Finding Articulation Points of Large Graphs in Linear Time

Martín Farach-Colton[1], Tsan-sheng Hsu[2], Meng Li[1], and Meng-Tsung Tsai[1(✉)]

[1] Rutgers University, New Brunswick, NJ 08901, USA
{farach,ml910,mtsung.tsai}@cs.rutgers.edu
[2] Academia Sinica, Taipei 115, Taiwan
tshsu@iis.sinica.edu.tw

Abstract. Given an n-node m-edge graph G, the articulation points of graph G can be found in $\mathcal{O}(m+n)$ time in the RAM model, through a DFS-based algorithm. In the semi-streaming model for large graphs, where memory is limited to $\mathcal{O}(n \operatorname{polylog} n)$ and edges may only be accessed in one or more sequential passes, no efficient DFS algorithm is known, so another approach is needed.

We show that the articulation points can be found in $\mathcal{O}(m+n)$ time using $\mathcal{O}(n)$ space and one sequential pass of the graph. The previous best algorithm in the semi-streaming model also uses $\mathcal{O}(n)$ space and one pass, but has running time $\mathcal{O}(m\alpha(n)+n\log n)$, where α denotes the inverse of Ackermann function.

Keywords: Articulation points · Semi-streaming algorithm · Linear-time algorithm · Space lower bound

1 Introduction

An ***articulation point*** is a node whose removal increases the number of connected components of a graph. There are efficient algorithms in various models for finding all articulation points in an n-node m-edge graph G. For example, in the RAM model, Hopcroft and Tarjan [10] give a DFS-based algorithm that runs in $\mathcal{O}(m+n)$ time.

This classical algorithm does not scale to graphs that are larger than memory. We consider algorithms in the semi-streaming model [11–13], in which we are allowed $\mathcal{O}(n \operatorname{polylog} n)$ working space and edges may be accessed in sequential read-only passes through the graph. The goal is then to minimize the number of passes and the time complexity of the algorithm.

Some graph problems, e.g. connectivity or minimum spanning tree, can be solved optimally [7]. Other graph problems, e.g. counting the number of 3-cycles, maximum matching and graph degeneracy, can be approximated [1,3,6]. Some

This research was supported in part by NSF grants CNS-1408782, IIS-1247750 and by Ministry of Science and Technology, Taiwan, Grant MOST 103-2221-E-001-033.

F. Dehne et al. (Eds.): WADS 2015, LNCS 9214, pp. 363–372, 2015.
DOI: 10.1007/978-3-319-21840-3_30

fundamental problems, such as breath-first search, depth-first search, topological sorting, and directed connectivity, are believed to be difficult to solve in a small number of passes [9,12,13]. Hence, the known algorithms [2,7] for finding articulation points take approaches other than computing a DFS tree.

Feigenbaum et al. [7] gave a first semi-streaming algorithm for finding articulation points. Their algorithm, which we refer to as the FKMSZ algorithm, has quadratic run time $\mathcal{O}(mn\alpha(n))$, where α denotes the inverse Ackermann function. Ausiello et al. [2] later gave an algorithm with run time $\mathcal{O}(m\alpha(n)+n\log n)$. Both these algorithms use $\mathcal{O}(n)$ space and perform one pass. Here, we present the first linear-time algorithm for this problem. It also uses $\mathcal{O}(n)$ space and performs one pass.

Instead of maintaining a structure that processes each incoming edge as it is scanned, we achieve optimality by buffering incoming edges and processing them in batches of size $\mathcal{O}(n)$. We extend this approach to the problems of computing spanning trees and of finding all ***bridges***, where a bridge is an edge whose removal increases the number of connected components. Our algorithm has run time $\mathcal{O}(m+n)$, which improves the run time $\mathcal{O}(m\alpha(n))$ that comes from directly using the disjoint union-find set data structure [14].

