Information Processing Letters 114 (2014) 50–55

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A simple 3-edge connected component algorithm revisited

Nima Norouzi [a], Yung H. Tsin [b,*,1]

[a] Computer Sciences Corporation, 32605 West 12 Mile Road, Farmington Hills, MI, USA
[b] School of Computer Science, University of Windsor, Windsor, Ontario, Canada

ARTICLE INFO

Article history:
Received 24 November 2012
Received in revised form 31 May 2013
Accepted 18 September 2013
Available online 25 September 2013
Communicated by Tsan-sheng Hsu

Keywords:
3-edge-connectivity
3-edge-connected components
Graph algorithms

ABSTRACT

Graph connectivity is a graph-theoretic concept that is fundamental to the studies of many applications such as network reliability and network decomposition. For the 3-edge-connectivity problem, recently, it has been shown to be useful in a variety of apparently unrelated areas such as solving the G-irreducibility of Feynman diagram in physics and quantum chemistry, editing cluster and aligning genome in bioinformatics, placing monitors on the edges of a network in flow networks, spare capacity allocation and decomposing a social network to study its community structure. A number of linear-time algorithms for 3-edge-connectivity have thus been proposed. Of all these algorithms, the algorithm of Tsin is conceptually the simplest and also runs efficiently in a recent study. In this article, we shall show how to simplify the implementation of a key step in the algorithm making the algorithm much more easier to implement and run more efficiently. The simplification eliminates a rather complicated linked-lists structure and reduces the space requirement of that step from $O(|E|)$ to $O(|V|)$, where V and E are the vertex set and the edge set of the input graph, respectively.

1. Introduction

Graph connectivity is a graph-theoretic concept that is fundamental to the studies of many applications such as network reliability and network decomposition. For the 3-edge-connectivity problem, recently, it has been shown to be useful in a variety of apparently unrelated areas such as solving the G-irreducibility of Feynman diagram in physics and quantum chemistry [3,4], editing cluster and aligning genome in bioinformatics [5,12–14], placing monitors on the edges of a network in flow networks [1], spare capacity allocation [8] and decomposing a social network to study its community structure [2]. Owing to this reason, a number of linear-time algorithms for 3-edge-connectivity have been proposed [7,10,15,17,18]. It is of practical interest to investigate which of these algorithms should be used in real-life applications. A study reported in [18] shows that [17] is a strong candidate. Since ease of implementation is a factor taken seriously by application programmers, in this article, we shall address an implementation issue of the algorithm by showing how to improve the implementation of one of its key steps originally described in [17]. The improvement results in eliminating the usage of a linked list whereby reducing the space requirement of that step from $O(|E|)$ to $O(|V|)$ (V and E are the vertex set and edge set of the input graph, respectively) and making the coding of the algorithm considerably easier. We have actually implemented the algorithm; the code is available at [11]. Recently, Mehlhorn, Neumann and Schmidt [9] presented a linear-time certifying algorithm for testing 3-edge-connectivity. Their algorithm returns a short certificate as a proof if the input graph is 3-edge-connected. If the input graph is not 3-edge-connected, the algorithm returns one cut-pair as a proof; it does not generate the 3-edge-connected components or a corresponding set of cut-pairs. By contrast, the existing algorithms [7,10,15,17,18] are non-certifying but generate all the 3-edge-connected components.

* Corresponding author.
E-mail addresses: nima@norouzi.net (N. Norouzi), peter@uwindsor.ca (Y.H. Tsin).
[1] Research partially supported by NSERC under grant NSERC-781103.

0020-0190/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.ipl.2013.09.010

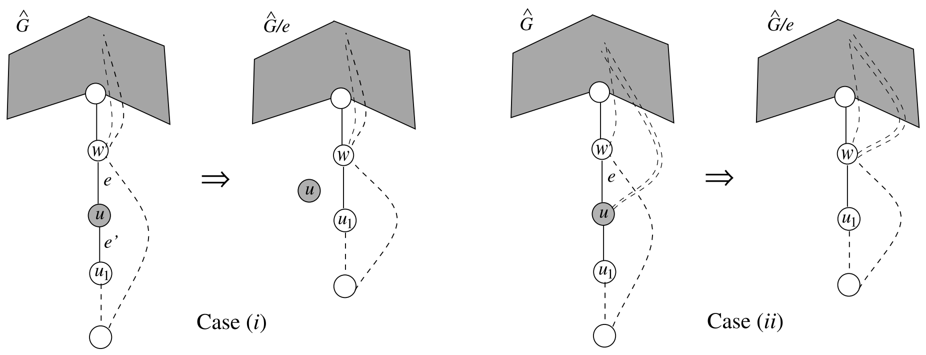

Fig. 1. Absorb–eject operation.

2. Some basic definitions

The graph-theoretic concepts used in this paper can be found in standard textbooks such as [6]. The definitions given in this section are some of the more basic and important ones.

