AN OPTIMAL DISTRIBUTED BRIDGE-FINDING ALGORITHM

DAVID PRITCHARD, UNIVERSITY OF WATERLOO, DAVEAGP@GMAIL.COM

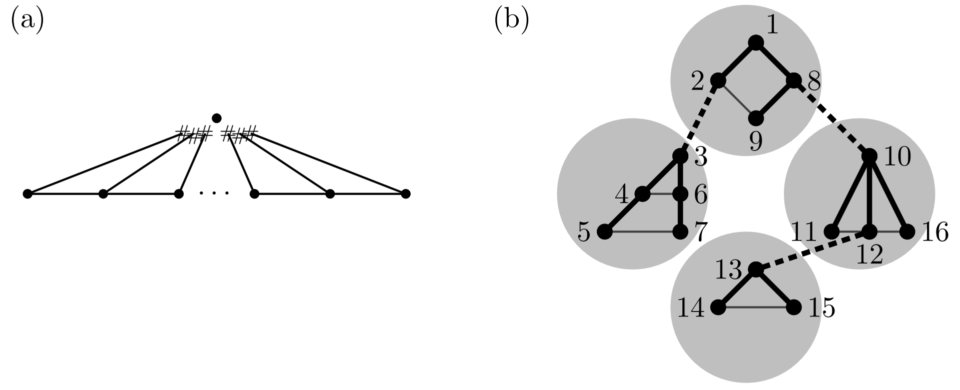
ABSTRACT. In this paper we model a computer network by a graph $\mathcal{G} = (V, E)$. We assume that $\mathcal{G}$ is connected, so that all computers can communicate with one another. A *bridge* is an edge e of $\mathcal{G}$ such that $\mathcal{G}\backslash\{e\}$ is not connected. We are interested in efficiently determining all of the bridges of a distributed system. We give a new distributed algorithm for bridge-finding, using messages of size $O(\log |V|)$. The algorithm is simpler, faster, and more communication-efficient than all previous algorithms. Under suitable assumptions, the algorithm has optimal complexity on *all* graphs.

1. Introduction

A *bridge* is an edge e of $\mathcal{G}$ such that $\mathcal{G}\backslash\{e\}$ is not connected. Let $\mathcal{H}$ be a maximal subgraph of $\mathcal{G}$ such that $\mathcal{H}$ remains connected despite the deletion of any one node. The edges of $\mathcal{H}$ are said to be a *block* of $\mathcal{G}$. The blocks of $\mathcal{G}$ form a partition of its edges [4]. All known distributed bridge-finding algorithms first compute the blocks of $\mathcal{G}$, and then identify the bridges by using the fact that e is a bridge if and only if $\{e\}$ is a block. Tarjan and Vishkin [6] gave a sequential method for computing the blocks of a graph. Namely, they construct a certain graph $\mathcal{G}'$ whose vertices are the *edges* of $\mathcal{G}$, and such that the blocks of $\mathcal{G}$ are precisely the connected components of $\mathcal{G}'$.

Huang [1] gave a distributed block-finding algorithm based off of the method of Tarjan and Vishkin. First we determine a spanning tree of $\mathcal{G}$. Using the tree, we compute $\mathcal{G}'$ distributively; then we use a (naïve) connected components algorithm on $\mathcal{G}'$. The resulting algorithm has time complexity $O(|V|)$. Thurimella [7] proved that on graphs with small diameter, one could do better than Huang's algorithm. The *diameter* Diam of $\mathcal{G}$ is the largest distance between any two nodes of $\mathcal{G}$. Thurimella gave a distributed algorithm that computes the blocks in $O(\mathsf{Diam} + \sqrt{|V|}\log^* |V|)$ time. In contrast, as shown in Figure 1(a), even when $\mathsf{Diam}(\mathcal{G}) = 2$, we may have $\mathsf{Diam}(\mathcal{G}') = |V| - 2$ and in that case Huang's algorithm takes $\Omega(|V|)$ time. Thurimella's algorithm uses a *scan-first* spanning tree. Thurimella's insight was twofold. First, he showed that it suffices to consider the connected components of a certain subgraph of $\mathcal{G}$, rather than treating edges as vertices. Second, he gave an efficient distributed algorithm for computing the connected components of this subgraph.

Figure 1. Left: A graph $\mathcal{G}$ where Huang's algorithm takes a long time. Right: A network. The biconnected components are inside light gray discs, tree edges are thick lines, and bridges are thick dashed lines. A preordering is shown.