The proposed algorithm not only has an optimal time complexity but has an optimal space complexity. A lower bound for space complexity can be obtained by noting that biconnectivity[1] is a ***balanced property*** [8]. For any balanced property $\mathcal{P}$, testing property $\mathcal{P}$ with probability at least 3/4 has a space lower bound of $\Omega(n)$ bits. Since finding articulation points is no easier than biconnectivity, it has a space lower bound of $\Omega(n)$ bits. In Section 6, we give a tighter analysis that finding articulation points in one sequential pass requires $\Omega(n\log n)$ bits. Hence, the space complexity of the proposed algorithm is optimal.

Organizations. In Section 2, we illustrate the idea of batches on two simpler problems. In Section 3, we revisit the FKMSZ algorithm. We explain a simple version of the proposed algorithm in Section 4 and defer the discussion of the full version to Section 5. In Section 6, we prove the space lower bound, $\Omega(n\log n)$ bits.

2 Preliminaries

We begin by showing how to reduce the running time for two simpler problems: finding a spanning tree and all bridges in a given graph G. We illustrate the idea of buffering scanned edges and processing them in a batch. This is the main idea used in our articulation-point algorithm.

Consider a spanning-tree algorithm in the semi-streaming model, and let F be a spanning forest of G, given the edges seen so far. As each edges e gets scanned, it can be added to F if it does not form a cycle. Testing cyclicity can be accomplished via a disjoint union-find data structure, which takes $\mathcal{O}(m\alpha(n))$ in total.

[1] A graph is biconnected iff it has no articulation point.

In order to reduce the total running time, process n edges for inclusion into the tree, instead of one at a time. Let B be the set of the next n edges to process, and let F be the current spanning forest. Compute a spanning forest of $B \cup F$ in $\mathcal{O}(n)$ time by an in-memory DFS. After all $\mathcal{O}(m/n)$ batches of edges have been processed in a single pass, the final F is a spanning forest of the original graph and the total computation time is $\mathcal{O}(m+n)$.

We apply the same idea to finding all bridges. Let F denote the spanning forest produced by the above algorithm. Note that if an edge $e \notin F$, then the edge e is on some cycle and thus cannot be a bridge. In addition to computing F, compute F_D, a spanning forest of $G \setminus F$, the discarded edges. This can be computed during the same pass where F is computed. Together they take $\mathcal{O}(n)$ space, one pass and $\mathcal{O}(m+n)$ time to compute. Once F and F_D are computed, the bridges in G can be reduced to find bridges in $F \cup F_D$ due to Lemma 1, thus in $\mathcal{O}(n)$ time by a DFS.

This approach improves the previously best $\mathcal{O}(m\alpha(n) + n \log n)$-time algorithm for finding bridges [2] to linear time.

Lemma 1. *An edge $(u,v) \in bridge(G)$ if and only if $(u,v) \in F \setminus F_D$ and $(u,v) \in bridge(F \cup F_D)$, where $bridge(H)$ denotes the set of bridges in graph H.*

Proof. Let $F_D = T_1 \cup T_2 \cup \cdots \cup T_k$, where each T_i is a maximal tree in F_D.

($\Rightarrow$) If $(u,v) \in bridge(G)$, then $(u,v) \in F$, $(u,v) \notin F_D$. Assume that $(u,v) \notin bridge(F \cup F_D)$, then there is a path P connecting nodes u, v in $F \cup F_D$ without passing through (u,v). The path P is also in G because $F \cup F_D \subseteq G$, a contradiction.

($\Leftarrow$) If $(u,v) \in F \setminus F_D$ and $(u,v) \in bridge(F \cup F_D)$, then $u \in T_a, v \in T_b$ for some $a \neq b$. Assume that $(u,v) \notin bridge(G)$, then there is a path P connecting nodes u, v in G without passing through (u,v). Since $(u,v) \in bridge(F \cup F_D)$, there are some edges $(x_1, y_1), (x_2, y_2), \ldots$ in P are discarded. Note that for any discarded edge (x_i, y_i) the nodes x_i, y_i are both contained in some T_j, implying that a path P_i in T_j connects nodes x_i, y_i. Since $u \in T_a, v \in T_b$ for some $a \neq b$, then $(u,v) \notin P_i$ for all i. Therefore, the closed loop formed by bridge (u,v) and path P with replacing the discarded edges with P_i's (note that $(u,v) \notin P_i$) implies a simple cycle passing through (u,v) in $F \cup F_D$, a contradiction. □

