Serdica J. Computing **12** (2018), No 4, 265–280

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

A SIMPLE RANDOMIZED 3-EDGE CONNECTED COMPONENT ALGORITHM

Vladislav Haralampiev

Abstract. Finding the 3-edge connected components of a graph is a well-researched problem for which many algorithms are known. In this paper, we present a new linear-time randomized algorithm for the problem. To the best of our knowledge, this is the first randomized algorithm for partitioning a graph into 3-edge connected components. The algorithm is a composition of simple building blocks, it is easy to understand and implement, and it has no corner cases.

1. Introduction. Connectivity is a fundamental concept in graph theory. It has applications in a wide variety of areas such as network reliability, VLSI design, and security. The 3-edge connectivity problem is a special case of the general edge connectivity problem in which we want to find the pairs of vertices remaining connected after the removal of any two edges from the input graph. In addition to the already mentioned areas, finding the 3-edge connected components of a graph is useful in fields such as bioinformatics [4] and quantum chemistry [2].

ACM Computing Classification System (1998): G.2.2, G.3.
Key words: graph connectivity, 3-edge connected components, randomized algorithm.

Finding the 3-edge connected components of a graph is an old and well-researched problem. Consequently, many algorithms are known for it. The first one is due to Galil and Italiano [5]. They reduce the problem to 3-vertex connectivity and use Hopcroft and Tarjan's linear-time algorithm for 3-vertex connectivity [6]. The resulting algorithm is rather complex and hard to implement. A number of alternative linear-time algorithms for the 3-edge connectivity problem have been published, for example [10], [8] and [7]. All these algorithms are based on properties of depth-first search and are rather complex, too.

In [9] Pritchard proposes a simple randomized circulation-based method for determining small cuts in graphs. This method can be used to find cut pairs, pairs of edges which are useful for determining 3-edge connectivity. In the same paper, Pritchard proposes an algorithm for finding 3-edge connected components, but his definition of a 3-edge connected component is different from the commonly accepted one.

In this paper we use Pritchard's method for finding cut pairs to develop a randomized algorithm for partitioning a graph into 3-edge connected components. The proposed algorithm is efficient and easy to understand and implement. For practical inputs the algorithm is linear with very small probability of error, which can be made arbitrarily small.

2. Definitions.

We consider connected, undirected graphs. The vertex set of the graph G is denoted by V and the edge set, by E.

Definition 1 (edge cut). *The set of edges $S \subseteq E$ is called an edge cut for the vectices $u, v \in G$ iff the removal of the edges in S disconnects u and v.*

Definition 2 (k-edge connected vertices). *Two vertices $u, v \in G$ are called k-edge connected iff there is no edge cut for these vertices of cardinality less than k.*

It is easy to show that k-edge connectivity is an equivalence relation.

Definition 3 (k-edge connected component). *The classes of the k-edge connectivity relation are called the k-edge connected components. Graph G is called k-edge connected iff G has precisely one k-edge connected component.*

Edge cuts of cardinality 1 are often called bridges. There is a well-known simple algorithm for finding all the bridges and 2-edge connected components in a graph in linear time. Since 3-edge connected components cannot span multiple 2-edge connected components, further in the paper it is assumed that the input graph G is 2-edge connected. If it is not, we can first extract the 2-edge connected components and then process them one by one. Alternatively, we can find the 2-edge connected components in the same framework as the proposed algorithm for finding the 3-edge connected components (this is briefly mentioned in Appendix A).

Definition 4 (cut pair)**.** *An edge cut of cardinality 2 for two vertices u and v in a 2-edge connected graph is called a cut pair for u and v.*

In [9] it is shown that we can assign short labels (bit strings) to the edges of the graph in such a way that two edges e_1 and e_2 form a cut pair iff $label(e_1) = label(e_2)$. Here and further in the paper we use $label(e)$ to refer to the label assigned to edge e. The procedure which generates these labels is very simple and elegant and is briefly described in Appendix A.

