Information Processing Letters 116 (2016) 47–52

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A space-efficient algorithm for finding strongly connected components

David J. Pearce

School of Engineering and Computer Science, Victoria University, New Zealand

ARTICLE INFO

Article history:
Received 19 March 2015
Received in revised form 25 August 2015
Accepted 31 August 2015
Available online 9 September 2015
Communicated by Ł. Kowalik

Keywords:
Graph algorithms
Strongly connected components
Depth-First Search

ABSTRACT

Tarjan's algorithm for finding the strongly connected components of a directed graph is widely used and acclaimed. His original algorithm required at most $v(2+5w)$ bits of storage, where w is the machine's word size, whilst Nuutila and Soisalon-Soininen reduced this to $v(1+4w)$. Many real world applications routinely operate on very large graphs where the storage requirements of such algorithms is a concern. We present a novel improvement on Tarjan's algorithm which reduces the space requirements to $v(1+3w)$ bits in the worst case. Furthermore, our algorithm has been independently integrated into the widely-used SciPy library for scientific computing.

1. Introduction

For a directed graph $D=(V,E)$, a *Strongly Connected Component (SCC)* is a maximal induced subgraph $S=(V_S,E_S)$ where, for every $x,y\in V_S$, there is a path from x to y (and vice-versa). Tarjan presented a now well-established algorithm for computing the strongly connected components of a digraph in time $\Theta(v+e)$ [14]. In the worst case, this needs $v(2+5w)$ bits of storage, where w is the machine's word size. Nuutila and Soisalon-Soininen reduced this to $v(1+4w)$ [10]. In this paper, we present for the first time an algorithm requiring only $v(1+3w)$ bits in the worst case. Furthermore, this algorithm has been independently integrated into the widely-used SciPy library for scientific computing specifically because of its ability to handle larger graphs in practice [13].

Tarjan's algorithm has found numerous uses in the literature, often as a subcomponent of larger algorithms, such as those for *transitive closure* [9], *compiler optimisation* [5], *program analysis* [1,11] and for *bisimulation equivalence* [2] to name but a few. Of particular relevance is its use in model checking [7,12], where the algorithm's storage requirements are a critical factor limiting the number of states which can be explored [8,4].

2. Depth-First Search

Algorithm 1 presents a well-known procedure for traversing digraphs, known as Depth First Search (DFS). We say that an edge $v\to w$ is *traversed* if visit(w) is called from visit(v) and that the value of *index* on entry to visit(v) is the *visitation index* of v. Furthermore, when visit(w) returns we say the algorithm is *backtracking* from w to v. The algorithm works by traversing along some branch until a leaf or a previously visited vertex is reached; then, it *backtracks* to the most recently visited vertex with an unexplored edge and proceeds along this; when there is no such vertex, one is chosen from the set of unvisited vertices and this continues until the whole digraph has been explored. Such a traversal always corresponds to a series of disjoint trees, called *traversal trees*, which span the digraph. Taken together, these are referred to as a *traversal forest*. Fig. 1 provides some example traversal forests.

Formally, $F=(I,T_0,\ldots,T_n)$ denotes a traversal forest over a digraph $D=(V,E)$. Here, I maps every vertex to its visitation index and each T_i is a traversal tree given by $(r,V_{T_i}\subseteq V,E_{T_i}\subseteq E)$, where r is its root. It is easy to see

E-mail address: david.pearce@ecs.vuw.ac.nz.

http://dx.doi.org/10.1016/j.ipl.2015.08.010
0020-0190/

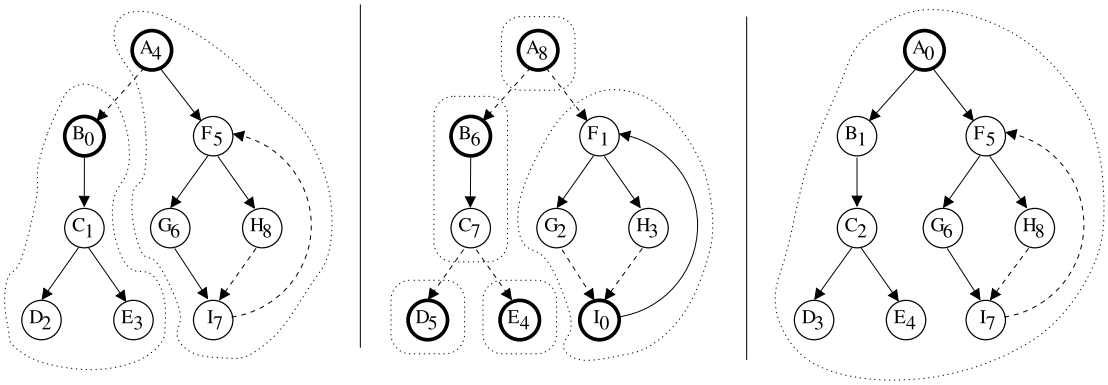

Fig. 1. Illustrating three possible traversal forests for the same graph. The key is as follows: vertices are subscripted with their visitation index; dotted lines separate traversal trees; dashed edges indicate those edges not traversed; finally, bold vertices are tree roots.

