An Optimal Distributed Bridge-Finding Algorithm

David Pritchard, Department of Combinatorics and Optimization
University of Waterloo, Ontario, Canada

Overview and Problem Statement

We model a computer network by a graph, where the vertices are computers and the edges are two-way communication links. It is a fundamental necessity that the graph be connected if the network is to operate as a whole. A *bridge* is an edge whose deletion causes the network to become disconnected. The bridges are the edges that are critical to network reliability. We give a simple, efficient distributed algorithm to determine the bridges of a network. Let the network be $\mathcal{G} = (V, E)$, and let $\mathsf{Diam}(\mathcal{G})$ be the largest distance between any pair of nodes in $\mathcal{G}$. Our protocol

- Assumes the network is synchronous.
- Assumes the existence of a distinguished *leader* node.
- Uses messages that are $O(\log V)$ bits long.
- Sends a total of $O(E)$ messages.
- Completes in $O(\mathsf{Diam}(\mathcal{G}))$ time.
- Has optimal message and time complexity (within a constant factor) on all graphs, under certain assumptions.

Background

Historically, algorithms for bridge-finding are linked to ones for determining the *cut points* — those nodes whose deletion disconnects the network — and *blocks* — maximal subgraphs having no cut vertices. The blocks partition $E(\mathcal{G})$, and from this partition, it is easy to determine the cut points and bridges. Note, we consider only small-message protocols since we could otherwise use a trivial Diam-time algorithm. This also aids our algorithm's practicality.

Table: A brief history of bridge-finding and block-finding algorithms.

When	Who	What	Model	Complexity
1972	Tarjan	Blocks	Sequential	$O(V+E)$ time
◇ Based off of properties of depth-first search				
1974	Tarjan	Bridges	Sequential	$O(V+E)$ time
◇ Extends older ideas to use any spanning tree, not just DFS				
1984	Tarjan-Vishkin	Blocks	Seq., Parallel	$O(V+E)$ time
◇ Works with any spanning tree. Blocks are connected components of an auxiliary graph $\mathcal{G}'$ with $V(\mathcal{G}') = E(\mathcal{G})$				
1989	Huang	Blocks	Distributed	$O(V)$ time, $O(E)$ msg
◇ Reformulation of Tarjan-Vishkin				
1990	Hohberg	Blocks	Distributed	$O(E + V\log V)$ msg
◇ Proved $\Omega(E + V\log V)$ msgs are necessary if no leader present				
1997	Thurimella	Blocks	Distributed	$\tilde{O}(\mathsf{Diam} + \sqrt{V})$ time
◇ Improves Huang/T-V. Uses subgraph $\mathcal{H} \subseteq \mathcal{G}$ and not auxiliary graph ◇ Calls fast MST algorithm to get connected components of $\mathcal{H}$				
2006	[this poster]	Bridges	Distributed	$O(\mathsf{Diam})$ time, $O(E)$ msg

Our Contribution

Thurimella's block-finding algorithm seems also to be the fastest known distributed bridge-finding algorithm. The main point of this work is that, if we only want to compute the bridges, then it is wasteful to compute the blocks. Specifically, Thurimella's algorithm takes $\Theta(\mathsf{Diam}+\sqrt{V}\log^* V)$ time, but ours takes only $O(\mathsf{Diam})$ time to find the bridges, and this seems to be optimal. Our algorithm uses the same key ideas as Tarjan's 1974 paper.

The author thanks the University of Waterloo Graduate Studies Office for their financial support, and thanks the poster referees for several helpful suggestions.

Bridge-Finding Technique

Step 1: Find a spanning tree. Our algorithm requires a spanning tree $\mathcal{T}$ of the network. Any spanning tree will do, but a BFS tree leads to the best running time. Since each cycle of $\mathcal{G}$ containing edge $\{u, v\}$ corresponds to a simple u-v path in $G\backslash\{\{u, v\}\}$, we have

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

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 $\mathcal{T}$. From property (1) we can deduce that

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

Step 2: Compute subtree sizes and preorder. We would like to efficiently determine for each node v whether $\{p(v), v\}$ meets condition (2). We can accomplish this with a *pre-order*, which is an *order of discovery* of a DFS on $\mathcal{T}$. **Important: hereafter we refer to all nodes by their preorder label.** So for example $p(v) < v$ for all non-root v.

Step 3: Compute low **and** $high$ **values.** For a node v we define its *subtree-neighbourhood* $SN(v)$ to comprise of the subtree rooted at v, along with any further nodes of $\mathcal{G}$ reachable from the subtree by a single non-tree edge:

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

For each node v the values $low(v)$ and $high(v)$ denote the minimum and maximum preorder label amongst its subtree-neighbourhood:

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

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

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

Our algorithm simply determines which v have property (3).