The method for partitioning a graph into 3-edge connected components, proposed in Section 3, is based on depth-first search (DFS), a well-known graph traversal algorithm. Given a connected, undirected graph, depth-first search produces a spanning tree and classifies the edges into two categories: tree edges and back edges [3]. The following definitions will often be used in the paper:

Definition 5 (DFS-tree, tree and back edges)**.** *Let G be a connected, undirected graph.*

- *A DFS-tree of G is the spanning tree produced by some depth-first search traversal of G. Note that the DFS-tree is a rooted tree.*
- *Given a DFS-tree T, every edge $e = (u, v) \in E$ is called a tree edge if it belongs to T. When we write tree edges as pairs of vertices, we assume that the first vertex in the pair is closer to the root of the DFS-tree.*
- *Given a DFS-tree T, every edge $e = (u, v) \in E$ is called a back edge if it is not a tree edge and, in T, vertex v is on the path from the root to u (or vice versa). When we write back edges as pairs of vertices, we assume that the second vertex in the pair is closer to the root of the DFS-tree.*

It is well-known that in undirected graphs there are only tree edges and back edges. Note that the DFS-tree of a graph is not unique. Different DFS traversals may produce different DFS-trees and classifications of the edges. From now on, when we write about DFS-tree, tree edges and back edges, we assume that we have fixed one arbitrary depth-first traversal of the input graph.

The algorithm presented in the next section uses properties of cut pairs and depth-first search to find parts of the DFS-tree which do not belong to the same 3-edge connected component. This information is stored using the notion of *color*. More specifically, assume we have a source of unique colors (objects which we can test for equality) and we assign a set of colors to each vertex of the tree. The set of colors assigned to vertex u is denoted as $ColorSet(u)$ and we will assign the colors in such a way that $ColorSet(u) = ColorSet(v)$ iff u and v are in the same 3-edge connected component. The algorithm from Section 3 modifies the colors of the vertices by *coloring operations*:

Definition 6 (coloring operations). *Denote by T the DFS-tree of G.*

- *Operation $AddColorToSubtree(vr, new\_color)$: for each vertex in the subtree of vr in T, including vr, add new_color to the set of colors of the vertex.*
- *Operation $RemoveColorFromSubtree(vr, new\_color)$: for each vertex in the subtree of vr in T, including vr, remove new_color from the set of colors of the vertex, assuming that the color is present in the set.*
- *Operation $GetColors(vr)$: return the set of colors of the vertex vr.*

The data structure that we use to implement the operations from Definition 6 can be offline, meaning that we first receive a sequence of $AddColorToSubtree$ and $RemoveColorFromSubtree$ operations, then we do some processing, and after that we need to be able to answer $GetColors$ queries. There are various data structures which can support coloring operations. In Section 4 we describe a randomized data structure that supports them in constant time.

3. 3-edge connected component algorithm.

It may seem that for finding the 3-edge connected components of a 2-edge connected graph we can just remove all cut pairs and return the connected components of the resulting graph. However, this is not correct and Fig. 1 shows a counterexample. The vertices A and B in the figure are 3-edge connected but if we remove all cut pairs, the graph will not have any edges at all. Intuitively, the problem is that vertices in one 3-edge connected component may be connected through vertices from other components.

Notice that, given two vertices are in different connected components after the removal of a cut pair, we surely know these vertices do not belong to the same 3-edge connected component. So, a simple algorithm for finding these components is to remove cut pairs one by one and remember which pairs of vertices do not belong to the same 3-edge connected component. Unfortunately, the number

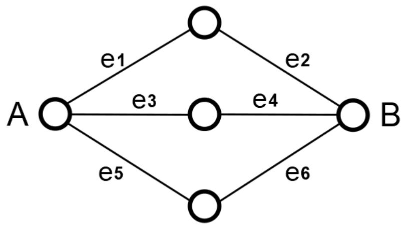

Fig. 1. $\{e_1, e_2\}, \{e_3, e_4\}$ and $\{e_5, e_6\}$ are the cut pairs. Removing them disconnects A from B. However, A and B are 3-edge connected

of cut pairs can be $\theta(n^2)$, so such an algorithm is slow. The main idea of our algorithm is to find a subset of $O(n)$ cut pairs that give complete information about the 3-edge connected components of the graph.

We first prove several lemmas that help us to efficiently identify the connected components of the input graph after the removal of a cut pair. After that we give the pseudocode of the algorithm and prove its correctness. As we said earlier, we assume that the input graph is 2-edge connected.

