COSC6326 Programming Assignment 2

Michael Yantosca

Contents

1 Introduction

The study of graph connectivity can yield insight into problem spaces in communication efficiency. Knowing where links are in a given graph can inform routing and help determine the best information dissemination strategies. Starting from a clean-room environment with graphs whose edges are a function of randomness, universal properties about graph connectivity may be uncovered that can then be applied to real-world examples to further refine the theory. Previously, experiments were conducted that tested the efficacy of a BFS algorithm in determining connected components, and a novel but incomplete method of randomized graph generation was used to speed up the generation process[13]. In this set of experiments, the discovery of connected components through the use of the Gallager-Humblet-Spira

(GHS) algorithm[12, pp. 102-106] is explored on the same set of Erdos-Renyi graphs[11, p. 5] and real world datasets from the SNAP database[10] for experimental consistency.

2 Analysis

2.1 `ghs-coco`

The GHS algorithm for finding connected components is given below in Algorithm 1. For the sake of legibility, several helper functions and procedures are abstracted out from the main body and presented following the main algorithm. In the implementation, only Algorithm 2 and Algorithm 3 are actually abstracted out into separate functions. All other sub-algorithms run within the main context to avoid unnecessary function call overhead, particularly Algorithm 4.

Algorithm 3 is abstracted out from the inside of the loop in Algorithm 2 for the purpose of enabling a simple gather of the component statistics upon the termination of Algorithm 1.

Algorithm 7 is a straightforward breakdown based on *vertex state*.

$s(u) =$ **FindTest** If the vertex u has neighbors in its active set, it sends a PING message along its minimum outgoing edge (here defined as the minimum opposite node label). It expects a PONG message from the opposite node indicating the other's group label. If u has no active neighbors, i.e., neighbors that are *not* in its current component, it makes an immediate ϵ-transition into the FINDSEND state.

Algorithm 1 GHS-Connected-Components, GHS Algorithm Modified to Determine Connected Components in an Arbitrary Graph

```
Require: $k$, the process/machine rank
Require: $F$, the binary-format, edge-centric graph model input file
  if $r = 0$ then
    Sample $h_a$ from an independent uniform distribution $U_a$ over $[1, M_{61} - 1]$.
    Sample $h_b$ from an independent uniform distribution $U_b$ over $[0, M_{61} - 1]$.
  Distribute $h_a$ and $h_b$ to all machines.
  $(V_r, E_r) \leftarrow$ LOAD-EDGE-TO-VERTEX-CENTRIC$(F, k)$
  Initialize exchange info $X_S$.
  $T_r \leftarrow \emptyset$ {local component set}
  $w_r \leftarrow |V_r|$
  for all $v \in V_r$ do
    if $M(v) = \emptyset$ and $N(v) = \emptyset$ then
      $T_0 \leftarrow T_0 \cup \{v\}$
    else
      $T_r \leftarrow T_r \cup \{v\}$
  $\omega_r \leftarrow |T_r| = 0$ {local termination = no nodes with outgoing edges}
  $\Omega \leftarrow \wedge_{r=1}^{k} \omega_r$ {global termination = no nodes with outgoing edges anywhere}
  while $\neg\Omega$ do
    for all $t \in T_r$ do
      $s(t) \leftarrow$ FINDTEST {Initialize component roots to find MWOE.}
      $u_e(t) \leftarrow i(t)$
      $v_e(t) \leftarrow i(t)$ {Reset MWOE to self-reference.}
      $S_r \leftarrow \{t\}$ {Add each node to the find phase work queue.}
    Execute GHS-CONNECTED-COMPONENTS-FIND-MWOE phase.
    $n_j \leftarrow 0$ {Initialize join count.}
    Execute GHS-CONNECTED-COMPONENTS-MERGE phase.
    $\omega_r \leftarrow n_j = 0$ {Local termination = no joins.}
    $\Omega \leftarrow \wedge_{r=1}^{k} \omega_r$ {Global termination = no joins anywhere.}
  Execute COMPONENT-CENSUS phase.
```

Algorithm 2 Exchange-All, Machine-wise Exchange of Information

```
Require: $X_i$, exchange information for machine $i$
Require: $send(X_i)[m]$, send buffer targeting machine $m$ from machine $i$
Require: $recv(X_m)$, receive buffer for machine $m$
Require: $k$, the number of machines participating
  for $m \in [0, k - 1]$ do
    EXCHANGE-ONE$(X_i)$
```

Algorithm 3 Exchange-One, Gathering of Information at One Machine

```
Require: $X_i$, exchange information for machine $i$
Require: $send(X_i)[m]$, send buffer targeting machine $m$ from machine $i$
Require: $recv(X_m)$, receive buffer for machine $m$
Require: $k$, the number of machines participating
  Machine $m$ gathers $len(send(X_i)[m])$ from all machines.
  Machine $m$ allocates $\sum_{i=0}^{k-1} len(send(X_i)[m])$ for $recv(X_m)$.
  $recv(X_m) \leftarrow concat(\{send(X_i)[m] \mid 0 \leq i < k\})$
```