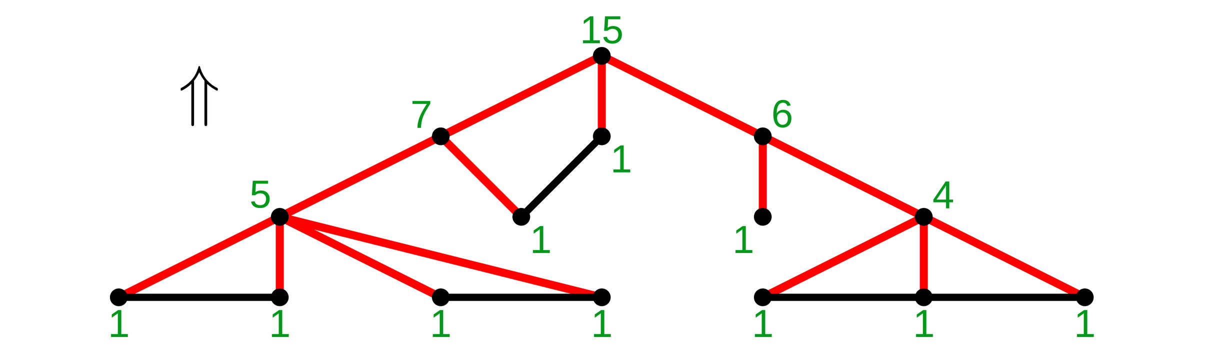
Implementation and Example

Step 1: We use a well-known greedy algorithm for distributively computing a BFS tree. That algorithm has time complexity $O(\mathsf{Diam})$ and uses $O(E)$ messages.

We repeatedly use two tree-based parallel communication techniques. In a *downcast*, messages are sent down each tree edge, starting at the root, and ending at the leaves. A *convergecast* is like a bottom-up acknowledgement for a downcast: node v waits for reports from each child, and then v reports to its parent.

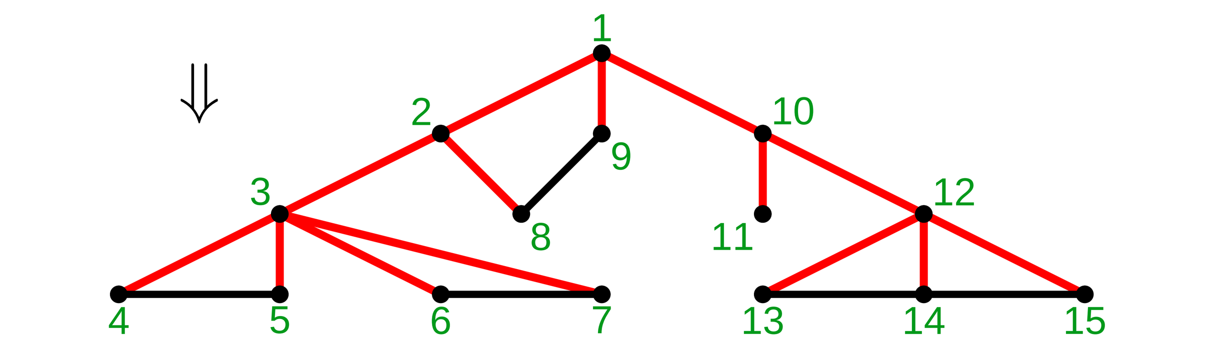
Step 2: Using a downcast, a request is sent that all nodes compute their subtree size $\#desc$. Then there corresponds a convergecast: each leaf node v immediately determines that $\#desc(v) = 1$ and reports this value to its parent; each non-leaf node v, upon learning the $\#desc$ values of its children $c_1, \dots, c_k$, computes $\#desc(v) = 1 + \sum_{i=1}^{k} \#desc(c_i)$, and reports this value to its parent. See Figure 1.

Figure 1: In this convergecast, each node v computes its subtree size $\#desc(v)$. Tree edges are red, with the root at the top.



Preordering is accomplished with a downcast. The root sets its preorder label to 1. Whenever a node v sets its preorder label to ℓ, it orders its children in $\mathcal{T}$ arbitrarily as $c_1, c_2, \dots$. Then v sends the message "Set your preorder label to ℓ_i" to each c_i, where v computes ℓ_i according to the formula $\ell_i = \ell + 1 + \sum_{j<i} \#desc(c_j)$. See Figure 2.

Figure 2: Using a downcast, each node computes its preorder label.