Lemma 1. *In any DFS-tree of G it is impossible for both edges in a cut pair to be back edges.*

Proof. Assume both edges in a cut pair are back edges. After removing them the graph will still be connected by tree edges, a contradiction to the definition of cut pair. □

The next lemma captures the intuition that, if both edges in a cut pair are tree edges, then they are in the same "branch" of the DFS-tree.

Lemma 2. *Let $\{e_1, e_2\}$ be a cut pair in which both edges are tree edges. Then there exists a vertex v for which both e_1 and e_2 lie on the path from the root of the DFS-tree to v.*

Proof. Assume the lemma is not true and let $\{e_1 = (x, y), e_2 = (z, t)\}$ be a counterexample. We will denote the lowest common ancestor of y and t by l (see Fig. 2). If l is equal to y, than both e_1 and e_2 will be on the path from the root to t, which is impossible. With similar reasoning we can show that l is not equal to t and neither e_1 nor e_2 lie on the path from the root to l. The only possible configuration left for the vertices and edges is shown in Fig. 2. After removing e_1 and e_2 the DFS-tree splits into three connected components: A that contains the root, B that contains y, and C that contains t. Since e_1 is not a bridge,

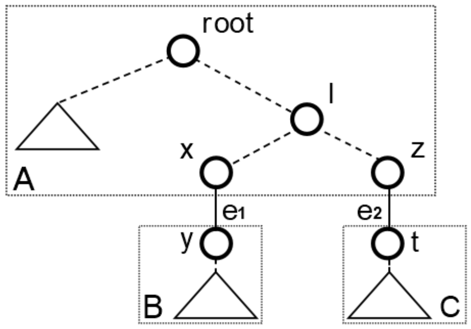

Fig. 2. Only tree edges are shown. Assuming that $\{e_1, e_2\}$ is a cut pair and neither e_1 nor e_2 is a bridge, this configuration is impossible

B is connected to A by back edges (B cannot be connected to C, because, by definition, there are no back edges between them). Similarly, C is connected to A by back edges. This means that after the removal of e_1 and e_2, the components A, B and C are still connected in the graph by back edges, so the whole graph is connected, which is a contradiction. □

For a given pair of edges, the next two lemmas characterize the vertices for which these edges are a cut pair. Intuitively, we can characterize these vertices through subtrees of the DFS-tree. This is very important because it allows us to compactly represent the information about pairs of vertices which do not belong to the same 3-edge connected component.

Lemma 3. *Let $\{e_1 = (x, y), e_2 = (z, t)\}$ be a cut pair in which e_1 is a tree edge and e_2 is a back edge. $\{e_1, e_2\}$ is a cut pair for the vertices u and v iff after removing e_1 from the DFS-tree u and v are in different connected components.*

Proof. Fig. 3 illustrates the lemma. After removing e_1, the DFS-tree splits into two connected components. Let A be the component containing y and B the other component. If $\{e_1, e_2\}$ is a cut pair for u and v, then the two vertices clearly cannot both belong to A or B (because they are connected by tree edges). We know that $\{e_1, e_2\}$ is a cut pair, so there exist $x \in A$ and $y \in B$ which become disconnected after removing $\{e_1, e_2\}$. If there are two vertices $p \in A$ and $q \in B$ for which $\{e_1, e_2\}$ is not a cut pair, we can construct a path from x to y. To achieve that, we first go from x to p inside component A, then use the path from p to q and finally go from q to y inside component B. Since no path between x and y should exist, we know that $\{e_1, e_2\}$ is a cut pair for any two vertices $p \in A$ and $q \in B$. □

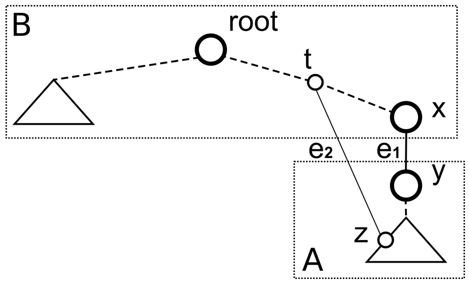

Fig. 3. Schematic graph showing the setup of Lemma 3.
Triangles and dashed edges stand for omitted subtrees and edges.
Edge (z, t) is a back edge, the other edges shown are tree edges