3 The FKMSZ Algorithm

The classical algorithm for finding articulation points in the RAM model generates a DFS tree T and detects articulation points by identifying backedges. However, in the semi-streaming model, no efficient algorithm is known for generating a DFS tree. The FKMSZ algorithm replaces the DFS tree with an arbitrary spanning tree, implicitly relying on Lemmas 2 and 4. Since these lemmas were not stated as such in [7], we provide a statement and proof for each here for completeness.

We define some notions before proceeding to the lemmas. Given a spanning tree T of graph G, if nodes u, v are both tree neighbors of some node x, then we

say nodes u, v are ***co-paired at node*** x or that they are a ***co-pair*** for short, since x is uniquely defined as the only shared neighbor of u and v.

We say that nodes u and v are ***tree-biconnected*** if there exists an edge $e \in G \setminus T$ such that u and v are biconnected in $T \cup \{e\}$. Note that if two nodes are tree-biconnected, they are biconnected, but the converse is not true. Tree-biconnectivity is easier to test for than biconnectivity.

Lemma 2. *Given a spanning tree T of graph G, a node x is an articulation point if and only if some co-pair at x is not biconnected.*

Proof. ($\Rightarrow$) By definition, if x is an articulation point in graph G, then, for some nodes $a, b \in G$, $a, b \neq x$, every path connecting a, b passes through x. This implies that, for some neighbors $u, v \in G$ of node x, every path connecting u, v passes through node x.

We divide the x's neighbors into two classes w.r.t. T: tree neighbors and non-tree neighbors. Suppose that node u is a non-tree neighbor of node x, then u, x are connected by a non-tree edge and therefore u and some x's tree neighbor are connected in $G \setminus \{x\}$. Therefore, no matter whether u, v are x's tree neighbors or non-tree neighbors, if nodes u, v are disconnected in $G \setminus \{x\}$, then some pair of x's tree neighbors are also disconnected in $G \setminus \{x\}$. Hence, some co-pair at x is not biconnected.

($\Leftarrow$) Suppose that x is not an articulation point, and let y be an articulation point that separates u and v. Such a y must exist because u and v are not biconnected. But removing $y \neq x$ leaves the u, x, v path intact, contradicting that y separates u and v. □

Corollary 3. *If x is a leaf node in any spanning tree T of graph G, then x is not an articulation point of graph G.*

Lemma 4. *Given a spanning tree T of graph G, a co-pair (u, v) at node x is biconnected if and only if there exist nodes $u = w_0, w_1, \ldots, w_t = v$ such that (w_{i-1}, w_i) is a tree-biconnected co-pair at node x for all $i \in [t]$.*

Proof. ($\Leftarrow$) If (w_{i-1}, w_i) is a tree-biconnected co-pair at node x, then nodes w_{i-1}, w_i are contained in some cycle of $T \cup \{e\}$ for some non-tree edge e. Therefore, nodes w_{i-1}, w_i are connected in $G \setminus \{x\}$. Since connectivity is transitive, u, v are connected in $G \setminus \{x\}$.

($\Rightarrow$) Observe that $T \setminus \{x\}$ is a set of subtrees. Each of x's tree neighbors belongs to an unique subtree and each subtree contains an unique tree neighbor of x. Observe further that $G \setminus \{x\}$ is a set of connected components. The connected components induced by the forest is a refinement of the connected components of the graph. That is, each connected component of the graph is spanned by one or more trees in the forest.