Algorithm 4 Machine-Hash, Universal Hashing Function for Mapping Vertices to Machines

```
Require: $M_{61}$, the Mersenne prime $2^{61} - 1$
Require: $h_a$, an integer in the range $[1, M_{61} - 1]$
Require: $h_b$, an integer in the range $[0, M_{61} - 1]$
Require: $k$, the number of machines
Require: $i_v$, the vertex id to be hashed to some machine $m_h$
  return $((h_a i_v + h_b) \mod M_{61}) \mod k$
```

$s(u) =$ **FindSend** If the vertex u has children, it sends a FIND message to them. It expects all children to reply with the minimum edge along their respective subtrees via the FOUND message. If there are no children, the vertex makes an immediate ϵ-transition into the FindReply state.

$s(u) =$ **FindReply** If the vertex u is not a component root, it sends a FOUND message to its parent. If u is a component root, it makes an immediate ϵ-transition into the MwoeSend state.

$s(u) =$ **MwoeSend** The vertex sends a MWOE message to all its component subchildren. It expects no response since termination is determined by a counter decremented on the receiving end.

Other States The Idle and FindWait states do not actively send messages but rather wait for messages to be sent to them. They are in a reactive state whose behavior is handled in Algorithm 8.

Algorithm 8 is a straightforward breakdown based on *message type*.

$q =$ **ping** The vertex receives a PING message. Regardless of state, it needs to reply. The send buffer has a message added with the recipient's group label. There is no need to wait for the sending phase since we don't want

Algorithm 5 LOAD-EDGE-TO-VERTEX-CENTRIC, Distributed Algorithm for Loading an Edge-Centric Storage Model and Converting to a Vertex-Centric Memory Model

```
Require: $F$, the binary-format, edge-centric graph model input file
  Initialize edge exchange info $X_e$.
  Open $F$ for reading.
  $S_F \leftarrow$ the input file size
  $|E| \leftarrow S_F/8$
  $|E|_{\bar{k}} \leftarrow S_F/k$
  $|E|_r \leftarrow S_F \mod k$
  if $r < |E|_r$ then
    $|E|_k \leftarrow |E|_{\bar{k}} + 1$
  else
    $|E|_k \leftarrow |E|_{\bar{k}}$
  Read $|E|_k$ bytes with offset proportional to $k$ into $E_r$.
  Close $F$.
  for $i \in [0, |E|_k - 1]$ do
    $e_{uv} \leftarrow (E_r[2i], E_r[2i+1])$
    $e_{vu} \leftarrow (E_r[2i+1], E_r[2i])$
    $m_u \leftarrow$ MACHINE-HASH$(h_a, h_b, k, e_{uv}[0])$
    $m_v \leftarrow$ MACHINE-HASH$(h_a, h_b, k, e_{vu}[0])$
    Add $e_{uv}$ to the exchange buffer for $m_u$.
    Add $e_{vu}$ to the exchange buffer for $m_v$.
  EXCHANGE-ALL$(X_e)$
  $E_r \leftarrow \emptyset$
  for $(i_u, i_v) \in E_{rcvd}$ do
    $u \leftarrow V_r[i_u]$
    if $u = \perp$ then
      $u \leftarrow \{i = i_p = i_g = u_e = v_e = i_u, |g| = 1, s = \perp, w = 0, N = \emptyset, C = \emptyset, M = \emptyset\}$
      $V_r[i_u] \leftarrow u$
    if $i_u \neq i_v$ then
      $N(u) \leftarrow N(u) \cup \{i_v\}$
      if $i_v \notin E_r$ then
        $E_r[i_v] \leftarrow \emptyset$
      $E_r[i_v] \leftarrow E_r[i_v] \cup \{i_u\}$
  return $(V_r, E_r)$
```

Algorithm 6 GHS-Connected-Components-Find-MWOE, Link Phase of Algorithm 1

```
Require: $T_0$, the set of local component roots where $|M \cup N| = 0$
Require: $T_r$, the set of local component roots where $|M \cup N| > 0$
Require: $V_r$, the full set of local nodes
Require: $E_r$, the reverse lookup table of incoming edges
Require: $S_r \leftarrow \emptyset$, the set of nodes sending messages
Require: $V_e \leftarrow \emptyset$, the set of nodes holding minimum edges
Require: $w_L \leftarrow |V_r| - |T_0|$, the number of components with at least one incident edge
  $\omega_e \leftarrow w_L = 0$
  $\Omega_e \leftarrow \wedge_{r=1}^{k} \omega_e$
  while $\neg\Omega_e$ do
    Execute GHS-Connected-Components-Find-MWOE-Send phase.
    Execute GHS-Connected-Components-Find-MWOE-Receive phase.
    $\omega_e \leftarrow w_L = 0$
    $\Omega_e \leftarrow \wedge_{r=1}^{k} \omega_e$
```

to incur the overhead and complexity of quasi-states or interfere with other state transitions.