Lemma 4. *Let $\{e_1 = (x, y), e_2 = (z, t)\}$ be a cut pair in which both edges are tree edges. From Lemma 2 we know that there exists a vertex s for which both e_1 and e_2 are on the path from the root to s. Let us denote by T' the DFS-tree of G from which e_1 and e_2 are removed. Assume the vertex x is closer to the root than z and call B the connected component of y in T'. $\{e_1, e_2\}$ is a cut pair for the vertices u and v iff $u \in B, v \notin B$ or vice versa.*

Proof. The lemma is illustrated in Fig. 4. We know that both e_1 and e_2 are on the same path from the root to some vertex s. This means that, after the removal of e_1 and e_2, the DFS-tree of G splits into three connected components: one containing x, one containing y, and one containing t. In the statement of the lemma we called B the component containing y. Let us denote the component containing x by A and the component containing t by C. After removing e_1 and e_2 the components A, B and C are clearly disconnected from each other in the DFS-tree, but they could still be connected by back edges in the graph G.

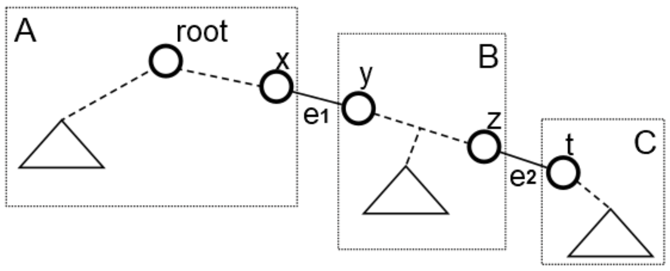

Fig. 4. Schematic graph showing the setup of Lemma 4. Only tree edges are shown. Triangles and dashed edges stand for omitted subtrees and edges

Note that the whole graph G without e_1 and e_2 is not connected, because $\{e_1, e_2\}$ is a cut pair.

Let us create boolean variables CB, CA and BA meaning that there exists a back edge from component C to component B, from component C to A and from component B to A. These are all possibilities for back edges. We know that e_1 is not a bridge, so $BA \vee CA$ is true. Similarly, $CB \vee CA$ is true because e_2 is not a bridge. If CA is false then the two statements above imply CB and BA are true, so there are back edges from C to B and from B to A. This in turn implies that the graph G with removed e_1 and e_2 is connected, a contradiction with $\{e_1, e_2\}$ being a cut pair. The contradiction is because we assumed CA is false, so CA needs to be true. With similar reasoning CB equals true or BA equals true implies G without e_1 and e_2 is connected, so we know that both CB and BA are false.

In the previous paragraph we have shown that in G with removed e_1 and e_2 A and C form one component (because CA is true, meaning there is a back edge between C and A) and B forms another component (because both CB and BA are false). This immediately implies the statement of the lemma, $\{e_1, e_2\}$ is a cut pair for the vertices u and v iff $u \in B, v \notin B$ or $v \in B, u \notin B$. $\square$

The previous two lemmas basically say that a cut pair gives us information that a connected part of the DFS-tree is in a different 3-edge connected component from the rest of the DFS-tree. This type of information can be conveniently represented using the coloring operations defined in Section 2. If we need to remember that part A of the DFS-tree is in a different 3-edge connected component from part B, we just need to add a unique color to the vertices in A. As we said earlier, iterating through all cut pairs and storing the information they give is too slow. So, what is left is to find a small subset of all the cut pairs which give us enough information to extract the 3-edge connected components. To achieve this, we will perform a depth-first traversal of the graph and, for each edge, we will find the closest edge with the same label on the path from the current vertex to the root (if it exists). We know that these two edges form a cut pair because they have equal labels. There can be more edges which form a cut pair with the current edge. Intuitively, we do not need to consider them because they all have the same label, so, in the previous steps of the depth-first traversal, we have already added unique colors to the parts of the DFS-tree defined by them. We do not need to duplicate this work. The intuition is formalized in Theorem 1, which proves the correctness of Algorithm 1, presented below.