The point of this paper is that if we want the bridges, it is wasteful to compute the blocks. Our bridge-finding technique goes back to Tarjan's 1974 paper [5] and the same ideas are used in [1, 7]. The important (and hitherto unrealized) fact is that we can implement this technique very efficiently in a distributed network. We thereby improve upon Thurimella's running time, obtaining $O(\mathsf{Diam})$ time complexity for bridge-finding.

2. Technique for Bridge-Finding

As a general note: we omit proofs of some claims, but the proofs are not difficult. See the references for more information. We say that two nodes are in the same *biconnected component* if they remain pairwise connected despite the deletion of any single edge. The biconnected components of $\mathcal{G}$ are exactly the connected components of $(\mathcal{G}\backslash bridges(\mathcal{G}))$. Figure 1(b) shows the bridges and biconnected components of a graph. Our algorithm uses a spanning tree $\mathcal{T}$ of $\mathcal{G}$ which is also shown. Any spanning tree will do, although a BFS tree leads to the best running time.

Since simple u-v paths in $G\backslash\{\{u, v\}\}$ correspond to simple $\{u, v\}$-containing cycles in $\mathcal{G}$, we have

(1) An edge is a bridge of $\mathcal{G}$ if and only if it is contained in no simple cycle of $\mathcal{G}$.

Then it follows that every bridge is a tree edge. Let $desc(v)$ denote the descendants of v in $\mathcal{T}$, including v itself. Let $p(v)$ denote the parent of v in $\mathcal{T}$; thus the edges of $\mathcal{T}$ are exactly those edges of the form $\{p(v), v\}$. From property (1) we can deduce that

(2) $\{p(v), v\}$ is a bridge if and only if no other edge meets both $desc(v)$ and $V(\mathcal{G})\backslash desc(v)$.

Finally, we would like to efficiently determine for each node v whether $\{p(v), v\}$ meets condition (2). The following method for this task can be derived from [4] and [5]. Compute a preorder of the vertices with respect to $\mathcal{T}$. Hereafter we will refer to nodes by their preorder label — this is important! In particular $p(v) < v$ for all non-root v. For a node v we define its *subtree-neighbourhood* $SN(v)$, its *low subtree-neighbour* $low(v)$, and its *high subtree-neighbour* $high(v)$ by

$$SN(v) := desc(v) \cup \{w \mid u \in desc(v), \{w, u\} \in E(\mathcal{G}\backslash\mathcal{T})\},$$

$$low(v) := \min SN(v) \qquad \text{and} \qquad high(v) := \max SN(v).$$

Then from property (2), and since in preorder $desc(v) = \{v, \ldots, v + \#desc(v) - 1\}$, we have

(3) Edge $\{p(v), v\}$ is a bridge if and only if $(low(v) \geq v)$ and $(high(v) < v + \#desc(v))$.

So to identify the bridges it suffices to determine the nodes v at which (3) holds.

3. Implementation

We assume that the network is synchronous, and that initially there is a single distinguished *leader* node. We make use of the parallel communication techniques *convergecasting* and *downcasting* as described in [2]. Essentially, this means that we efficiently communicate up and down all branches of the tree at the same time. First we use a well-known greedy algorithm for distributively computing a BFS tree [2]. This algorithm has time complexity $O(\mathsf{Diam})$ and uses $O(|E|)$ messages.

Next, we downcast a request from the root to ask each node v to compute $\#desc(v)$. Using a convergecast, each node will report its $\#desc$ value back to its parent. Each leaf node v immediately determines that $\#desc(v) = 1$ and reports this value to its parent; each non-leaf node v, upon learning the $\#desc$ values of its children $c_1, \ldots, c_k$, computes $\#desc(v) = 1 + \sum_{i=1}^{k} \#desc(c_i)$ and reports this value to its parent.

The preorder computation is similar. The root sets its preorder label to 1. Whenever a node v sets preorder label to ℓ, it orders its children in $\mathcal{T}$ arbitrarily as $c_1, c_2, \ldots$. Then v sends the message "Set your preorder label to ℓ_i" to each c_i, where ℓ_i is computed by v as $\ell_i = \ell + 1 + \sum_{j<i} \#desc(c_j)$.