$q =$ **pong** The pinger receives a reply. If the group labels are the same, the pinger must continue to ping and is added back into the sender queue. The edge is moved to the inactive set M so the next highest edge in N can be chosen in the next sending phase. If the group labels are different, the vertex transitions into the FindSend state and is added to the sender queue. It will notify its children in the next sending phase.

$q =$ **find** Receiving this message indicates that it is the recipient's turn to find an incident minimum weight edge. The vertex transitions into the FindTest state and is added to the sender queue. From this point forward, it acts as the root of its component subtree.

$q =$ **found** Receiving this message indicates that a child subtree has propagated its minimum edge to the subtree root. The waiting counter is decremented. A comparsion is made to determine whether to accept the reported minimum edge or keep the minimum edge found by the vertex's own testing. Ordinality is determined by comparing the minimum node labels in each edge. Ties are broken by comparison of the maximum node labels in each edge. The edges themselves, however, are stored with respect to direction radiating from the component, i.e., the first label belongs to the component, the second outside. Once the counter hits zero (0), the vertex

Algorithm 7 GHS-Connected-Components-Find-MWOE-Send, Send Phase of Algorithm 6

```
while $S_r \neq \emptyset$ do
  $u \leftarrow$ Pop($S_r$)
  if $s(u) =$ FindTest then
    if $N(u) > 0$ then
      $i_v \leftarrow \min_{v \in N(u)} v$
      $m_v \leftarrow$ Machine-Hash($h_a, h_b, k, i_v$)
      $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\textsc{ping}, i_v, i(u), i_g(u))$ {Test vertex $i_v$.}
    else
      $s(u) \leftarrow$ FindSend
      $S_r \leftarrow S_r \cup \{u\}$ {$\epsilon$-transition: no neighbors outside component}
  else if $s(u) =$ FindSend then
    $w(u) \leftarrow |C(u)|$
    if $w(u) > 0$ then
      $s(u) \leftarrow$ FindWait
      for all $i_v \in C(u)$ do
        $m_v \leftarrow$ Machine-Hash($h_a, h_b, k, i_v$)
        $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\textsc{find}, i_v, i(u), i_g(u))$ {Downcast find.}
    else
      $s(u) \leftarrow$ FindReply
      $S_r \leftarrow S_r \cup \{u\}$ {$\epsilon$-transition: no children}
  else if $s(u) =$ FindReply then
    if $i_g(u) = i(u)$ then
      $s(u) \leftarrow$ MwoeSend
      $S_r \leftarrow S_r \cup \{u\}$ {$\epsilon$-transition: component root}
    else
      $s(u) \leftarrow$ Idle
      $m_p \leftarrow$ Machine-Hash($h_a, h_b, k, i_p(u)$)
      $send(X_S)[m_p] \leftarrow send(X_S)[m_p] + (\textsc{found}, i_p(u), u_e(u), v_e(u))$ {Upcast found.}
  else if $s(u) = \textsc{mwoe}$ then
    $s(u) \leftarrow$ Idle
    $w_L \leftarrow w_L - 1$
    for all $i_v \in C(u)$ do
      $m_v \leftarrow$ Machine-Hash($h_a, h_b, k, i_v$)
      $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\textsc{mwoe}, i_v, u_e(u), v_e(u))$ {Downcast mwoe.}
Exchange-All($X_S$)
Rewind($X_S$)
```