Algorithm 1 3-edge connected components algorithm

```
 1: last_edge ← dict(default: None)
 2: visited ← [NO for vr ∈ V]
 3:
 4: function CreateColorSets(vr, parent ← None)
 5:     visited[vr] ← ON_PATH
 6:     for all (vr, u) ∈ E, u ≠ parent do
 7:         (x, y) ← last_edge[label(vr, u)]
 8:         if visited[u] = ON_PATH then                ▷ (vr, u) is a back edge
 9:             if (x, y) ≠ None then
10:                 AddColorToSubtree(y, NewColor())
11:         else
12:             if visited[u] = NO then                 ▷ (vr, u) is a tree edge
13:                 if (x, y) ≠ None then
14:                     color ← NewColor()
15:                     AddColorToSubtree(y, color)
16:                     RemoveColorFromSubtree(u, color)
17:                 last_edge[label(vr, u)] ← (vr, u)
18:                 CreateColorSets(u, vr)
19:                 last_edge[label(vr, u)] ← (x, y)
20:     visited[vr] ← YES
```

Theorem 1. *If we denote by* $ColorSet(vr)$ *the set of colors given to vertex vr by Algorithm 1, then two vertices u and v are in one 3-edge connected component* $\iff$ $ColorSet(u) = ColorSet(v)$.

Proof. $\Rightarrow$ Assume that for two vertices p and q there exists a color c, added by line 10 of the algorithm, such that $c \in ColorSet(p)$ and $c \notin ColorSet(q)$. This color was added by a pair of edges $e_1 = (vr, u)$ and $e_2 = (x, y)$ such that e_1 is a tree edge and e_2 is a back edge. We know that these edges form a cut pair, because they have equal labels (line 7). Also, we know that p is in the subtree of u in the DFS-tree and q is not (because we add the color c only to the subtree of u). Now, from Lemma 3 it follows that (e_1, e_2) is a cut pair for the vertices p and q. With similar reasoning for the case when the color c was added by lines 15–16 (and using Lemma 4) we can show that whenever we have two vertices p and q and a color c for which $c \in ColorSet(p)$ and $c \notin ColorSet(q)$, we can find a cut pair (e_1, e_2) for these vertices. Let u and v be two 3-edge connected vertices. If

$ColorSet(u) \neq ColorSet(v)$, then there exists a color c, which is in one of the sets, but not in the other. This means we can find a cut pair for u and v, so they are not 3-edge connected, a contradiction.
$\Leftarrow$ Let p and q be two vertices which are not 3-edge connected, but for which $ColorSet(p) = ColorSet(q)$. We know there exists a cut pair for these vertices because they are not 3-edge connected. Take one such cut pair $\{e_1 = (x, y), e_2 = (vr, u)\}$. There are two cases: either both e_1 and e_2 are tree edges or e_1 is a tree edge and e_2 is a back edge. The two cases are illustrated in Fig. 5. We will show that both lead to a contradiction, which proves that it is impossible for two vertices p and q to have $ColorSet(p) = ColorSet(q)$, but not be 3-edge connected.

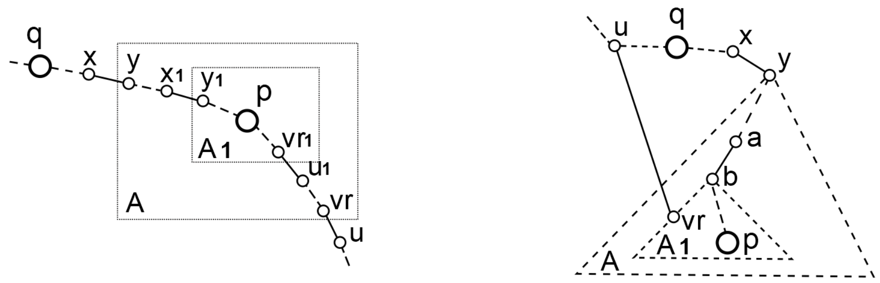

Fig. 5. Schematic graph showing the two possible cases for the cut pair $\{e_1 = (x, y), e_2 = (vr, u)\}$. Either both e_1 and e_2 are tree edges (left), or e_1 is a tree edge and e_2 is a back edge. Dashed lines in the graphs stand for omitted parts of edges and subtrees

