on $\overline{G}$ and produces, modulo the swap just mentioned, the decomposition at Fig. 1.

```
(PARALLEL, numChildren=2
  (SERIES, numChildren=2
    (label=f, neighbours:a,d,i),
    (PARALLEL, numChildren=3
      (label=i, neighbours:f), (label=a, neighbours:f), (label=d, neighbours:f))),
  (PRIME, numChildren=4
    (label=b, neighbours:c), (label=c, neighbours:b,e), (label=e, neighbours:c,g,h),
    (SERIES, numChildren=2
      (label=h, neighbours:e,g), (label=g, neighbours:e,h))))
```

3. FAULTY LEMMA

We were able to trace the flaw down to Lemma 4 in [7] (which is Lemma 3.1 in the preprint version [6]). The latter lemma plays a crucial role in the proof of correctness and is shown false on our example.

Let x be an arbitrary vertex of G .

Lemma (Lemma 4 in [7]). The nodes in the ordered list of trees resulting from refinement that do not have marked children correspond exactly to the strong modules **not** containing x .

First, note a typo in the statement of Lemma 4, which we corrected above in **boldface** - the missing “not”.

This is a misprint. The proof of the Lemma is clearly proving the statement with “not” inserted. As well, the way it is used in the proof of Lemma 3 in [7] also indicated the correct statement should have the missing “not”.

We now show where the algorithm is going wrong on G . It starts by choosing a vertex x to be the first *pivot* (this choice is arbitrary in the algorithm), and recurses on $G(x)$, the set of the neighbours of x , a tentative (strong) module. Then it processes the non-neighbours $\overline{G(x)}$, resulting in a number of tentative modules as in (1), and, finally, does a refinement step: rearranging tentative modules into the modules for the tree. For a detailed complete description, see [7].

$$(1) \quad \underbrace{T(N_0), x}_{G(x)}, \underbrace{T(N_1), \dots, T(N_k)}_{\overline{G(x)}}$$

Let the algorithm choose the vertex i to be the first *pivot*. We get the neighbours of i , $G(i) = \{a, b, c, d, e, g, h\}$, $\{i\}$, and $\overline{G(i)} = \{f\}$.

It then recursively processes the neighbour partition. The recursively computed modular decomposition for $G(i)$ can be seen on Fig. 1; one has to remove i , f , and the two nontrivial decomposition tree nodes (strong modules) containing i (i.e. the two nested boxes on the left of the green box).

The next step of the algorithm, “pull-forward”, is skipped, as there is only one node, f , in $\overline{G(i)}$.

It then calculates the modular decomposition of $\overline{G(i)}$, which is just the single-vertex tree consisting of f .

It then goes on to the refinement step, where the error lies.

The refinement process consists of Algorithms 1 and 2 from [7], which we copy here verbatim. We process the following decomposition of the vertices of G into subtrees of modules.

Lemma 4 is necessary for the correctness of the main algorithm. However, we will be going through Algorithm 1 on the given example G , and show that its result contradicts Lemma 4. We start its loop from vertex f . Then $v = f$ and $\alpha(f)$ is the list of incident active edges of f .

Definition 1 ([7]). An edge becomes *active* when one of its endpoints is a pivot or if its endpoints reside in different layers.

As f is in its own layer, all its incident edges are active. Therefore $\alpha(f) = \{bf, cf, ef, gf, hf\}$. Algorithm 2 is then run on the ordered list of trees, with the set $X = \{b, c, e, g, h\}$.

It first calculates $T_1, \dots, T_k$, the maximal subtrees in the forest whose leaves are all in X . The list of maximal subtrees in this case is a single subtree corresponding to the whole of X is given by the prime node in Fig. 1, which we refer to as T_1 .

P_1 , the parent of T_1 , is the topmost node from the modular decomposition of $G(i)$.

Algorithm 1: Refinement of the ordered list of trees (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 then
    | refine using left splits, and when a node is marked, mark it with "left";
  else if  $v$  is to  $x$ 's right and refines a tree to  $x$ 's left then
    | refine using left splits, and when a node is marked, mark it with "left";
  else if  $v$  is to  $x$ 's right and refines a tree to  $x$ 's right then
    | refine using right splits, and when a node is marked, mark it with "right";
  end
end

```

Algorithm 2: Refinement 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  $T_i$ 's;
foreach non-prime  $P_m$  do
  Let  $A$  be the set of  $P_m$ '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;
  Assign  $P_m$ 's label to  $T_a$  and  $T_b$ ;
  if  $P_m$  is a root then
    | Replace  $P_m$  in the forest with either  $T_a, T_b$  (left split) or  $T_b, T_a$  (right split)
  else
    | Replace the children of  $P_m$  with  $T_a$  and  $T_b$ ;
  end
  Mark the roots of  $T_a$  and  $T_b$  as well as all their ancestors;
end
foreach prime  $P_m$  do
  | Mark  $P_m$  as well as all of its children and all of its ancestors;
end

```

As there is only one P_k , the outer loop is only run once, on P_1 .