Algorithm 8 GHS-Connected-Components-Find-MWOE-Receive, Receive Phase of Algorithm 6

```
for all $(q, i_v, a, b) \in recv(X_S)$ do
  $v \leftarrow V_r[i_v]$
  if $q = $ PING then
    $m_u \leftarrow$ Machine-Hash$(h_a, h_b, k, a)$
    $send(X_S)[r] \leftarrow send(X_S)[r] + ($PONG$, a, i_v, i_g(v))$ {Direct buffer dump.}
  else if $q = $ PONG then
    if $i_g(v) = b$ then
      $M(v) \leftarrow M(v) \cup \{a\}$
      $N(v) \leftarrow N(v) \setminus \{a\}$ {Keep testing.}
    else
      $e_u(v) \leftarrow i(v)$
      $e_v(v) \leftarrow a$
      $s(v) \leftarrow$ FindSend {Found minimum incident edge.}
    $S_r \leftarrow S_r \cup \{v\}$
  else if $q = $ FIND then
    $s(v) \leftarrow$ FindTest
    $S_r \leftarrow S_r \cup \{v\}$
  else if $q = $ FOUND then
    $w(v) \leftarrow w(v) - 1$
    if $a \neq b$ then
      if $e_u(v) \neq e_v(v)$ then
        $e_{ab} \leftarrow (\min(a, b), \max(a, b))$
        $e_v \leftarrow (\min(e_u(v), e_v(v)), \max(e_u(v), e_v(v)))$
        if $e_{ab}[0] < e_v[0]$ or $(e_{ab}[0] == e_v[0]$ and $e_{ab}[1] < e_v[1])$ then
          $e_u(v) \leftarrow a$
          $e_v(v) \leftarrow b$ {Choose the edge with the smallest endpoints.}
      else
        $e_u(v) \leftarrow a$
        $e_v(v) \leftarrow b$ {Real edges take precedence over virtual edges.}
    if $w(v) = 0$ then
      $s(v) \leftarrow$ FindReply
      $S_r \leftarrow S_r \cup \{v\}$ {All children have responded. Transition to next state.}
  else if $q = $ MWOE then
    if $i(v) = a$ then
      if $i(v) \neq i_g(v)$ then
        $T_r \leftarrow T_r \cup \{v\}$
        $T_r \leftarrow T_r \setminus V_r[i_g(v)]$ {Re-root fragment to vertex with MWOE.}
    $(u_e(v), v_e(v)) \leftarrow (a, b)$
    $s(v) = $ MwoeSend
    $S_r \leftarrow S_r \cup \{v\}$
```

transitions to the FindReply state and is added to the sender queue to upcast its own minimum determination to its parent.

q = **mwoe** Receiving this message indicates that the root has chosen a MWOE to downcast through the tree. The vertex accepts this MWOE and passes it on to its own subtree. It is added to the sender queue for propagating the MWOE to its children. If the MWOE is incident to it, it adds itself to the merging set T_r to participate in the merging phase and removes the component root from the set of roots unless the vertex is already the component root itself.

The merge phase in Algorithm 9 consists of a few simple graph-wide operations. The local component roots are checked for the presence of outgoing edges, and those which need to perform a join add a message to be sent during a k-way exchange among all the machines. Those that receive the messages become the targets of the join operation and adopt their respective senders. In the case where two vertices issue a mutual join, i.e., each sends a join request to the other, the vertex with the lesser vertex id becomes the parent and remains a component root. All other participants in the joins whether sender or receiver lose their root status so as to accomplish the necessary reduction dictated by the GHS algorithm. As argued in the text, at most one pair of nodes in a component-to-be will issue that mutual join[12, p. 105], and one of them must be the new component root. The remainder of the merge phase is a simple propagation of the component root elect to its enlarged family. The root takes parentage of any old parent which was not itself along with its old children, if it had any. Its descendants are then reckoned through their prior children and in some cases parents, where a NEW message has caused a reparenting downstream, as well as through the join edges that brought the formerly disjoint fragments together.

The component census in Algorithm 10 simply performs a downcast and convergecast on all the components in the forest, after which it gathers the statistics at the machine with rank 0 for reporting to the user. This is a simple post-hoc operation with $O(n)$ message complexity.

Algorithm 9 GHS-Connected-Components-Merge, Merge Phase of Algorithm 1

```
for all $u \in T_r$ do
    if $|N(u)| > 0$ then
        $m_v \leftarrow$ Machine-Hash$(h_a, h_b, k, v_e(u))$
        $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\textsc{join}, v_e(u), i(u), i_g(u))$
        $n_j \leftarrow n_j + 1$ {Increment join counter to avoid premature termination.}
Exchange-All$(X_S)$
Rewind$(X_S)$
$S_r \leftarrow \emptyset$
for all $(\textsc{join}, i_v, i_u, i_{g(u)}) \in recv(X_S)$ do
    $v \leftarrow V_r[i_v]$
    if $i_u = v_e(u)$ then
        if $i_v < i_u$ then
            if $i_p(v) \neq i_v$ then
                $C(v) \leftarrow C(v) \cup \{i_p(v)\}$
            $i_p(v) \leftarrow v$
            $i_g(v) \leftarrow v$
            $(e_u(v), e_v(v)) \leftarrow v$
            $C(v) \leftarrow C(v) \cup \{i_u\}$
            $S_r \leftarrow S_r \cup \{v\}$ {Combined root goes to the minimum vertex in a mutual join.}
    else
        $C(v) \leftarrow C(v) \cup \{i_u\}$ {Receiving a join means adopting the sender.}
    $M(v) \leftarrow M(v) \cup \{i_u\}$
    $N(v) \leftarrow N(v) \setminus \{i_u\}$ {Prune joined edge.}
$\omega_M \leftarrow S_r = \emptyset$
$\Omega_M \leftarrow \wedge_{r=1}^{k} \omega_M$
while $\neg\Omega_M$ do
    for all $u \in S_r$ do
        for all $i_v \in C(u)$ do
            $m_v \leftarrow$ Machine-Hash$(h_a, h_b, k, i_v)$
            $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\textsc{new}, i_v, i(u), i_g(u))$
    Exchange-All$(X_S)$
    Rewind$(X_S)$
    $S_r \leftarrow \emptyset$
    for all $(\textsc{new}, i_v, i_u, g_u) \in recv(X_S)$ do
        $v \leftarrow V_r[i_v]$
        $i_g(v) \leftarrow g_u$ {Regroup.}
        if $i_p(v) \neq i(v)$ and $i_p(v) \neq i_u$ then
            $C(v) \leftarrow C(v) \cup \{i_p(v)\}$ {Adopt old parent.}
        $i_p(v) \leftarrow i_u$
        $C(v) \leftarrow C(v) \setminus i_u$ {Reparent.}
        $M(v) \leftarrow M(v) \cup \{i_u\}$
        $N(v) \leftarrow N(v) \setminus \{i_u\}$ {Prune new parent edge.}
        $(e_u(v), e_v(v)) \leftarrow v$ {Reset MWOE.}
        $T_r \leftarrow T_r \setminus \{v\}$ {Remove non-root from component roots.}
        $S_r \leftarrow S_r \cup \{v\}$ {Add to sender queue.}
    $\omega_M \leftarrow S_r = \emptyset$ {Local termination = no more NEW messages on this machine.}
    $\Omega_M \leftarrow \wedge_{r=1}^{k} \omega_M$ {Global termination = no more NEW messages anywhere.}
```