Case 1: Both $e_1 = (x, y)$ and $e_2 = (vr, u)$ are tree edges.
From Lemma 4 we know that one of the vertexes p or q is in the part of the DFS-tree between $e_1 = (x, y)$ and $e_2 = (vr, u)$ and the other is not. Without loss of generality, let p be the vertex in the part between e_1 and e_2. Since e_1 and e_2 form a cut pair we, know that they have the same label L. Let (x_1, y_1) be the closest edge to p with label L on the path from p to x. Similarly, let (vr_1, u_1) be the closest edge to p with label L on the path from u to p (see Fig. 5). It is possible that $(x, y) = (x_1, y_1)$ and $(vr, u) = (vr_1, u_1)$. When we are processing the edge (vr_1, u_1) in Algorithm 1 (in the cycle from line 6), the condition on line 13 needs to be true, so we will add a unique color C to the vertexes in the part of the DFS-tree between y_1 and vr_1. Note that we have chosen (x_1, y_1) and (vr_1, u_1) in such a way that p is between them, so we will add C to $ColorSet(p)$. We know q is not in the part of the DFS-tree between y and vr. This means q is also not between y_1 and vr_1, so $C \notin ColorSet(q)$. Contradiction with $ColorSet(p) = ColorSet(q)$.

Case 2: $e_1 = (x, y)$ is a tree edge and $e_2 = (vr, u)$ is a back edge. From Lemma 3 we know that one of the vertices p and q is in the subtree of y in the DFS-tree, and the other is in the rest of the tree. Without loss of generality, let p be in the subtree of y. We know e_1 and e_2 have the same label L, because they form a cut pair. Let (a, b) be the closest edge to vr on the path from vr to x with label L (it is possible that $(a, b) = (x, y)$). When we process the edge (vr, u) in the cycle from line 6, the condition on line 9 is true (with edge (a, b)), so a unique color C is added to the subtree of b in the DFS-tree. The vertex q is not in this subtree, because we know q is not in the subtree of y, so $C \notin ColorSet(q)$. If p is in the subtree of b, then $C \in ColorSet(p)$, a contradiction with $ColorSet(p) = ColorSet(q)$. The other possibility is for p to be in the part of the DFS-tree between y and a. This means the edges (x, y) and (a, b), which have the same label, form a cut pair for the vertices p and q. Both (x, y) and (a, b) are tree edges, so we are in Case 1 and, again, have a contradiction with $ColorSet(p) = ColorSet(q)$. □

After the completion of Algorithm 1 we are able to determine if two vertices belong to the same 3-edge connected component by comparing their color sets. Partitioning the vertices into 3-edge connected components is equivalent to grouping them by color sets, which can easily be done with a hash table in expected linear time. Since Algorithm 1 is a simple extension of depth-first search, it also runs in linear time in the size of the graph, so we can find the 3-edge connected components with this complexity. However, we assume that operations with labels of edges and with color sets run in constant time. These are defined (in the next section) as bit strings and their length L controls the probability of error for the whole algorithm. The probability decreases exponentially with L. For practical instances of the problem, $L = 64$ makes the probability of error very close to 0. Since 64 is the size of a machine word, it is reasonable to assume that bitwise operations with bit strings of length 64 take constant time.

4. A data structure for the coloring operations.

The coloring operations are introduced by Definition 6. There are various ways to implement them, here we will present a randomized data structure achieving $\theta(1)$ time for all the operations with linear additional processing.

Colors will be represented as bit strings (nonnegative integers) and producing a new color will be implemented as generating a random integer. Also, instead of maintaining directly the set S of colors of a vertex, we will maintain a hash of the set defined as $hash(S) = \bigoplus_{s_i \in S} s_i$. Here $\oplus$ is the exclusive-or (XOR) operation. Later it will be shown that, if for two sets A and B $hash(A) = hash(B)$, then with high probability $A = B$.

It can be easily verified that, for a color c, $hash(S \cup \{c\}) = hash(S) \oplus c$ and $hash(S \setminus \{c\}) = hash(S) \oplus c$. This means we only need to support one operation – XOR the subtree of a given vertex (in the DFS-tree) by a given value. Additionally, all *GetColors* operations will be after the operations which modify the color sets, so we can process the modification operations offline (process them all at once during one traversal of the tree). Algorithm 2 presents the operations in pseudocode.

