Strongly Connected Components in Graph Streams

Alexandra Porter
Stanford University
amporter@stanford.edu

CS 224W Project, Fall 2017

Abstract

Frequently studied networks, such as social networks, are often massive, directed, and change over time. Thus studying network analysis algorithms in terms of *data streaming* is a practical model for understanding how companies and other organizations can process data in a useful, realistic way. One potentially interesting feature of a directed graph is *strongly connected components*. We design and analyze a strongly connected components algorithm, *Distributed Look and Select* in a recently developed data streaming model, *X-Stream*. We analyze the effectiveness of the algorithm and the key data features which allow it to perform best. Finally, we provide empirical results on the performance of the algorithm and how the setup can be used to derive additional properties of the network.

1 Introduction

With the rapid pace of the modern internet, large networks of interest to companies and other organizations are changing at a high rate. As a result, the ability to perform network analysis in real time is extremely valuable, making data stream processing algorithms an area of interest. In a normal streaming setting, only a single pass of the data is allowed, meaning each item can only be seen once, at the time it arrives in the stream. In a graph streaming setting, each data item generally represents an edge. If the stream continues for sufficient time, complete storage of the graph could exhaust any amount of constant storage space. If the processor were to attempt to run a traditional network analysis algorithm at some point in time, the results may also be outdated by the time they are returned. As a result of these challenges, effective graph stream analysis algorithms cannot simply store the entire graph and perform traditional analysis algorithms. Instead, various streaming models and their corresponding algorithms have been studied.

In this work, we will focus on an algorithm for computing *strongly connected components* on data streams of graph edges. In real-world applications, graphs are often directed, and thus the more challenging problem of strongly connected components, as compared to undirected connected components, is a valuable tool. Furthermore, information about data streams may be needed before the stream reaches an endpoint, so algorithms which can answer queries about connectivity in real time and run indefinitely are valuable. Our contributions are the design and analysis of the *Distributed Look and Select* strongly connected components algorithm and an implementation with empirical results.

2 Definitions

In this section we give definitions and notation for graphs and stream processing. Let $G = (V, E)$ be a directed graph. Let the functions $V(G)$ and $E(G)$ denote the vertex set and edge set, respectively, of a graph. Let $G' = (V, E')$ be an undirected graph, such that E' is the set of edges in E without direction. A *connected component* C' in G' is a subgraph of G' such that for every $v_1, v_2 \in V(C') \subseteq V(G')$, there exists a v_1, v_2-path in C'. A subgraph C of G is called a *weakly connected component* of G if the subgraph C' of G' such that $V(C') = V(C)$ is a connected component in G'. A subgraph C of G is called a *strongly connected component* if for every $v_1, v_2 \in V(C)$, there exists a v_1, v_2-path and a v_2, v_1-path in C.

We use the definitions given by Henzinger et al. [4] for the basic data streaming model:

Definition 1. *A* data stream *is a sequence of data items, $x_1, ..., x_i, ..., x_n$ such that items are read a single time each, in increasing order of indices.*

Computation models are then described by the

parameters p and s, where p is the number of passes on the stream, or times the stream is repeated, and s is the storage space used during the stream processing. In the case of graph problems, each data item x_i is an edge. Let $G = (V, E)$ denote the graph described by the set of edges in the stream: $E = \{x_1, ..., x_n\}$ and $V = \{v : \exists i \text{ s.t. } v \in x_i\}$. We use stream items as our measure of time, and define the subgraph G_t at time t:

Definition 2. *Let $E_t = \{x_i : i \in [t]\} = \{(v_i, u_i) : i \in [t]\}$ and $V_t = \bigcup_{i\in[t]}\{v_i, u_i\}$. Then $G_t = (V_t, E_t)$ is the subgraph of G represented by the data stream up to time t.*

3 Related Work

Initial work on data stream processing, motivated by storage limitations relative to the scale of data processing tasks, first considered searching and sorting problems on one-way read-only tape [7]. In addition to defining the basic data stream model, Henzinger et al. [4] give bounds on the storage needed for various problems in this basic stream processing model, including finding nodes with maximum outdegree, finding nodes with the largest neighborhoods within some number of hops, and finding node pairs with the most connecting paths.

A variety of extensions to this basic streaming model have been explored, each making a trade off in parameters such as storage space, stream passes, or accuracy guarantees. We describe a series of extended streaming models which have been iteratively developed in recent work to better handle graph streams in practical settings.

Aggarwal, et al. [1] introduce a computational model constructed by augmenting the basic streaming model with a sorting primitive. They continue to use the above definition of a stream, and define a *memory m streaming pass* as a Turing machine with memory m reading the stream and outputting a new stream. As in the above definitions, the Turing machine can only read the input a single time, in the given order, and the output must also be sequential. A *memory m sorting pass* is defined to use a Turing machine with memory m, and computes a partial order on a received set of data elements $\{x_1, ..., x_n\}$. Putting these together, $StrSort(p_{Str}, p_{Sort}, m)$ is defined as the class of functions computable by up to p_{Str} streaming passes and p_{Sort} sorting passes, each with memory m, which is maintained between passes.