Since (u, v) is a co-pair at x, nodes u, v belong to different subtrees T_u, T_v. Since nodes u, v are biconnected, T_u, T_v are subgraphs of the same connected component $\mathcal{C}$. Suppose $\mathcal{C}$ contains k subtrees, then $k - 1$ non-tree edges suffice

to connect the subtrees. Each of the $k-1$ non-tree edges indicates that a co-pair at x is tree-biconnected, implying that there exist nodes $u = w_0, w_1, \ldots, w_t = v$ such that (w_{i-1}, w_i) is a tree-biconnected co-pair at node x. □

To realize the procedure in Lemma 4, we need an Union-Find data structure. In Section 4, we will introduce an Union-Find data structure that improves the run time of FKMSZ, but for now we will use a standard solution [14]. Let $S(x)$ be such a data structure for x, and initialize $S(x)$ with x's tree neighbors. The main idea of the algorithm is, for each tree-biconnected co-pair (u, v) at node x, to union u and v in $S(x)$. Thus, by Lemmas 2 and 4, we know that when we are done processing all edges, x is an articulation point iff $S(x)$ contains multiple sets, which we can check by performing a find on each element in $S(x)$. Putting this together gives the FKMSZ Algorithm:

```
1 Find a spanning tree T of graph G;
2 Prepare a union-find data structure S(x) for each node x and make an element
  in S(x) for each of x's tree neighbors ;
3 foreach incoming non-tree edge (u, v) do
4 |   Find the path P_T(u, v), a_1 = u, a_2, ..., a_t = v in tree T;
5 |   For each co-pair (a_{i-1}, a_{i+1}), union a_{i-1} and a_{i+1} in S(a_i);
6 foreach node x do
7 |   Let r_x be the find of any element in S(x).;
8 |   foreach element y in S(x) do
9 |   |   if find(y) ≠ r_x, report x as an articulation point & break;
```

Algorithm 1. Pseudo-code of FKMSZ algorithm.

4 A Two Pass Algorithm for Articulation Points

We explain a simple, two-pass version of our algorithm in this section and defer the full one-pass version to Section 5. The simplified algorithm finds all articulation points of an n-node m-edge graph G in $\mathcal{O}(m+n)$ time after two sequential passes on the entire graph. We assume that graph G is connected; otherwise, one can adapt our algorithm to the unconnected cases in a straightforward way.

Our algorithm proceeds as follows. In the first pass, we find a spanning tree T of graph G and preprocess T. In the second pass, we execute Algorithm 1, achieving linear time by exploiting our preprocessing.

4.1 First Pass

We find a spanning tree T of graph G in $\mathcal{O}(m+n)$ time. Before the second pass, we root T at an arbitrary node and preprocess T in $\mathcal{O}(n)$ time to answer the following queries in $\mathcal{O}(1)$ time:

(1) $\text{DEG}_T(x)$: the degree of node x in tree T,
(2) $\text{DEPTH}_T(x)$: the depth of node x in tree T,
(3) $\text{LCA}_T(u, v)$: the lowest common ancestor of nodes u and v in rooted tree T [4],
(4) $\text{LA}_T(u, d)$: the ancestor of node u that has depth d in rooted tree T [5].

In addition, we need to build, for each node x, an union-find data structure, $\text{UF}(x)$. We initialize $\text{UF}(x)$ with all of its neighbors. We specify an union in the typical manner: $\text{UF}(x).\text{union}(u, v)$ performs an union in $\text{UF}(x)$ between the set that contains u and the set that contains v.

In order to beat the bound for union find, we do two things. First, rather than allow arbitrary find queries, we only allow queries $\text{UF}(x).\text{one}()$, which returns TRUE if $\text{UF}(x)$ contains only one set, that is, if all sets have been merged into one. Second, we favor unions over queries. As we will see in our analysis, unions are much more common than queries, so this tradeoff will give us a better total run time than using an off-the-shelf union-find algorithm would.