Algorithm 2 Implementation of coloring operations

```
function XorSubtree(vr, color)
    operations_for_vertex[vr] ← operations_for_vertex[vr] ⊕ color

function ExecuteOperations(vr, accumulated_hash ← 0)
    accumulated_hash ← accumulated_hash ⊕ operations_for_vertex[vr]
    color_set[vr] ← accumulated_hash
    for all followers nxt of vr do
        ExecuteOperations(nxt, accumulated_hash)

function AddColorToSubtree(vr, color)
    XorSubtree(vr, color)
function RemoveColorFromSubtree(vr, color)
    XorSubtree(vr, color)
function GetColors(vr)
    return color_set[vr]
```

Theorem 2. *Let A and B be two unequal sets of colors, in which each color is chosen independently at random from a universe of size T. Then, the probability that $hash(A) \neq hash(B)$ is at least $1 - \frac{1}{T}$.*

P r o o f. Assume A consists of the colors $\{a_1, a_1, ..., a_n\}$ and B consists of the colors $\{b_1, b_2, ..., b_m\}$. We know that $A \neq B$, which means there exists a color present in A but not in B, or vice versa. Without loss of generality, let $a_1 \in A$, $a_1 \notin B$. $hash(A) = hash(B) \iff \bigoplus_{i \in \{1...n\}} a_i = \bigoplus_{i \in \{1...m\}} b_i$. If we XOR both sides with $\bigoplus\limits_{i\in\{2\ldots n\}} a_i$, we get $a_1 = \bigoplus\limits_{i\in\{1\ldots m\}} b_i \oplus \bigoplus\limits_{i\in\{2\ldots n\}} a_i$. This means there is only one possible value of a_1 (assuming the other elements of A and B are fixed) such that $hash(A) = hash(B)$. We choose a_1 independently at random, so the probability to choose this specific value for a_1 is $\frac{1}{T}$. Now it follows that $hash(A) \neq hash(B)$ with probability $1 - \frac{1}{T}$. □

The previous theorem shows that the probability for a single hash equality check to give wrong result is at most $1/T$. Clearly, if, for every pair of vertices in the graph, this check gives correct result, we will be able to correctly partition the graph into 3-edge connected components. Let us denote by n the number of vertices. Assuming the size T of the universe of colors is large enough, we can approximate the probability of all checks to be correct by $\left(1 - \frac{1}{T}\right)^{n^2} \approx e^{\frac{-n^2}{T}}$. For example, if we use 64-bit type for storing the hash and the number of vertices is 1000000, then the probability of error is approximately $1 - e^{-\frac{1000000^2}{2^{64}}} \approx 5 \cdot 10^{-8}$.

5. Conclusion.

In this paper, a simple randomized algorithm for finding the 3-edge connected components of a graph is presented. The algorithm builds upon existing work for compactly describing cut pairs using random circulations. It finds a small subset of all cut pairs which gives enough information for partitioning the graph into 3-edge connected components. The partitioning is done with the help of the introduced coloring operations, for which one possible randomized implementation is given.

REFERENCES

[1] Ahuja R. K., T. L. Magnanti, J. B. Orlin. Network Flows: Theory, Algorithms, and Applications. New York, Prentice-Hall, Inc., Upper Saddle River, 1993.

[2] Corcoran J. N., U. Schneider, H.-B. Schüttler. Perfect Stochastic Summation in High Order Feynman Graph Expansions. *International Journal of Modern Physics C*, **17** (2006), No 11, 1527–1549.

[3] Cormen T. H., C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms. Third Edition. The MIT Press, 2009.

[4] Dehne F., M. Langston, X. Luo, S. Pitre, P. Shaw, Y. Zhang. The Cluster Editing Problem: Implementations and Experiments. In: H. L. Bodlaender, M. A. Langston (eds). Parameterized and Exact Computation. IWPEC 2006. *Lecture Notes in Computer Science*, **4169** (2006), 13–24.

[5] Galil Z., G. F. Italiano. Reducing Edge Connectivity to Vertex Connectivity. *ACM SIGACT News*, **22** (1991), No 1, 57–61.

[6] Hopcroft J. E., R. E. Tarjan. Dividing a Graph into Triconnected Components. *SIAM J. Comput.*, **2** (1973), No 3, 135–158.

[7] Nagamochi H., T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan J. Indust. Appl. Math.*, **9** (1992), 163–180.