Demetrescu et al. [3] continue the study of extended streaming models by showing that the streaming and sorting model of Aggarwal et al. is useful even without any sorting passes, since it is still more powerful than the traditional streaming model for some problems. They refer to this multi-pass model without a sorting primitive as *W-Stream*.

3.1 Look and Select Algorithm

Some of the algorithms that have been explored on these and similar models include identifying connected components, minimum spanning trees, single-source shortest path, and strongly connected components. We focus on the algorithm for strongly connected components introduced by Laura and Santaroni [5]. They introduce the *Look and Select* (LS) algorithm for strongly connected components (SCCs).

The LS algorithm uses two data structures: a tree, T, and a union-find data structure, U. In T, all vertices are initially connected to a dummy vertex r, which remains the root of the tree. As the algorithm progresses, each node of T represents an SCC, and at any point in time all vertices are either a node in T or contained in an SCC which is a node in T. The union-find structure U is then used to store which graph vertices correspond to which known SCCs. The LS algorithm *looks* at each edge $e = (u, v)$ as it is received and then *selects* a case relative to the tree T. The five specific cases are broadly distinguished by: if edges connect branches of T, if edges create SCCs, and if edges are not relevant to connectivity.

3.2 X-Stream Model

Berry, et al. introduce the *X-Stream* model [2], which uses a directed ring of processors instead of intermediate files. Each processor receives input as a stream from the previous processor in the ring and writes to the following processor. Additionally, one processor acts as the main input point to the ring and receives a stream from an external source, in addition to the stream from its predecessor on the ring. The *X-Stream* model represents an unrolling of the *W-Stream* model, because the data streams between the processors in *X-Stream* approximately correspond to the multiple passes in *W-Stream*. In [2] an undirected connected components algorithm using the *X-Stream* model is analyzed and implemented.

4 Model and Algorithm

In this section we first give a formal description of the *Look and Select* (LS) algorithm, which is the basis for our algorithm. As described in Algorithm 1, the main step to the LS algorithm requires categorizing a received edge $e = (u, v)$ as one of the following,

with respect to the tree T representing graph G, with $h : V(G) \to \mathbb{Z}^+$ denoting the height of vertices:

- *forward*: u is an ancestor of v
- *backward*: u if a descendent of v
- *cross forward*: u and v belong to different subtrees, $h(u) \geq h(v)$
- *cross non-forward*: u and v belong to different subtrees, $h(u) < h(v)$
- *self-loop*: $u = v$

Since the LS algorithm is built on the *W-Stream* model of multiple passes, some edges are put into future streams so that they are seen again later.

Algorithm 1 Look and Select for a Single Processor

```
1: Input stream is for graph G
2: Tree T
3: Union-find structure U
4: Edge e = (u, v) ∈ E for u, v ∈ V(G)
5: p(v) : V(G) → V(G) is parent of vertex in T
6: Dummy vertex r ∈ V(T), r ∉ V(G) initially parent of all vertices
7: procedure RelabelU(e_in = (v_in, u_in))
8:     Look up u_in, v_in in U to get subsets s_u, s_v
9:     Return T edge (s_u, s_v)
10: procedure LookAndSelect(Edge e_in)
11:     e = RelabelU(e_in)
12:     if p(v) = r then
13:         Add e to T, remove (r, v)
14:     else if e is backward edge then
15:         collapse nodes u to v in T
16:     else if e is cross forward edge then
17:         return e                    ▷ Into next stream
18:     else if e is cross non-forward edge then
19:         remove (p(v), v) from T
20:         return (p(v), v)            ▷ Into next stream
21:         add e to T
22:     else if e is forward edge then
23:         drop e
        return null
24: procedure AcceptInputStream(S_0)
25:     i = 0
26:     while S_i not empty do
27:         initialize S_{i+1} to empty
28:         for e_in in S_i do
29:             e_out = LookAndSelect(e_in)
30:             if e_out != null then
31:                 append e_out to S_{i+1}
32:         i++
```

4.1 Distributed Look and Select

We can now describe our adaptation of the LS algorithm to the *X-Stream* model, called *Distributed Look and Select* (DLS), which allows a ring of processors to determine strongly connected components. Suppose that there are p processors labeled $P_0, P_1, ..., P_{p-1}$, as in [2]. They are assigned so that P_0 receives the input stream to the system and P_{p-1} is the final processor in the ring, as shown in Figure 1. Then, because the ring is closed, P_{p-1} also sends a stream of data to P_0. No communication other than the connections show are allowed, and each connection is used to transfer exactly one block of data per time unit.