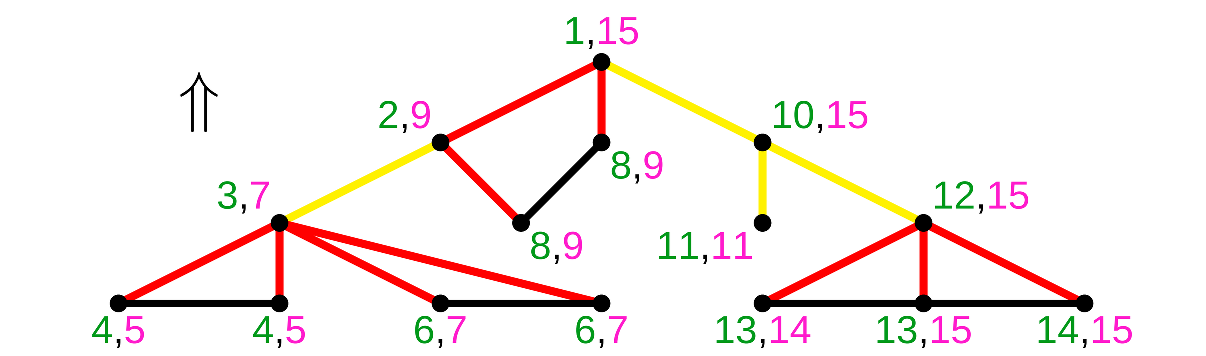


Step 3: Each node v initializes $low(v) \leftarrow v$ and $high(v) \leftarrow v$. Then, each node *announces* its preorder label to all of its neighbours except its parent and children. When node v hears such an announcement from u it sets $low(v) \leftarrow \min(low(v), u)$ and $high(v) \leftarrow \max(high(v), u)$. Using a convergecast, each node computes

$$low(v) \leftarrow \min\Big(\{low(v)\} \cup \{low(u) \mid u \text{ a child of } v\}\Big),$$
$$high(v) \leftarrow \max\Big(\{high(v)\} \cup \{high(u) \mid u \text{ a child of } v\}\Big).$$

See Figure 3. Finally, one additional message along each tree edge allows us to determine where property (3) holds.

Figure 3: With another convergecast, each node v computes $low(v)$ and $high(v)$. The bridges are shown in yellow.



We further note: if all nodes begin simultaneously, we can derive a near time-optimal *local* algorithm with implicit termination by using Elkin's *neighbourhood cover* protocol.

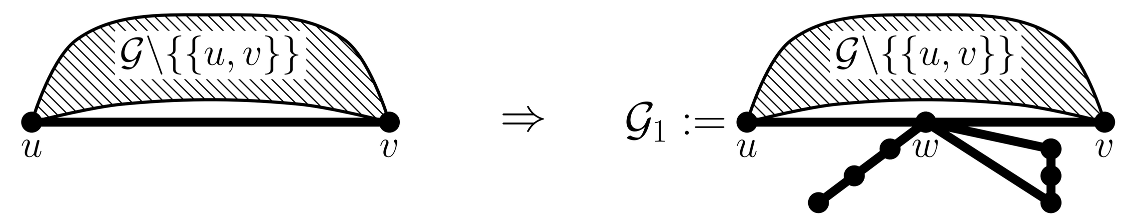
Complexity Analysis

Let h be the *height* of $\mathcal{T}$. Each downcast and convergecast takes h time and uses $V - 1$ messages. Since h is a BFS tree, $\frac{\mathsf{Diam}}{2} \leq h \leq \mathsf{Diam}$. The remaining operations — tree construction, announcement, and bridge identification — together take $O(\mathsf{Diam})$ time and $O(E)$ messages. Thus our algorithm's total complexity is $O(\mathsf{Diam})$ time and $O(E)$ messages. Each message has a single datum between 1 and V, and so can be encoded using $O(\log V)$ bits. Note, in an asynchronous environment, downcasts and convergecasts still take h time and $V - 1$ messages, but the complexity-dominating BFS tree construction step is more costly.

Universal Optimality

Our distributed algorithm is *deterministic, event-driven,* has a *single initiator,* and assumes neighbour identities are initially unknown (*no local knowledge*). Under these assumptions, *any* correct bridge-finding algorithm sends at least E messages and takes at least $\frac{\mathsf{Diam}}{2}$ time on *all* graphs.

Suppose that a bridge-finding protocol running on a graph $\mathcal{G}$ doesn't ever send a message on some edge, say $\{u, v\}$. Obtain graph $\mathcal{G}_1$ from $\mathcal{G}$ by subdividing $\{u, v\}$ with a new node w and attaching some cycles and bridges to w, as shown in the figure below. When we run the protocol on $\mathcal{G}_1$,



the same messages are sent; as no messages are sent on $\{u, w\}$ or $\{w, v\}$, and because the algorithm is event-driven, no messages reach the new parts of the graph. Hence the new edges cannot possibly be classified correctly. Thus in a correct protocol, every edge carries at least one message, and the E message lower bound follows.

The time lower bound has essentially the same proof. If a bridge-finding algorithm terminates in less than $\frac{\mathsf{Diam}(\mathcal{G})}{2}$ time on some graph $\mathcal{G}$, then some node receives no messages; we would attach new cycles and bridges to that node. So, no algorithm of the described form can beat ours by more than a constant factor on *any* graph. Other "optimal" algorithms are, in contrast, optimal only on *some* graphs.

Open Questions

Is bridge-finding strictly easier than finding blocks? Peleg and Rubinovich proved a $\tilde{\Omega}(\sqrt{n} + \mathsf{Diam})$ time lower bound for the minimum spanning tree problem. We may be able to adapt this proof to a lower bound on block-finding.

There are sequential algorithms for *strong components, triconnected components* and *planarity testing* in $O(V+E)$ time which are based on properties of DFS. In fact, if a spanning directed DFS tree is given, then our algorithm essentially computes the strong components. Do these problems admit $(o(n) + O(\mathsf{Diam}))$-time distributed solutions?