Algorithm 10 Component-Census, Final Phase of Algorithm 1

```
for all $u \in T_r$ do
    $w(u) \leftarrow |C(u)|$
    $S_r \leftarrow S_r \cup \{\}$
$w_G \leftarrow |S_r|$
$\omega_G \leftarrow w_G = 0$
$\Omega_G \leftarrow \wedge_{r=1}^{k} \omega_F$
while $\neg\Omega_G$ do
    for all $u \in S_r$ do
        if $w(u) = 0$ then
            if $i_g(u) = i(u)$ then
                $w_G \leftarrow w_G - 1$
            else
                $m_v \leftarrow$ Machine-Hash$(h_a, h_b, k, i_p(u))$
                $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\text{PONG}, i_p(u), i(u), |g|(u))$ {Upcast count.}
        else
            for all $i_v \in C(u)$ do
                $m_v \leftarrow$ Machine-Hash$(h_a, h_b, k, i_v)$
                $send(X_S)[m_v] \leftarrow send(X_S)[m_v] + (\text{PING}, i_v, i(u), \perp)$ {Downcast query.}
    Exchange-All$(X_S)$
    Rewind$(X_S)$
    $S_r \leftarrow \emptyset$
    for all $(q, i_v, i_u, b) \in recv(X_S)$ do
        $v \leftarrow V_r[i_v]$
        if $q = \text{PING}$ then
            $w(v) \leftarrow |C(v)|$
            $S_r \leftarrow S_r \cup \{v\}$ {Queue for downcast.}
        else if $q = \text{PONG}$ then
            $w(v) \leftarrow w(v) - 1$
            $|g|(v) \leftarrow |g|(v) + b$ {Add to component subtotal.}
            if $w(v) = 0$ then
                $S_r \leftarrow S_r \cup \{v\}$ {Queue for upcast.}
    $\omega_G \leftarrow w_G = 0$
    $\Omega_G \leftarrow \wedge_{r=1}^{k} \omega_G$
for all $t \in T_r$ do
    $send(X_S)[0] \leftarrow send(X_S)[0] + (i(t), |g|(t))$
for all $t \in T_0$ do
    $send(X_S)[0] \leftarrow send(X_S)[0] + (i(t), 0)$
Exchange-One$(X_S, 0)$ {Collect component counts at machine with rank 0.}
```

Theorem 1. *Algorithm 1 determines the connected components in an arbitrary graph of n vertices distributed over k machines with communication complexity $O(m + n \log n)$.*

Proof. Because the implementation uses a strictly unicast messaging model, the communication complexity follows the message complexity. Since each root vertex in the component forest yields one MWOE per phase, the worst case scenario is that all the roots align themselves in pairwise joins, causing a reduction in components by at least half. This constant reduction indicates that the number of phases will be on the order of $O(\log n)$.

Each phase consists of a find subphase and merge subphase. The find subphase takes $O(n)$ messages on account of the complexities of the FIND downcast, the FOUND convergecast, and the MWOE downcast that comprise it. Each operation takes $O(n)$ messages, so their summation is also $O(n)$. The merge subphase has a simple exchange which in the initial worst case is $\frac{n}{2}$ (pairwise mutual join) followed by a simultaneous per-component downcast which may hit all nodes in the worst case, so the message cost is likewise $O(n)$ for the merge subphase. Over all phases, the total message complexity in this regard will be $O(n \log n)$.