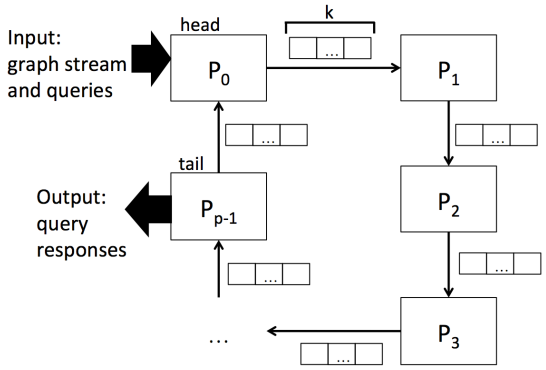

Figure 1: Setup of processors and their communication patterns.

We call P_0 the *head* and P_{p-1} the *tail*. We refer to processors with higher indices as *downstream* of a given processor. Assume all processor indices are taken modulo p when adding or subtracting.

In DLS each processor P_i maintains a union-find structure U_i such that combining U_0, U_1,...U_{p-1} gives the union-find structure for the strongly connected components. Each U_i contains information about which SCC vertices belong to, and also which previously identified SCCs have been discovered to be connected as a larger SCCs. Thus U_i is a tree, with leaf vertices corresponding to graph nodes and all other vertices corresponding to identified SCCs and sub-SCCs. Figure 2 shows an example of how some edges and vertices could be distributed on the first three processors. Figure 2a. shows the actual edges and vertices. On processor P_0 the edges are (a, b) and (b, a). On processor P_1, edges include (c, d), and (d, x) and (y, d) for some x, y in SCC_1; it does not matter the specific vertices x and y. Figure 2b. shows the corresponding union-find structures, U_0, U_1, and U_2.

Each processor also maintains a tree T_i as in the LS algorithm. While the tree U_i contains edges representing relationships between SCCs and vertices, the edges of T_i are actual graph edges. The only exception to T_i edges being graph edges is the following: as

in LS, T_i contains a placeholder root vertex r which is not part of the graph G, and some edges connecting r to graph vertices. The DLS algorithm consists of each processor P_i executing a modified version Algorithm 1 for some period of time. Processor P_0 takes its turn executing the algorithm on edges it receives from the input stream, but P_i for $i \geq 1$ executes the algorithm on only the edges it receives from P_{i-1}.

The *filling* processor at any point in time is the one currently active in the LS algorithm, meaning it is accepting edges and updating its corresponding tree and union find structures. Initially, P_0 is the filling processor, followed by P_1 and then P_2 and so on. A processor remains filling until it has identified enough edges in strongly connected components to be above the minimum storage usage m. Without such a minimum, the algorithm could continually move around the assignment of the filling processor without actually accomplishing anything.

The ring setup allows processors to defer processing an edge until later, in the same way the original LS algorithm places edges in subsequent streams to revisit later. For this to work, the non-filling processors at any point in time pass any edges they receive downstream, and the tail passes edges to the head. Then to defer handling an edge, the filling processor simply passes the edge downstream and processes it again when it comes back around.

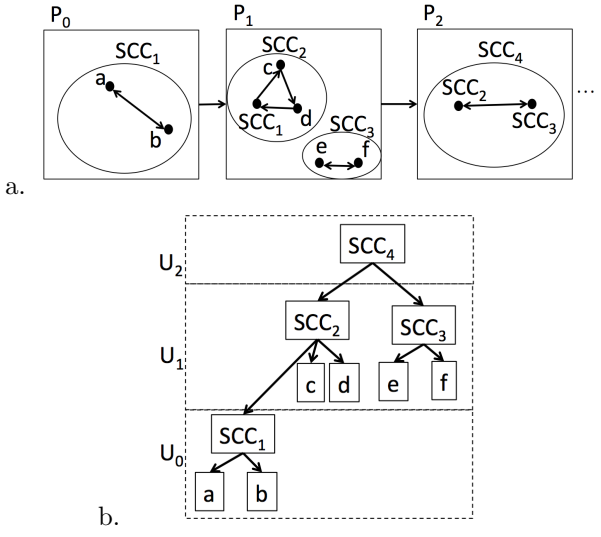

Figure 2: Example of distributed union-find structures.

When a processor is no longer filling, it needs to pass any edges not used in an SCC downstream, since they might be useful for the next filling processor to identify SCCs. This process is described in more detail in Algorithm 2. Each message between processors is a block of k constant-sized data bins, each possibly containing an edge. Thus when a processor is done filling it sends edges downstream in groups of size k.