A is the set of P_1 's children among the T_j 's, which is just the singleton set T_1 , and B is the set of remaining children, which is the singleton—the bottommost parallel node from the modular decomposition of $G(i)$.

As $|A| = |B| = 1$, we have that $T_a = A = T_1$, and $T_b = B$.

P_m is a root, so P_m is replaced by T_a, T_b , as the subtree is to the left of the pivot.

Then, the roots of T_a and T_b are marked, and so are all their ancestors.

This means that the root of T_a , the prime node at the top of T_1 , is marked.

The algorithm then marks the children of all prime nodes marked this way, so the children of the prime node at the top of T_1 is marked, meaning the nodes b, c, e , and the parallel node from T_1 are all marked.

This is a contradiction to Lemma 4. Lemma 4 states that the nodes in the ordered list of trees resulting from refinement that do not have marked children correspond exactly to the strong modules not containing x . This means that the strong modules not containing i must not have a marked child. However, as you can see from Fig. 1, T_1 is a strong module not containing i , but it has marked children. $\square$

4. CONCLUSION

From the implementation point of view, the fact that the children of T_1 get marked means that when the Promotion step happens, the children of T_1 get split from T_1 , and the T_1 node gets deleted. As the rest of the algorithm assumes that strong modules not containing x are not affected by refinement, these nodes do not get reassembled back into a prime node, so you get the error occurring in the example implementation, where the prime node is missing.

This is a fundamental problem with the algorithm, as Lemma 4 is used to prove correctness of the algorithm, and the fact that children of prime nodes get marked in Lemma 2 is important for other cases of the algorithm to work correctly. Apparently the idea is not easy to salvage, as [2] appears to take a quite different approach, using LexBFS.

REFERENCES

- [1] W. Atherton. *Implementing Linear-time Modular Decomposition in SageMath*. 3rd year project dissertation, supervised by D. Pasechnik. Department of Computer Science, Oxford University, May 2023.
- [2] D. Corneil, M. Habib, C. Paul, and M. Tedder. *A recursive linear time modular decomposition algorithm via LexBFS*. 2024. arXiv: 0710.3901 [cs.DM].
- [3] T. Gallai. “Transitiv orientierbare Graphen”. German. In: *Acta Math. Acad. Sci. Hung.* 18 (1967), pp. 25–66. ISSN: 0001-5954. DOI: 10.1007/BF02020961.
- [4] M. Habib and C. Paul. “A survey of the algorithmic aspects of modular decomposition”. In: *Computer Science Review* 4.1 (2010), pp. 41–59. ISSN: 1574-0137. DOI: j.cosrev.2010.01.001. URL: <https://www.sciencedirect.com/science/article/pii/S157401371000002X>.
- [5] M. Tedder. *Homepage with a link to Java code*. The page was removed in Nov. 2023, a copy is in [8]. A copy of the code, retrieved earlier, is included here. 2008. URL: <https://web.archive.org/web/20231117180003/h>.
- [6] M. Tedder, D. Corneil, M. Habib, and C. Paul. *A recursive linear time modular decomposition algorithm via LexBFS*. 2008. arXiv: 0710.3901v2 [cs.DM].
- [7] M. Tedder, D. Corneil, M. Habib, and C. Paul. “Simpler Linear-Time Modular Decomposition Via Recursive Factorizing Permutations”. In: *Automata, Languages and Programming*. Ed. by L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfssdóttir, and I. Walukiewicz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 634–645. ISBN: 978-3-540-70575-8. DOI: 10.1007/978-3-540-70575-8_52.
- [8] *Archive of M. Tedder’s Java code for [7]*. URL: <https://github.com/dimpase/MDTreeJavaCode>.
- [9] The SageMath Developers. *SageMath, the Sage Mathematics Software System*. Version 10.3. 2024. DOI: 10.5281/zenodo.8042260. URL: <https://www.sagemath.org>.

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF OXFORD
 Email address: william.atherton@keble.ox.ac.uk

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF OXFORD
 Email address: dima.pasechnik@cs.ox.ac.uk