Suppose that the graph is a complete graph. The total component reduction will occur in the first round, but the active edges which have not been traversed in a join must be checked regardless. The checking will incur a total message cost of $O(m)$, i.e., the number of edges.

The post-hoc component census takes $O(n)$ messages. This is readily subsumed into the $O(n \log n)$ complexity posed by the find and merge subphases.

Depending on the nature of the graph, one of the two aforementioned total message complexity terms will dominate. As such, the total message complexity and therefore the communication complexity will be the sum of the two terms or $O(m + n \log n)$. □

3 Results

3.1 Test Procedures

Tests to validate correctness were performed locally on a dual-core laptop [1] running Pop!OS. [2] Tests for which results were collected systematically and graphed were done on the UH `crill` cluster with the following parameters:

- $n \in 2^{[10,20]}$, the total population size
- $k \in \{1, 2, 4, 8, 16, 32\}$, the number of distributed nodes
- $\epsilon = 0.2$, the threshold error
- p, the existential probability of a given edge,

$$p \begin{cases} < \frac{(1-\epsilon)\ln n}{n} \\ = \frac{\ln n}{n} \\ > \frac{(1+\epsilon)\ln n}{n} \\ < \frac{(1-\epsilon)}{n} \\ = \frac{1}{n} \\ > \frac{(1+\epsilon)}{n} \end{cases} \tag{1}$$

Each parameter combination was executed on 3 pre-generated graphs per regime. The graphs generated with the naive combinatorial edge selection scheme only covered graphs with $n \leq 2^{17}$ because of the immense time required for generation. However, this should still provide enough data points to observe a general trend and make cogent comparisons with the much faster but slightly incorrect binomial edge selection scheme.

Several executables contributed to the testing process:

`txt2mpig` A utility program for converting SNAP edge-centric model text files into a compact edge-centric binary format suitable for accessing via MP/IO.

`genmpig` A utility program for generating Erdos-Renyi random graphs and saving in a compact edge-centric binary format.

[1] 2 x Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz.
[2] An Ubuntu 18.04 variant.

`ghs-coco` A program which reads a compact edge-centric binary format file and distributes the vertex-centric equivalent across a k-machine context and subsequently executes Algorithm 1 on the distributed graph.

The reader is directed to the accompanying `README.md` for explicit usage instructions on the various programs as well as for a deeper explanation of implementation considerations, engineering tradeoffs, and known issues with the programs.

It should be noted that the MPI 2.1 standard[9] and g++-7.2 was used for development since they were available through the development laptop's package system, but the program compiled and ran on the `crill` cluster with g++-5.3.0 and MPI 3.0. The code requires the C++-11 standard.

In order to facilitate rapid development, extensive use was made of the C++ STL libraries. The vertex input map was a `std::map`[4] of vertex structures keyed by vertex ID. In each vertex structure, the collection GHS tree children was of type `std::unordered_set<uint32_t>`[8]. The set of neighbors was divided into an active and inactive set using the `std::set<uint32_t>`[6] type. Since the sets were ordered, the MWOE was always the head of the iterator for the active set. An incoming edge map that could have been used to facilitate faster lookup on incoming messages was constructed with a `std::map<std::unordered_set<uint32_t>>`, but it played no part in the unicast messaging scheme. The component map was a `std::map`[4] of vertex structures keyed by vertex ID. Its initial collection held all the vertices that had at least one incident edge. A separate component map was kept for vertices that had no incident edges. Component statistics were gathered at the end by iterating over both component maps.

Randomized elements[1] made use of the Mersenne twister PRNG `std::mt19937`[5] as the basis for all probabilistic distributions. The primary distributions used were the `std::uniform_int_distribution`[7], `std::bernoulli_distribution`[2], and `std::binomial_distribution`[3].

3.2 Erdos-Renyi Graphs

Results for experiments on the Erdos-Renyi graphs generated by `genmpig` are given in the following figures. The graphs are divided into group plots characterized by edge probability, i.e., $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$ or $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$,