Algorithm 2 Distributed Look and Select for Processor P_i

```
1: Input stream is for graph G
2: SCC map U_i: maps vertices and SCC labels in set V(U_i) to SCC labels (each node's parent is its label)
3: Tree T_i: tree with vertex set V(T_i) ⊆ V(U_i) (plus a placeholder root vertex)
4: Edge e = (u, v) ∈ E for u, v ∈ V(G)
5: p(v) : V(G) → V(G) is parent of vertex in T_i
6: Dummy vertex r ∈ V(T_i), r ∉ V(G)
7: m = minimum storage usage of processor
8: procedure RelabelU(e_in = (v_in, u_in))
9:     if v_in ∈ V(U_i) then s_v = U_i[v_in]
10:    else s_v = v_in
11:    if u_in ∈ V(U_i) then s_u = U_i[u_in]
12:    else s_u = u_in
13:    Return T_i edge (s_u, s_v)
14: procedure DistributedLS(Edge e_in)
15:    e = RelabelU(e_in)
16:    if p(v) = r then
17:        Add e to T_i, remove (r, v)
18:    else if e is backward edge then
19:        collapse nodes u to v in T_i
20:        label the collapsed path as an SCC
21:    else if e is cross forward edge then
22:        return {e}                    ▷ Send downstream
23:    else if e is cross non-forward edge then
24:        remove (p(v), v) from T_i
25:        add e to T
26:        return {(p(v), v)}            ▷ Send downstream
27:    else if e is forward edge then
28:        drop e
29:    if |V(U_i)| ≥ m then
30:        let E = {the first k edges not in an SCC}
31:        remove E from T_i and any endpoints from U_i
32:        return E
33: procedure AcceptInputBlock(B_in)
34:    initialize block B_out = ∅
35:    for e_in ∈ B_in do
36:        if P_i is filling processor then
37:            E_out = DistributedLS(e_in)
38:            B_out = B_out ∪ E_out
39:            if |B_out| = k then break loop
40:        else
41:            B_out = B_out ∪ {Relabel(e_in)}
42:    send B_out to P_{i+1}
```

To use the DLS system for SCC information in real time, queries are inserted in the input stream. Like an edge, a query is a pair (u, v) of vertices. When a processor P_i receives a query, it uses U_i to label the endpoints with their highest known parents, which are the largest SCCs containing them known by P_i. When the query reaches the tail processor, it outputs whether or not the queried vertices are in the same SCC. The final labels are the root or roots of the subtrees of the overall union-find structure containing u and v.

We next give an example for the system answering SCC queries. Suppose the system is in the configuration in Figure 2 and the query (a, d) is received. Processor P_0 labels a with SCC_1. Processor P_1 sees that SCC_1 and c are both children of SCC_2, and labels both SCC_2. Processor P_2 can relabel SCC_2 with SCC_4 for both ends and passes the message showing they are the same SCC to the rest of the system.

5 Analysis Results

In this section we first prove the correctness of the algorithm for identifying strongly connected components. We then analyze how properties of the graph effect the algorithm, and particular the amount of storage space needed on each processor. Finally, we describe other graph properties which can also be computed with our system.

5.1 Strongly Connected Components

We first show that Distributed Look and Select (DLS) correctly answers queries about the strongly connected components in the graph stream. Recall from Section 2 that at time t while the graph is streamed into the system, G_t represents the subgraph of G represented by the first t received edges $e_1, e_2, ..., e_t$. Let U^t_{LS} denote the union-find structure at time t during executing of the LS algorithm and let U^t_{DLS} denote the collective union find structures for all processors at time t of the DLS algorithm.

Theorem 1. *For all t, the union-find tree $U^t_{DLS} := \bigcup_{i\in[p-1]} U^t_i$ represents an equivalent union-find structure to U^t_{LS} of the LS algorithm.*

Proof. We induct on t. At $t = 0$, all structures are empty. Suppose at time $t = k$, the union-find tree $U^k_{DLS} = \bigcup_{i\in[p-1]} U^k_i$ maps every vertex v to the same outer-most SCC as U^k_{LS} (possibly with different intermediate mappings). Suppose the edge $e_k = (u, v)$ that arrives at time $t = k$ combines SCCs C_1 and C_2 into a larger SCC, C_3. W.l.o.g., suppose v previously mapped to C_1 and u mapped to C_2. By assumption of correctness of LS, U^{k+1}_{LS} now maps C_1 and C_2 to C_3, and thus any children of C_1 and C_2 to C_3. By inductive hypothesis, there exist U_i and U_j, $i, j \in [p-1]$ such that U_i maps u to C_1 and U_j maps v to C_2. Then the filling processor will receive the message that C_1 and C_2 are now combined, and map both to C_3. Thus by induction, U_{LS} and U_{DLS} give the same SCCs for any pair of vertices at all times. □

Theorem 2. *If SCC query (u, v) is inserted in the input stream at time t, a correct answer is outputted by the tail at time $t + p - 1$.*

Proof. We first show the tail outputs some answer at time $t + p - 1$. Queries, like edges, are guaranteed to be read and passed by each processor at an activation because a processor always looks at the messages it receives, and a query is always passed in the same activation it is opened, whether any information is added to the message or not. Each processor P_i for P_0,...,P_{p-2} then receives the query at time t_i and passes it to P_{i+1}. The tail, P_{p-1} receives the query at $t + p$ and outputs the answer in that activation.

We now show the answer to the query at time $t + p - 1$ is also correct for graph G_t. As the query passes through the ring, u and v are each relabeled with the maps U_0, U_1, etc. Since we know the LS algorithm computes the correct union-find structure U under an execution of LS up to time t, we apply Theorem 1, which states that the set $\{U_i : i \in [p-1]\}$ contains an equivalent union-find structure to U. Then once u and v have been relabeled by every U_i, the labels s_u and s_v will be equal if and only if u and v are in the same SCC. □

Various other properties of SCCs can also be determined from the system by adding small, constant amounts of data to the structures U_0,...,U_{p-1}. For example, each element of U_i can also track the total size of its children in the tree U. Then the following will hold.