[8] La Poutré J. A., J. van Leeuwen, M. H. Overmars. Maintenance of 2- and 3-edge-connected components of graphs I. *Discrete Mathematics*, **114** (1993), No 1–3, 329–359.

[9] Pritchard D. Fast Distributed Computation of Cuts Via Random Circulations. In: L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, I. Walukiewicz (eds). Automata, Languages and Programming. ICALP 2008. *Lecture Notes in Computer Science*, **5125** (2008), 145–160.

[10] Taoka S., T. Watanabe, K. Onaga. A Linear-Time Algorithm for Computing All 3-Edge-Connected Components of a Multigraph. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, **E75-A** (1992), No 3, 410–424.

Appendix A. Here we briefly describe the method from [9] for generating labels for the edges of a 2-edge connected graph such that, for any two edges e_1 and e_2, $label(e_1) = label(e_2) \iff e_1$ and e_2 form a cut pair.

A circulation is a network flow of zero value [1]. It can be defined over any field. For us it is convenient to work with the remainders modulo 2. They naturally correspond to bits and addition corresponds to XOR. Pick two edges e_1 and e_2 which form a cut pair, as in Fig. 6. Let us call A and B the two parts in which the graph splits if we remove e_1 and e_2. In any circulation $flow(e_1) = -flow(e_2)$, because these edges are the only ones connecting A with B and in a circulation any flow going from A to B should eventually return again to A. We work with

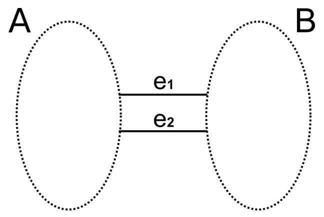

Fig. 6. $\{e_1, e_2\}$ is a cut pair separating the subgraphs A and B

remainders modulo 2, for which it holds that $x = -x$, so $flow(e_2) = -flow(e_2)$. This means $flow(e_1) = flow(e_2)$. We have proven that if two edges form a cut pair, the flow through them in any circulation is equal. Assuming the edges do not form a cut pair, in [9] it is shown that the probability for the flow through them to be equal is $\frac{1}{2}$. If we concatenate the flows through an edge in L random circulations, we get a L-bit label for the edge, such that, with high probability, two edges form a cut pair iff their labels are equal (the probability of error is $\frac{1}{2^L}$). Setting $L = 64$ should be enough for most situations.

Algorithm 3 Initialization of edge labels (for clarity, we denote them as flow)

```
 1: visited ← [NO for vr ∈ V]
 2:
 3: function InitEdgeLabels(vr, parent ← None, label_length)
 4:     visited[vr] ← ON_PATH
 5:     flow_at_vertex ← 0
 6:     for all (vr, u) ∈ E, u ≠ parent do
 7:         if visited[u] = ON_PATH then
 8:             flow[(vr, u)] ← RandomBitStringOfLength(label_length)
 9:         else
10:             if visited[u] = NO then
11:                 InitEdgeLabels(u, vr, label_length)
12:         flow_at_edge ← flow_at_edge ⊕ flow[(vr, u)]
13:     if parent ≠ None then
14:         flow[(parent, vr)] ← flow_at_edge
15:     visited[vr] ← YES
```

What is left is to show an easy way to build random circulations. This can be accomplished by modified depth-first search. Intuitively, if we fix the DFS-tree of a graph, the back edges act as “basis” for the space of circulations. We can randomly set the flow through the back edges and this will uniquely determine the flow through the tree edges. Algorithm 3 illustrates this procedure in pseudocode. The random circulations it creates are used to initialize the edge labels.

As a side note, if the input graph is not 2-edge connected, then, with high probability, edges with zero label are bridges (and do not form cut pairs with each other). If we know the bridges in the graph, partitioning it into 2-edge connected components can be done using coloring operations. In this way, we don't need an additional algorithm for splitting the graph into 2-edge connected components, before finding the 3-edge connected components.

Vladislav Haralampiev
Faculty of Mathematics and Informatics
St Kliment Ohridski University of Sofia
5, James Bourchier Blvd
1164 Sofia, Bulgaria
e-mail: `vladislav.haralampiev@gmail.com`

Received 23 April 2018
Final Accepted 25 July 2018