Algorithm 1 DFS(V,E).

```
1: index = 0
2: for all v ∈ V do visited[v] = false
3: for all v ∈ V do
4:     if ¬visited[v] then visit(v)

procedure visit(v)
5: visited[v] = true ; index = index + 1
6: for all v → w ∈ E do
7:     if ¬visited[w] then visit(w)
```

that, if visit(x) is called from the outer loop, then x is the root of a traversal tree. For a traversal forest F, those edges making up its traversal trees are *tree-edges*, whilst the remainder are *non-tree* edges. Non-tree edges can be further subdivided into *forward-*, *back-* and *cross-edges*:

Definition 1. For a directed graph, $D=(V,E)$, a node x *reaches* a node y, written $x \overset{D}{\rightsquigarrow} y$, if $x=y$ or $\exists z.[x \rightarrow z \in E \wedge z \overset{D}{\rightsquigarrow} y]$. The D is often omitted from $\overset{D}{\rightsquigarrow}$, when it is clear from the context.

Definition 2. For a digraph $D=(V,E)$, an edge $x \rightarrow y \in E$ is a forward-edge, with respect to some tree $T=(r,V_T,E_T)$, if $x \rightarrow y \notin E_T \wedge x \neq y \wedge x \overset{T}{\rightsquigarrow} y$.

Definition 3. For a digraph $D=(V,E)$, an edge $x \rightarrow y \in E$ is a back-edge, with respect to some tree $T=(r,V_T,E_T)$, if $x \rightarrow y \notin E_T \wedge y \overset{T}{\rightsquigarrow} x$.

Cross-edges constitute those which are neither forward- nor back-edges. A few simple observations can be made about these edge types: firstly, if $x \rightarrow y$ is a forward-edge, then $I(x) < I(y)$; secondly, cross-edges may be *intra-tree* (i.e. connecting vertices in the same tree) or *inter-tree*; thirdly, for a back-edge $x \rightarrow y$ (note, Tarjan called these *fronds*), it holds that $I(x) \geq I(y)$ and all vertices on a path from y to x are part of the same strongly connected component. In fact, it can also be shown that $I(x) > I(y)$ always holds for a cross-edge $x \rightarrow y$ (see Lemma 1, page 51).

Two fundamental concepts behind efficient algorithms for this problem are the *local root* (note, Tarjan called these LOWLINK values) and *component root*: the local root of v is the vertex with the lowest visitation index of any in the same component reachable by a path from v involving at most one back-edge; the root of a component is the member with lowest visitation index. The significance of local roots is that they can be computed efficiently and that, if r is the local root of v, then $r=v$ iff v is the root of a component (see Lemma 3, page 52). Thus, local roots can be used to identify component roots.

Another important topic, at least from the point of view of this paper, is the additional storage requirements of Algorithm 1 over that of the underlying graph data structure. Certainly, v bits are needed for *visited*[·], where $v=|V|$. Furthermore, each activation record for visit(·) holds the value of v, as well as the current position in v's out-edge set. The latter is needed to ensure each edge is iterated at most once. Since no vertex can be visited twice, the call-stack can be at most v vertices deep and, hence, consumes at most $2vw$ bits of storage, where w is the machine's word size. Note, while each activation record may hold more items in practice (e.g. the return address), these can be avoided by using a *non-recursive* implementation (see Section 4). Thus, Algorithm 1 requires at most $v(1+2w)$ bits of storage. Note, we have ignored *index* here, since we are concerned only with storage proportional to $|V|$.

3. Improved algorithm for finding strongly connected components

Tarjan's algorithm and its variants are based upon Algorithm 1 and the ideas laid out in the previous section. Given a directed graph $D=(V,E)$, the objective is to compute an array mapping vertices to component identifiers, such that v and w map to the same identifier iff they are members of the same component. Tarjan was the first to show this could be done in $\Theta(v+e)$ time, where $v=|V|$ and $e=|E|$. Tarjan's algorithm uses the *backtracking* phase of Depth-First Search to explicitly compute the local root of each vertex. An array of size $|V|$, mapping each vertex to its local root, stores this information. Another array of size $|V|$ is needed to map vertices to their visitation index. Thus, these two arrays consume $2vw$ bits of storage between them.

The key insight behind our improvement is that these arrays can, in fact, be combined into one. This array,