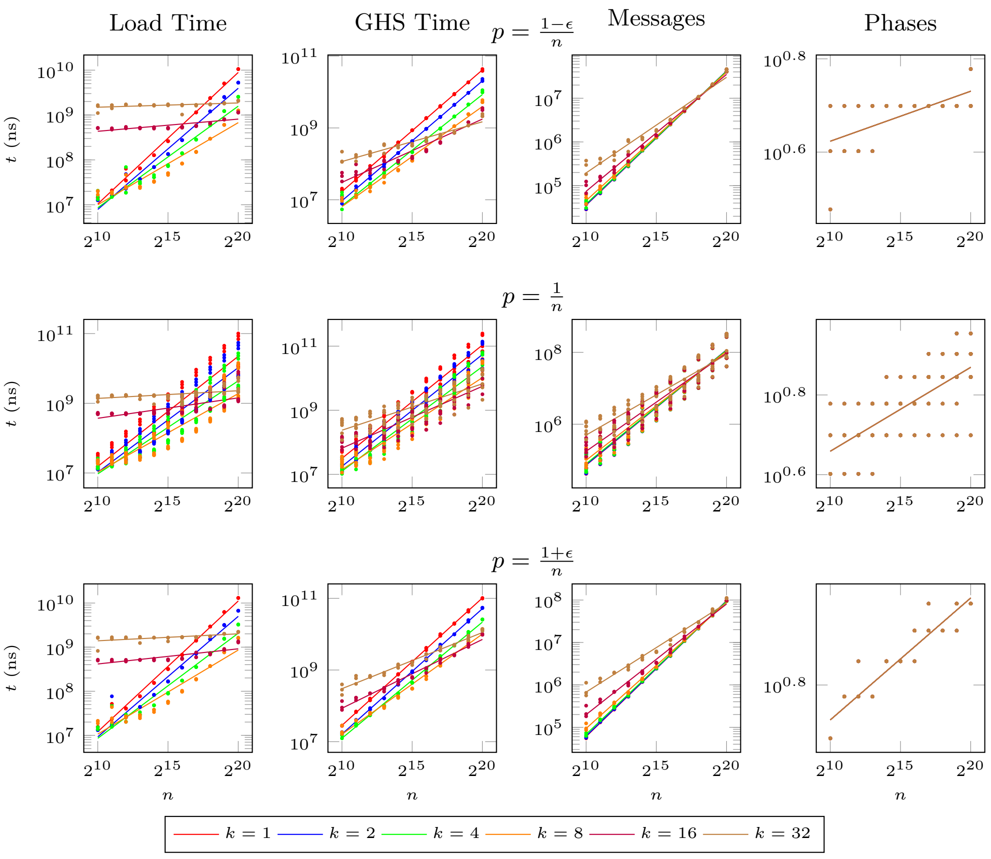

Figure 1: Empirical Complexity of Algorithm 1 on MCER Graphs, $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$

and by edge selection scheme, i.e., Monte Carlo (MCER) or Las Vegas (LVER) Erdos-Renyi graphs.

The graph loading and construction times follow the expected decrease as k scales out with the exception of the cases where $k = 16$ and $k = 32$. There is likely a machine boundary bottleneck coming into play here since the cases where $1 \leq k \leq 8$ could all fit on one machine according to the allocation strategy. Communication over the network is bound to induce an additional latency.

The execution times for the GHS algorithm actually seem to model the expected decrease as k scales out better than the BFS algorithm. The aforementioned exceptions for the higher values of k still apply, likely for the same reasons. For the sparse graphs in Figure 1, the decrease in execution time is an entire order of magnitude. As predicted in the previous trials[13, p. 19], removing the known singleton components from active participation at the start seems to have had the desired effect. The message complexity is also substantially dropped. The reader is directed to Appendix A for a set of graphs directly comparing message and time complexity between the two algorithms.

Unlike the graphs in the prior report[13, pp. 11-16] which depicted the number of rounds, the graphs here depict the number of phases, with a phase consisting of one MWOE search and one merge. Within each subphase, multiple exchanges may occur to the order of the depth of the largest component's tree. Seen here, the number of phases is clearly a constant fraction of $O(\log_2 n)$, as the phase count barely reaches the neighborhood of 10 when $n = 2^{20}$. The aggressive pruning of active edges during the merge subphase instead of waiting for them to be considered during the MWOE search subphase may have played a pivotal role in achieving this performance.

The results depicted in Figure 1 offset the improvements seen in Figure 2. The empirical time and message complexity is roughly the same if not slightly greater than the BFS algorithm (see Appendix A). The likely cause is the unicast nature of the communications in this implementation of the GHS algorithm. Whereas the BFS algorithm could utilize broadcast techniques to keep the messages small in many cases, the simultaneous though pair-wise communication employed through most of the GHS algorithm seems to have been a detriment. Under this particular implementation, the GHS algorithm seems better suited to sparse graphs, as opposed to the previous implementation of the BFS algorithm, which performed better with dense graphs[13, pp. 11,19].