Lemma 5. *The union-find data structure* $\text{UF}(x)$ *can be implemented using* $\mathcal{O}(\text{DEG}_T(x))$ *space such that* $\text{UF}(x).\text{union}(u, v)$ *takes amortized constant time and* $\text{UF}(x).\text{one}()$ *takes* $\mathcal{O}(\text{DEG}_T(x))$ *time.*

Proof. Let each set in $\text{UF}(x)$ be a node, and let $d = \text{DEG}_T(x)$. We maintain a forest F of all nodes, where two nodes are in the same tree iff they are in the same set. This takes space $O(d)$. We use a buffer of size d. During $\text{UF}(x).\text{union}(u, v)$, an edge (u, v) is placed in the buffer. If the buffer is not full, then $\text{UF}(x).\text{union}(u, v)$ takes constant time. If the buffer is full, let B be the set of edges in the buffer. We compute a new spanning forest of $F \cup B$ in time $\mathcal{O}(d)$. The new spanning forest takes space $\mathcal{O}(d)$, and the buffer is now empty. Since this flushing step happens after every d edge insertions, the amortized edge insertion cost is $\mathcal{O}(1)$.

The query returns true iff $F \cup B$ has a single connected component, which can be checked in $\mathcal{O}(d)$ time. □

4.2 Second Pass

We need to apply the unions specified by Algorithm 1 for each tree-biconnected co-pair found. However, if we do this, then each of the m non-tree edges found during the second pass would take time equal to the length of the cycle induced by adding the edge to T. In the worst case, we would end up with $\mathcal{O}(mn)$ time.

The problem is that this approach unions the same sets many times. To improve this, instead of enumerating the co-pairs on path $P_T(u, v)$ for each non-tree edge (u, v) individually, we defer the enumeration until there are n such paths waiting for enumeration. Then, we enumerate the co-pairs on n paths in a batch. In this way, we can avoid much of the work of finding the same co-pair many times, as follows.

Decompose each path $P_T(u, v)$ into paths $P_T(u, w)$ and $P_T(v, w)$, where $w = \text{LCA}_T(u, v)$, the lowest common ancestor of node u and node v in tree T. Then,

the set of co-pairs on path $P_T(u, v)$ is the union of co-pairs on path $P_T(u, w)$, those on path $P_T(v, w)$, and the co-pair (w_u, w_v) if w_u, w_v exist, where by w_u we denote the child of node w that is an ancestor of node u in tree T and likewise for node w_v. Since there are at most n co-pairs of this last form, the enumeration of such co-pairs takes $\mathcal{O}(n)$ time. Hence, the only difficulty lies in how to union the short paths to reduce the repeated enumeration.

Note that all such paths go from a descendant to an ancestor. We partition the paths by their deepest node. Now, for each u, we union all the paths in u's partition. Notice that if $P_T(u, a)$ and $P_T(u, b)$ are in u's partition, then a and b are both ancestors of u, so one is an ancestor of the other. Furthermore, $P_T(u, a) \subseteq P_T(u, b)$ if b is an ancestor of a, a condition we can check in $\mathcal{O}(1)$ time since we have precomputed the depth of every node. Thus, all we need to do is find the shallowest node in u's partition, and we can discard all other paths. There are at most $2n$ paths total, so these steps take $\mathcal{O}(n)$ time for all paths and all nodes.

This is not enough, however, because the paths we have remaining can still add to length $O(n^2)$. In order to compute all co-pairs specified by these paths, we need to compute, for each node, if it and its grandparent is in one of the specified paths. But we can test this by a single DFS of the tree as follows. Mark every node u with path $P_T(u, w)$ with $\textsc{depth}_T(w)$. Now by DFS, we can compute for every node v the depth of the shallowest endpoint of every path that goes through v. If this depth is $\textsc{depth}_T(v) - 2$ or less, then v and its grandparent form a tree-biconnected co-pair. Thus, we can find all tree-biconnected co-pairs specified by n non-tree edges in $\mathcal{O}(n)$ time. We summarize the result in Lemma 6.

Lemma 6. *Given n paths on a tree of n nodes, the (multi-) set of co-pairs on these n paths can be enumerated in $\mathcal{O}(n)$ time.*

We are ready to prove the claimed time complexity. In the second pass, for each n non-tree edges, we enumerate $\mathcal{O}(n)$ tree-biconnected co-pairs in $\mathcal{O}(n)$ time due to Lemma 6. We perform all the unions specified by those co-pairs, that is, if (x, z) is a co-pair at y, we call $\textsc{uf}(y).\text{union}(x, z)$, and repeat for each such triple. This part also takes $\mathcal{O}(n)$ due to Lemma 5. Therefore, after processing m edges, the running time so far is $\mathcal{O}(m + n)$.