Here is how we compute *low* and *high*. Each node v initializes $\mathit{low}(v) := \mathit{high}(v) := v$. Then, each node announces its preorder label to all of its neighbours except its parent and children. When node v hears such an announcement from u it sets $\mathit{low}(v) := \min(\mathit{low}(v), u)$ and $\mathit{high}(v) := \max(\mathit{high}(v), u)$. Finally, using a convergecast, from the leaves up to the root, each node computes

$$\mathit{low}(v) := \min\bigl(\{\mathit{low}(v)\} \cup \{\mathit{low}(u) \mid p(u) = v\}\bigr); \quad \mathit{high}(v) := \max\bigl(\{\mathit{high}(v)\} \cup \{\mathit{high}(u) \mid p(u) = v\}\bigr).$$

The announcement along the non-tree edges uses $O(|E|)$ messages and takes $O(1)$ time. Each of the tree-based steps — computing $\#\mathit{desc}$, preordering, computing *low* and *high* — takes $O(h(\mathcal{T}))$ time and uses $O(|V|)$ messages, where $h(\mathcal{T})$ is the height of $\mathcal{T}$. But the height of a BFS tree satisfies $\mathsf{Diam}/2 \le h(\mathcal{T}) \le \mathsf{Diam}$; thus the entire algorithm, from start to finish, takes $O(\mathsf{Diam})$ time and sends $O(|E|)$ messages.

We also note that in $O(|V|)$ more messages and $O(h(\mathcal{T}))$ more time we can label the nodes by their biconnected components. The key is that each biconnected component induces a connected subgraph of $\mathcal{T}$. Node v is the "top" of its component if if either v is the root or $\{p(v), v\}$ is a bridge, and this is easy to compute. We make each such node send its label down the non-bridge tree edges, to be copied by the other nodes in its component. See [3] for more information.

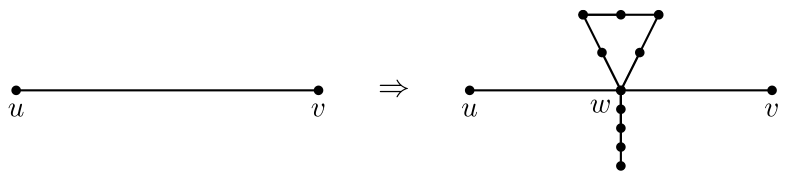
4. Optimality

Consider any *event-driven* algorithm with a *single initiator* where the identities of neighbours are initially unknown (no *neighbour knowledge*). We argue that under this model, any correct bridge-finding algorithm sends at least $|E|$ messages and takes at least $\mathsf{Diam}/2$ time on *all* graphs.

Suppose a bridge-finding protocol, when it is run on $\mathcal{G}$, doesn't ever send a message on edge $\{u, v\}$. Let $\mathcal{G}'$ be a graph obtained from $\mathcal{G}$ by subdividing (u, v) with a new node w, as shown in Figure 2. We also attach some cycles and bridges to the new node. When we run the protocol on $\mathcal{G}'$, because the algorithm is event-driven, no messages are sent within the new portions of the graph. Thus the algorithm could not possibly correctly determine which of the new edges are bridges.

Since every edge has at least one message sent along it, the lower message bound follows. The lower time bound is essentially the same: if an algorithm takes less than $\mathsf{Diam}(\mathcal{G})/2$ steps on some

FIGURE 2. Proof of message lower bound: we modify $\mathcal{G}$ into $\mathcal{G}'$.



graph $\mathcal{G}$, then there are parts of the graph which no messages reach, and we can modify those parts. Thus no algorithm can beat the one described above by more than a constant factor on *any* graph.

5. Closing

Although we have improved the distributed complexity of bridge-finding, it remains to improve upon Thurimella's algorithm for the more general problem of computing the blocks. The intermediate problem of computing *articulation points* — nodes v such that $\mathcal{G}\backslash\{v\}$ is not connected — does not seem to yield to our approach. It would be good either to find an $O(\mathsf{Diam})$ time algorithm for articulation points, or show that this is impossible in some model of distributed computation.

The given algorithm can be easily transformed into a *strong components* algorithm, if we use a depth-first search tree instead of a BFS. If there were a distributed DFS algorithm with time complexity $(O(\mathsf{Diam}) + o(n))$, then it seems we could improve upon the fastest-known distributed strong components algorithm. It would (similarly) be good either to find such a fast DFS algorithm, or else show that this is impossible.

References

[1] S. T. Huang. A new distributed algorithm for the biconnectivity problem. In *Proc. 1989 International Conf. Parallel Processing*, pages 106–113, 1989.

[2] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM, 2000.

[3] D. Pritchard. Robust network computation. Master's thesis, MIT, 2005.

[4] R. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[5] R. E. Tarjan. A note on finding the bridges of a graph. *Inform. Process. Lett.*, 2:160–161, 1974.

[6] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[7] R. Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. *J. Algorithms*, 23(1):160–179, 1997.