The input graph is denoted by $G = (V, E)$, where V is the vertex set and E is the edge set. An edge e in E is represented by $e = (x, y)$, where x and y are the end-vertices. The graph G may contain parallel edges but no self-loops as self-loop has no impact on the connectivity of a graph. If $E = \emptyset$, the graph is a *null* graph. The *degree* of a vertex u in G, denoted by $deg_G(u)$, is the number of edges incident on u in G.

A $u - v$ *path* is a path connecting the vertices u and v in G. A graph $G = (V, E)$ is *connected* if $\forall u, v \in V$, there is a $u - v$ path in it. It is *disconnected* otherwise. Let G be a connected graph. An edge is a *bridge* in G if removing it from G results in a disconnected graph. The graph G is *2-edge-connected* if it has no bridge. A *cut-pair* of G is a pair of edges whose removal results in a disconnected graph and neither is a bridge. A *cut-edge* is an edge in a cut-pair. The graph G is *3-edge-connected* if it has neither a bridge nor a cut-pair. A *3-edge-connected component* of G is a maximal 3-edge-connected subgraph of G.

Depth-first search (henceforth, abbreviated as DFS) augmented with vertex labeling is a graph traversal technique first introduced by Tarjan [16]. When a DFS is performed over a graph, each vertex w is assigned a *depth-first number*, $dfs(w)$, such that $dfs(w) = k$ if vertex w is the kth vertex visited by the search for the first time. The search also partitions the edge set into two types of edges, *tree-edge* and *back-edge*. An edge $e = (u, v)$ is a tree-edge, denoted by $u \rightarrow v$, if $dfs(u) < dfs(v)$ and back-edge, denoted by $u \hookrightarrow v$, if $dfs(v) < dfs(u)$. In the former case, u is the *parent* of v while v is a *child* of u. In the latter case, u is the *tail* while v is the *head* and the back-edge is an *incoming back-edge* of v and an *outgoing back-edge* of u.

The tree-edges form a spanning tree of G, denoted by T, rooted at the vertex r from which the search begins. A vertex u is an *ancestor* of vertex v (vertex v is a *descendant* of vertex u) if u is a vertex on the $r - v$ path in T. A $u - v$ path in T is a *tree-path* if u is an ancestor of v. The *subtree* of T *rooted at vertex* w, denoted by T_w, is the subtree of T containing all the descendants of w.

$\forall w \in V$, $lowpt(w) = dfs(z)$, where $dfs(z)$ is the smallest among the dfs numbers of all the vertices that can be reached from w via a (possibly null) $w - v$ tree-path following by a back-edge $v \hookrightarrow z$.

3. The simplified implementation

First, we shall briefly explain the key idea underlying the algorithm in [17].

Beginning with the input graph $G = (V, E)$, the graph is gradually transformed so that vertices that are confirmed to be belonging to the same 3-edge-connected component are merged into one vertex, called a *supervertex*. Each supervertex is represented by a vertex $v \in V$ and a set $\sigma(v)$ consisting of a subset of vertices that have been confirmed to be belonging to the same 3-edge-connected component as v. Initially, each vertex v is regarded as a supervertex with $\sigma(v) = \{v\}$. When two supervertices w and u are merged, one of them, say w, absorbs the other resulting in $\sigma(w) = \sigma(w) \cup \sigma(u)$. When a supervertex containing all the vertices of a 3-edge-connected component is formed, it is separated from the graph becoming an isolated vertex. At the end, the graph is transformed into a collection of isolated supervertices each of which contains the vertices of a distinct 3-edge-connected component of G.

The graph is transformed by the *absorb–eject* operation. Let G be transformed to $\widehat{G}$ when the operation is applied to an edge $e = (w, u)$,

(i) If $deg_{\widehat{G}}(u) = 2$, let $e' = (u, u_1)$ be the other edge incident on u. Then $\{e, e'\}$ is a cut-pair. So, e and e' are replaced by a new edge (w, u_1) and u becomes an isolated supervertex (Fig. 1(i)).

(ii) If $deg_{\widehat{G}}(u) \neq 2$ and it is confirmed that e cannot be a cut-edge, then vertex w adsorbs vertex u as they must belong to the same 3-edge-connected component. As a consequence, the edges incident on u become edges incident on w (Fig. 1(ii)) and $\sigma(w) = \sigma(w) \cup \sigma(u)$.

To carry out the transformation, the algorithm performs a depth-first search over G starting from an arbitrary vertex r. During the depth-first search, whenever the search backtracks from a vertex u to a vertex w, the subgraph of G induced by the vertex set of T_u has been transformed into a set of isolated supervertices and a $u - u_k$ tree-path, $u - u_1 - u_2 - \cdots - u_k$ (Fig. 2). Each isolated supervertex v