Since a node x is an articulation point if and only if $\textsc{uf}(x).\text{one}()$ returns False, due to Lemmas 2 and 4, one can find all articulation points in

$$\mathcal{O}\left(\sum_{x \in T} \textsc{deg}_T(x)\right) = \mathcal{O}(n)$$

time.

Theorem 7. *Given an n-node m-edge graph G, all articulation points of G can be found in $\mathcal{O}(m + n)$ time using $\mathcal{O}(n)$ space and two sequential passes on the entire graph.*

5 A One Pass Algorithm for Articulation Points

In this section, we modify the above two-pass algorithm into a one-pass algorithm. We do so by bypassing the first pass of the two-pass algorithm and directly moving into the second pass as if the spanning tree T were given. We are able to do this because, for every step of pass two, we don't actually need all of T, but only the parts of T that have some intersection with edges seen so far during the second phase. Thus T can be built incrementally, and the first-pass preprocessing can be computed incrementally, as we encounter edges in the "second" pass.

We first make one modification to the two-pass algorithm. Note that we did not specify which spanning tree T was needed for the two-pass algorithm. Any spanning tree suffices. Thus we have the flexibility to pick one that is suitable for our one-pass algorithm. In Section 2, we present a procedure for finding a spanning tree T of graph G in linear time. In the procedure, we use a buffer of size n to accommodate incoming edges and trim the edges to obtain an intermediate spanning forest every time the buffer is full. We denote those intermediate spanning forests by $F_0 = \phi, F_1, \ldots, F_{m/n} = T$. We say that a such procedure is ***stable*** if F_i is a subgraph of F_j for all $i < j$. In Lemma 8, we prove that one can generate a spanning tree with a stable procedure in linear time.

Lemma 8. *There is a stable procedure for finding a spanning tree T of an n-node m-edge graph G that runs in $\mathcal{O}(m+n)$ time using $\mathcal{O}(n)$ space and one sequential pass on the entire graph.*

Proof. To make the procedure stable, one need to assert that the newly generated spanning forest F_{i+1} is a supergraph of F_i. In other words, one needs to keep the newer edges with a lower priority than the older ones. To achieve this, one can contract the connected component in the spanning forest F_i and conduct a DFS on the contracted graph F_i union newly added edges. Both the contraction and DFS both takes linear time. □

To mimic the two-pass algorithm, consider the ith batch of n edges. At this stage, we have spanning forest F_i, which is a subgraph of the spanning tree T. Then, for each non-tree edge (u, v) in the current batch, we need to find the path $P_T(u, v)$ given the subgraph F_i. Node u and node v cannot be contained in two different trees of forest F_i. Otherwise, we would have added edge (u, v) to F_i. We conclude that $P_{F_i}(u, v) = P_T(u, v)$.

The last problem is how to deal with the co-pairs on these paths in the claimed bound. First, we do not know $\text{DEG}_T(x)$ without the entire tree T. However, we only use $\text{DEG}_T(x)$ to allocate space for the union-find data structure $\text{UF}(x)$. One can achieve the same effect without knowing $\text{DEG}_T(x)$ by allocating $2s = \mathcal{O}(1)$ space for $\text{UF}(x)$ and iteratively doubling s whenever a new forest is computed and the degree of a node exceeds it's $s - 1$. In this way, each $\text{UF}(x)$ grows to the size $\mathcal{O}(\text{DEG}_T(x))$ and each $\text{UF}(x)$.union(u, v) still takes $\mathcal{O}(1)$ amortized time.