Corollary 1. *The size of the largest strongly connected component can be identified in $p-1$ time steps.*

Proof. We prove by providing a simple algorithm. Pass a query message around the ring along with a maximum value M. Initialize $M = 0$. Let each processor P_i set M to be the size of the largest SCC known to P_i if that size is larger than the received M. Suppose some P_i reports the largest SCC size to be $|V(C)|$ for some SCC subgraph C, and some P_j with $j > i$ contains the larger SCC subgraph C' which contains C. Then P_j will set $M = |V(C')|$.

Thus the nesting of SCCs across processors does not interfere with correct maximum reporting. □

The parallelism of the ring of processors also allows the system to count the total number of SCCs faster than a single processor executing LS. Let S_{max} be the largest number of unique SCC labels on any processor, i.e. the largest set $V(U_i)$ for some $i \in [p-1]$.

Corollary 2. *The number of SCCs can be identified in at most $S_{max} + p - 1$ time steps.*

Proof. When processor P_i receives the query, pass the query immediately, followed by a message for each SCC label in $V(U_i)$. When a processor P_i receives a SCC label from P_{i-1}, forward it to P_{i+1} if the label is not in $V(U_i)$, otherwise discard it. The discarded labels do not count toward the total number of unique SCCs since they are not maximal SCCs in G. Let any processors downstream of the filling processor forward any messages received without modification. Thus the labels corresponding to the true SCCs, and only those labels, will reach the tail. They will all reach the tail by time $S_{max} + p - 1$ since it takes S_{max} time steps for the processor with the most SCC labels to pass a message for each, and each message travels at most $p - 1$ steps. □

5.2 Impact of Graph Characteristics on DLS

In this section, we describe the storage usage for the ring processors in best- and worst-case scenarios for graph G. We define the worst-case as the most imbalanced storage usage, and the highest storage capacity requirement, while best-case is the lowest storage requirement for any single processor and the most balanced.

Definition 3. *Let $m = \left\lceil \frac{|V(G)|}{p} \right\rceil$ where p is the number of processors. Then m is the* minimum usage *for the system to successfully accept the entire graph G.*

Definition 4. *The* optimal storage distribution *is exactly m vertices on the first $\left\lfloor \frac{|V(G_T)|}{m} \right\rfloor$ processors and no more than m vertices on any other processor.*

Using Definition 4, we show which graph configurations are best and worst.

Lemma 1. *The best case graph is one of two cases:*

1. *a set of disconnected SCCs of size* 2
2. *An initial SCC of size* 2 *such that every additional pair of edges creates G_t as one big SCC.*

Proof. Assume m is even, since otherwise the result may be off by a single vertex.

For the first case, at time t, every SCC, C_i for $i = 1, 2, ..., \frac{|V(G_t)|}{2}$ is completely disconnected from any other SCC. For time $t \leq m$, all vertices are stored on P_0. For $m < t \leq 2m$, all new vertices and edges that did not appear in the first m time units are stored on P_1. This continues for all processors: when a processor is finished filling, it holds exactly m vertices (and m edges) and begins to forward edges.

In the second case, every pair of edges that arrives at time t and $t+1$ add a new node to the SCC that makes up the entire graph G_t. If a processor P_i has finished filling at time t when an edge with a new vertex arrives, P_i can forward the edge e_t to P_{i+1} without removing anything it stores or taking anything new. Thus each processor P_i can finish filling when it holds exactly m vertices and the resulting storage distribution is optimal. □

Theorem 3. *In the best case, the system will have $|V(U_i)| = O(m)$ for every processor P_i, $i \in [p-1]$.*

Proof. In the cases described in Lemma 1, each processor stores m graph vertices. Additionally, each pair of vertices adds exactly 1 or 2 new SCC labels to the tree U_i. Thus there are $m+m$ or $m+2m$ vertices in U_i. □

When the storage distribution is not optimal, we measure it using the following two metrics. Let P_f be the final filling processor; f may take any value $0, ..., [p-1]$.

Definition 5. *The* processor usage *is $u = \frac{f+1}{p}$.*

Definition 6. *The* distribution imbalance *of a system and graph is $d = \frac{v_f}{m}$ where v_f is the number of vertices stored on P_f (using Definition 3 for m).*

We then show that the same type graph results in the worst values for both of these measures.

Lemma 2. *The graph configuration with the minimum processor usage u and maximum distribution imbalance d is one in which there are no SCCs.*

Proof. Suppose the system receives a graph G with no SCCs. Without identifying any SCCs, at no point can P_0 finish being the filling processor. Then $P_f = P_0$, and $u = \frac{1}{p}$. There is no case in which u could be less because P_0 always starts as the filling processor.

Additionally, $d = \frac{|V(G)|}{m}$, and applying Definition 3, $d = p$. Any graph G' with at least one SCC, C with at least 2 nodes will have $d = \frac{|V(G')|-|V(C)|}{m} < p$. Thus a graph with no SCCs precisely describes the conditions resulting in a maximum value of d. □

Theorem 4. *In the worst case, the system will store* $|V(U_0)| = |V(G)|$ *vertices on processor* P_0.

Proof. As shown in Lemma 2, in the worst case, the entire graph with have to be stored on P_0. Since no SCCs are created, there are also no vertices added to U_0 other than the actual graph vertices. □

5.3 Other Graph Properties

We next show that other features of the graph G_t can be approximated. Some can also be computed exactly by storing small amounts of additional information.

The maximum outdegree of the graph and the vertex with the highest outdegree can be approximated by passing a message around the ring once, i.e. in $p-1$ time units, by using the edges of T_0,...,T_{p-1}. In Section 6.3 we show how well this works in practice.

Determining values such as a particular vertex outdegree or the maximum outdegree exactly is not possible without storing additional information, since some edges are deleted from the system entirely during the DLS algorithm. However, for some graphs, the remaining edges during DLS execution could lead to good approximations of the degree properties of the actual graph. For graphs that we cannot use the DLS information to approximate, we could also implement a "storage heap" structure on the tail processor to track any edges that are discarded by the algorithm.

Other information related to undirected connectivity can also be retrieved easily as a natural extension of the algorithm. Defining S_{max} to be the most unique SCCs on any one processor as in Section 5.1, we have:

Theorem 5. *The number of weakly connected components and their sizes can be computed in* $S_{max} + p - 1$ *messages.*

Proof. Let P_i be the filling processor. On P_0,...,P_{i-1}, apply the same algorithm given for Corollary 2. Have P_i send the labels of each node in the second level of T_i, i.e. the nodes directly connected to the placeholder root node, rather than SCC names. Then on P_i, also do not forward any SCC labels which are found in T_i. Since the tree T_i indicates how an SCCs are connected in WCCs, the labels that reach P_{p-1} correspond to unique WCCs of the graph. □

6 Empirical Results

In this section, we describe the datasets chosen to test the algorithm. We then describe our implemented simulation of the system, and give experimental results.

6.1 Data Set Analysis

We first analyze the relevant properties of the data sets which will be used in our experiments. All are from the SNAP database [6]. The selected data sets are directed and temporal. Thus the ordering of the edges to produce a data stream is well defined by the set, using the provided timestamps, and may be determined by inherent properties of the data.

The first dataset used, *EU Email*, is email data from a European research institution. The set is based on the weakly connected component of a larger data set. The second set used, *College Messages*, is comprised of private messages sent by students at the University of California, Irvine. The final set used, *Wikipedia Talk*, represents Wikipedia users editing each other's talk pages. We summarize the basic features of these sets in Table 1. The value "Static Edges" is the number of unique edges in the graph represented by the entire dataset. The value "Temporal Edges" counts each instance of an edge in the set separately, and so it is at least the number of static edges and generally greater. To use each of these datasets, we first sort the edges by the time stamps. Then the corresponding data stream can be simulated by simply reading the sorted file in sequential order.

We describe characteristics of the data sets which will have an effect on how the DLS algorithm runs. Figure 3a. shows the proportion of vertices in the largest SCC in G_t over time t in each data set. While the main SCC is the majority of the graph for *EU Email*, it is smaller in the other networks. The *College Messages* set also has the interesting property of significant but slow growth in the relative size of the largest SCC. More nodes in the main SCC increase the likelihood that new edges will be simple to process due to at least one endpoint in the SCC. Thus the total number of vertices in an SCC, shown in Figure 3b. may also give some indication on how the algorithm can be expected to perform.

Figure 3c. shows the number of distinct SCCs. More small SCCs instead of a single large one may give the algorithm more flexibility in how the stored information is distributed. Finally, in Figure 3d. we show the maximum outdegree of a node over time, for comparison to our approximation using DLS.

6.2 Implementation Design

To experiment with the Distributed Look and Select algorithm, we implement a C++ simulator of a synchronized ring of processors. At each time step, simulated processors are activated in the ring order. While

Table 1: Summary of Dataset Statistics

Dataset	Nodes	Temporal Edges	Static Edges
EU Email	986	332334	24929
College Messages	1899	59835	20296
Wikipedia Talk (subset)	50118	500000	178186