Second, for each F_i we preprocess the data structures to answer the queries used in the two-pass algorithm in constant time. Since the preprocessing can be done in time linear to the size of F_i, the total preprocessing time is thus $\mathcal{O}\left(\sum_i |F_i|\right) = \mathcal{O}(m+n)$. Therefore, this variation of the two-pass algorithm can be simulated by one-pass.

Theorem 9. *All articulation point of an n-node m-edge graph G can be done in $\mathcal{O}(m+n)$ time using $\mathcal{O}(n)$ space and one sequential pass on the entire graph.*

6 Space Lower Bound

In this section, we prove the following theorem.

Theorem 10. *Any semi-streaming algorithm that can output all articulation points of an n-node m-edge graph after one sequential pass requires $\Omega(n \log n)$ bits of space.*

Proof. Let function h be a bijection function from $[n]$ to $[n]$. Function h can be encoded with the graph G_h in Figure 1 without the dashed edge $e = (0, n+k)$ where $k \in [n]$ and there are n possible choices for e.

Then, we construct a stream for all edges in $G \cup \{e\}$, where the dashed edge e is placed last. The articulation points of graph $G \cup \{e\}$ are node 0 and every node $h(i)$ for $i \neq k$. Therefore, an algorithm that can output all articulation points of the graph $G \cup \{e\}$ also answers what $h(k)$ is, by computing the sum S_{AP} of the node labels of articulation points

$$S_{AP} = n(n+1)/2 - h(k).$$

At the time that a semi-streaming algorithm processes the last edge $e = (0, n+k)$, the state of memory must include an encoding of the bijection function $h : [n] \to [n]$ because based on the state of memory and the last edge $e = (0, n+k)$, one has to answer what $h(k)$ is, for any possible k. Since the number of possibilities of such a bijection function $h : [n] \to [n]$ is $n!$, the memory must have size at least $\Omega(n \log n)$ bits. □

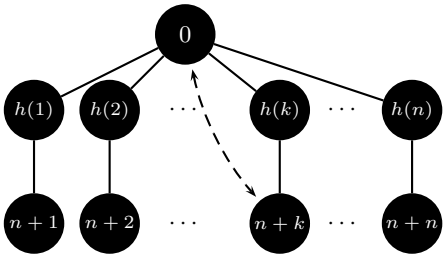

Fig. 1. Graph encoding of the bijection function $h : [n] \to [n]$

References

1. Ahn, K.J., Guha, S.: Linear programming in the semi-streaming model with application to the maximum matching problem. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 526–538. Springer, Heidelberg (2011)
2. Ausiello, G., Firmani, D., Laura, L.: Real-time monitoring of undirected networks: Articulation points, bridges, and connected and biconnected components. Network **59**(3), 275–288 (2012)
3. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: 13th Annual ACM-SIAM Symposium on Discrete algorithms (SODA), pp. 623–632. SIAM (2002)
4. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
5. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theoretical Computer Science **321**(1), 5–12 (2004)
6. Farach-Colton, M., Tsai, M.-T.: Computing the degeneracy of large graphs. In: Pardo, A., Viola, A. (eds.) LATIN 2014. LNCS, vol. 8392, pp. 250–260. Springer, Heidelberg (2014)
7. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. Theoretical Computer Science **348**(2), 207–216 (2005)
8. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the data-stream model. SIAM Journal on Computing **38**(5), 1709–1727 (2008)
9. Guruswami, V., Onak, K.: Superlinear lower bounds for multipass graph processing. In: 28th Conference on Computational Complexity (CCC), pp. 287–298. IEEE (2013)
10. Hopcroft, J., Tarjan, R.: Efficient algorithms for graph manipulation. Commun. ACM **16**(6), 372–378 (1973)
11. Muthukrishnan, S.: Data streams: Algorithms and applications. Tech. rep. (2003)
12. O'Connell, T.C.: A survey of graph algorithms under extended streaming models of computation. In: Fundamental Problems in Computing, pp. 455–476. Springer (2009)
13. Ruhl, J.M.: Efficient Algorithms for New Computational Models. Ph.D. thesis, Massachusetts Institute of Technology, September 2003
14. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM **22**(2), 215–225 (1975)