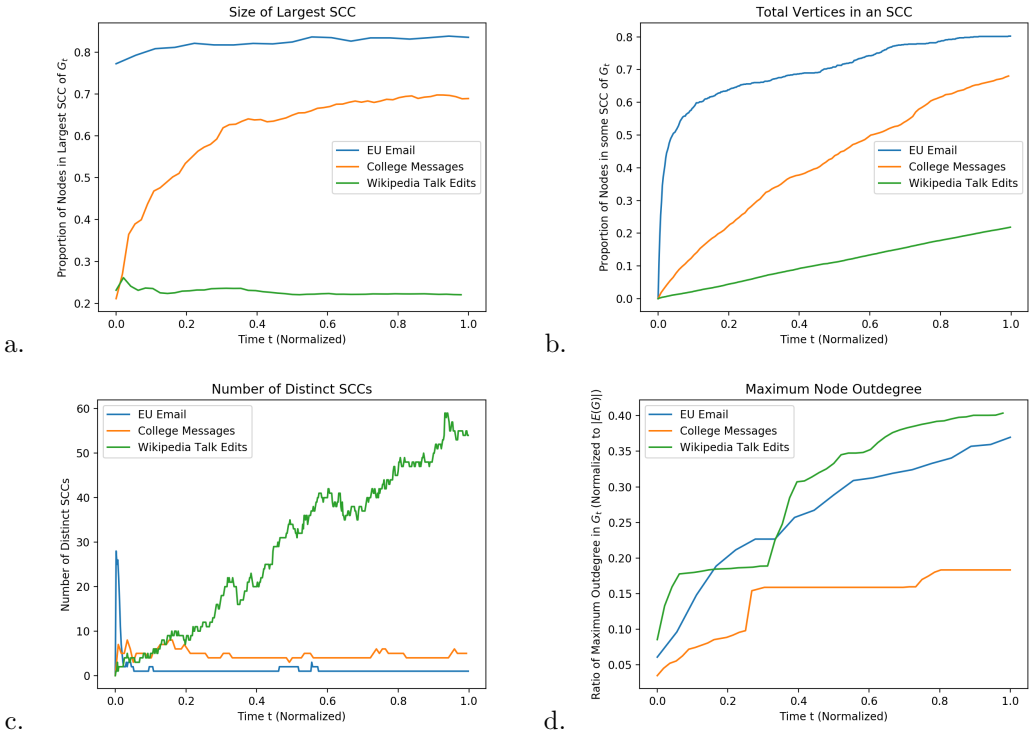

Figure 3: Analysis of data sets over time.

an applied implementation of the algorithm would not require synchronized processors, the grouping of messages into blocks of size k at each step of the algorithm would enforce a similar execution sequence among processors. The parallelization of a true distributed system would then have the effect of speeding secondary processes in the algorithm, by allowing processors to perform computations such storage optimization outside of their DLS activation steps.

We use the simulator to measure the algorithm on datasets in terms of global time steps, which correspond to processor activations and executions of Algorithm 2. We verify query correctness and report the number of edges on each processor when execution completes.

6.3 Experiments

We first use the implementation to verify that the algorithm is correct and returns the same SCCs as the Snap.py library on any graph G_t. The correctness experiment consists of randomly selecting pairs of known vertices at regular intervals in the input stream and asking both the DLS implementation and a Python Snap.py implementation if they are in the same component. As expected, since the DLS algorithm matches the LS algorithm it is correct.

In the first set of experiments, we examine how well the algorithm utilizes the distributed system when it has processed the entire data set, using the metrics of *processor usage* and *distribution imbalance* defined in Section 5.2. For both of these experiments, we measure the performance as the number of processors increases, since one of the main goals of the distributed implementation is scalability. As shown in Figure 4a., the DLS algorithm made the best use of

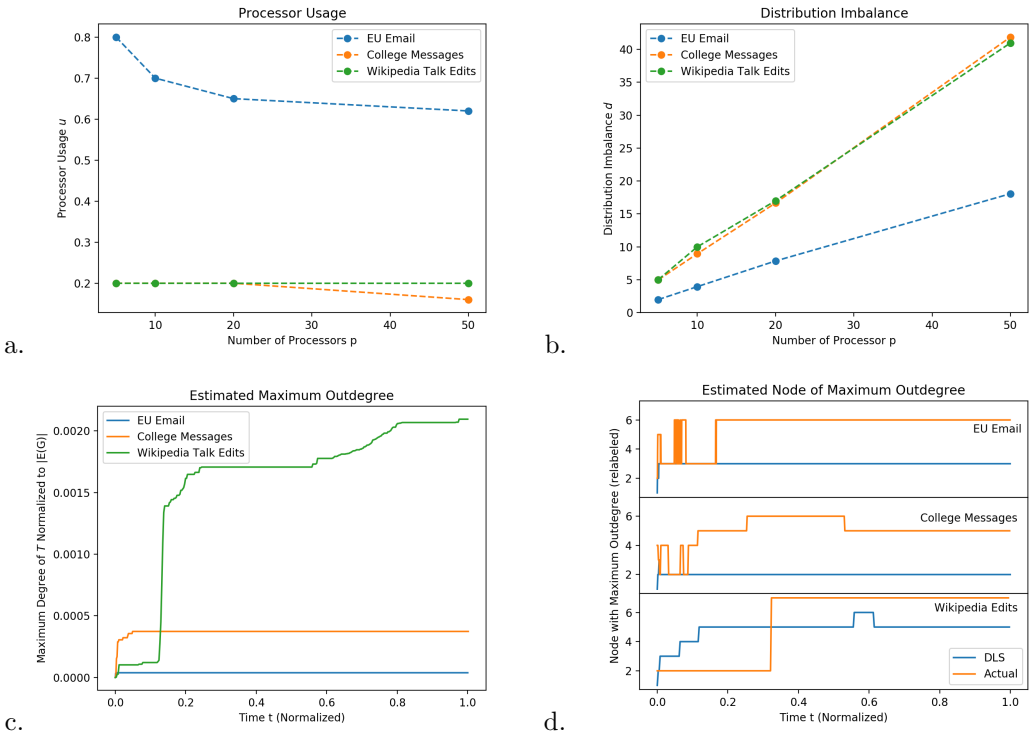

Figure 4: Experimental results: system utilization and outdegree approximation

the processors when executing on the EU Email data set. As shown in Figure 4b., the distribution imbalance was also significantly lower for this set compared to the Wikipedia Talk Edits and College Messages.

Additionally, we tested how well the approximation method described in Section 5.3 works for maximum outdegree and identifying the node with the maximum outdegree. The estimated outdegree of the system over time is shown in Figure 4c.. Note that we normalize the estimated maximum degree to the total edges in the graph as in Figure 3d., but expect it to be much lower in the experiments. The goal here was to see if we could approximate the shape of the plot, or trends over time. This was somewhat successful for the Wikipedia Talk Edits data set but not the others.

Finally, Figure 4d. compares which node is identified as having the maximum outdegree over time. To interpret this plot, the key features are the unique y-axis values, not the actual heights. Each y value corresponds to a unique node, so changes in y value correspond to changes in node of highest outdegree. Wherever a DLS line and an actual line are the same over time, both the DLS algorithm and the ground truth Snap.py implementation returned the same node. Thus we can see that in EU Email and College Messages, the actual node of highest outdegree does not change after an initial frequent oscillation. However the DLS algorithm barely changes at all before settling on a single node for the rest of the duration. The trends in the Wikipedia Talk Edit set are very different, with only one change in the ground truth, and more spaced out changes in the DLS output.

6.3.1 Result Analysis

Based on the results, we can make some predictions about how the data sets resemble our best- and worst-case graph scenarios from Section 5.2. The significantly better performance on the EU Email data set seems to indicate that graph is closer to a best-case scenario. Since the size of the largest SCC is very high and the number of distinct SCCs is low (Figure 3a. and c.), it is most likely the second of the two best-case possibilities in Lemma 1. In general, we can conclude that some real data probably does exist for which our method of SCC analysis is well-suited.

For node of highest outdegree, the EU Email and College Message sets have some success in correct identification early on which could potentially be leveraged into a method for long-term correctness. The fact that the approximation captured a more accurate trend shape over time for Wikipedia Talk Edits

also indicates that some data may be better suited to approximation with DLS. Finally, the amount of additional data needed to track actual outdegree would be relatively low, so it may be worth the additional cost in a setting where outdegree is a priority.

7 Conclusion

Through our experiments and analysis, we have shown that the graph properties affect how the Distributed Look and Select Algorithm performs. This observation implies that, in general, choosing good algorithms for applications of strongly connected components should depend on the data set features.

The initial approach of this project was to make a general algorithm that would work the same regardless of graph. While this allowed for some bounds to be shown for all graphs, in practice it was rarely very good. Thus the algorithm was modified to the current version, so that it is potentially very good for some graphs and very bad for others, but this is useful since we also showed well-suited graphs exist in real data sets.

In the future, it would be useful to explore how the *X-Stream* model and how algorithms similar to Distributed Look and Select can be used approximately, so that very large data can be processed. This would include approaches such as determining which edges are unimportant to store in the long run, or an aging protocol to delete the oldest edges from the system. It would also be interesting to study how important the temporal ordering is on a particular data set. For example, testing if an algorithm performs the same on the original ordering of edges and on a random ordering, equivalent to a random assignment of timestamps.

References

[1] Gagan Aggarwal, Mayur Datar, Sridhar Rajagopalan, and Matthias Ruhl. On the streaming model augmented with a sorting primitive. In *Foundations of Computer Science, 2004. Proceedings. 45th Annual IEEE Symposium on*, pages 540–549. IEEE, 2004.

[2] Jonathan Berry, Matthew Oster, Cynthia A Phillips, Steven Plimpton, and Timothy M Shead. Maintaining connected components for infinite graph streams. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 95–102. ACM, 2013.

[3] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading off space for passes in graph streaming problems. *ACM Transactions on Algorithms (TALG)*, 6(1):6, 2009.

[4] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. *External memory algorithms*, 50:107–118, 1998.

[5] Luigi Laura and Federico Santaroni. Computing strongly connected components in the streaming model. *Theory and Practice of Algorithms in (Computer) Systems*, pages 193–205, 2011.

[6] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[7] J Ian Munro and Mike S Paterson. Selection and sorting with limited storage. *Theoretical computer science*, 12(3):315–323, 1980.