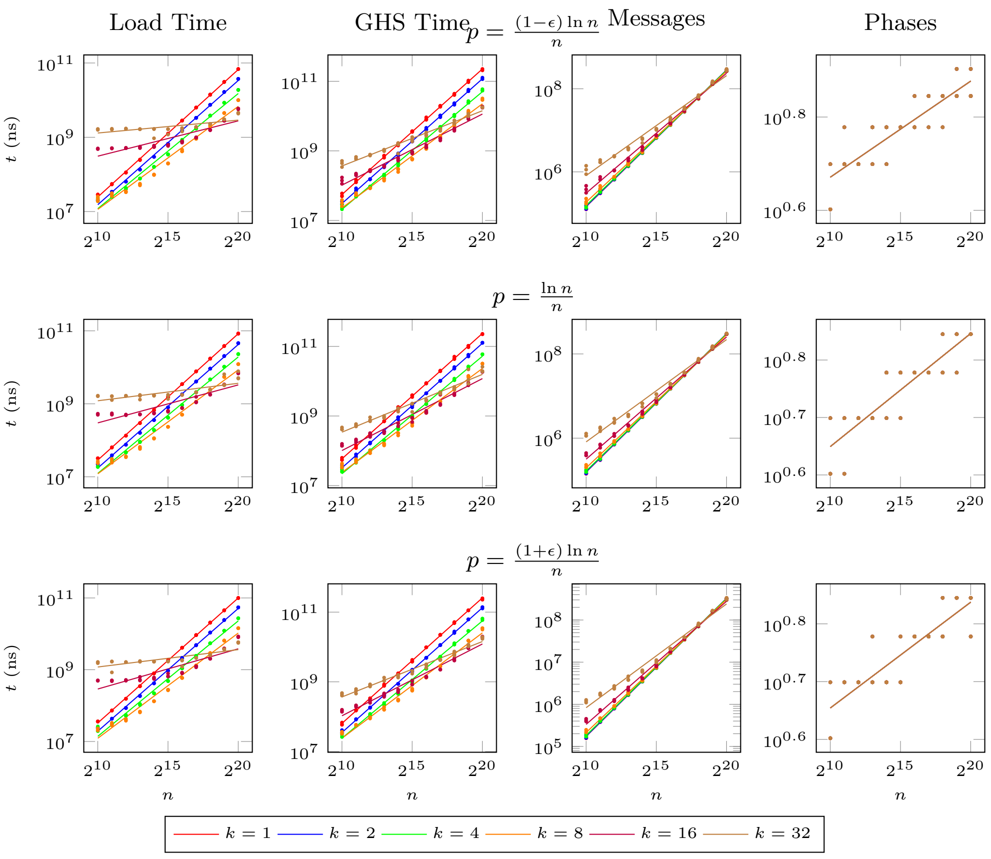

Figure 2: Empirical Complexity of Algorithm 1 on MCER Graphs, $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$

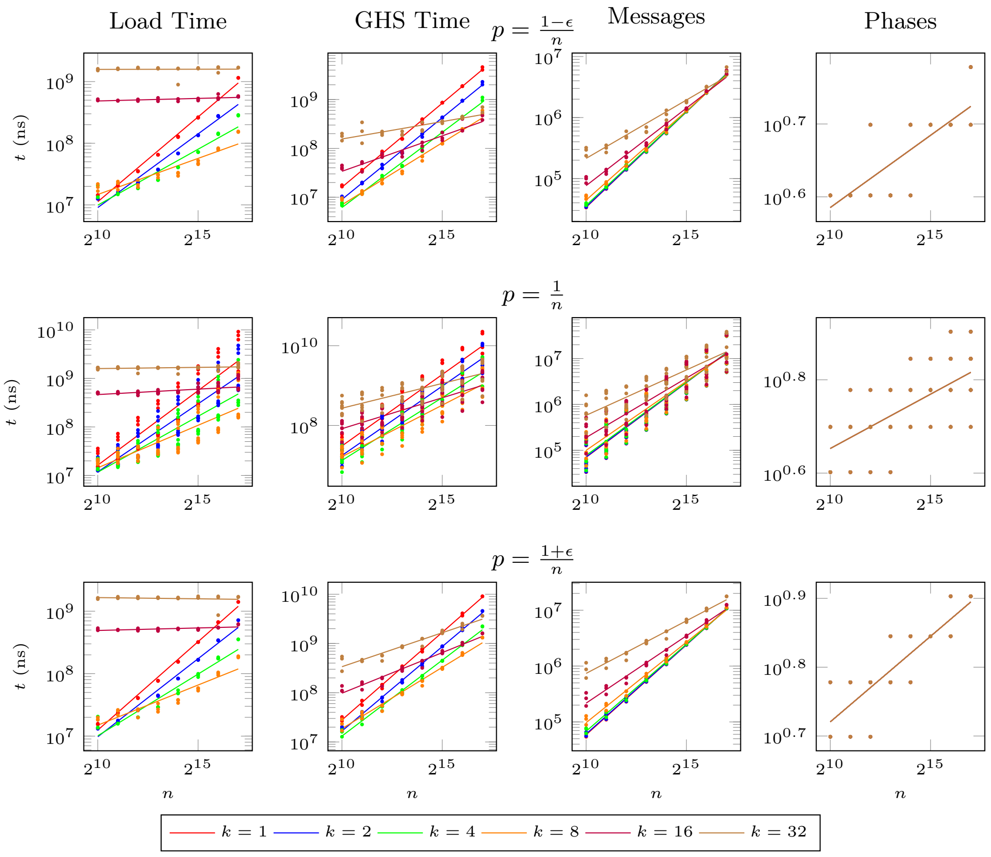

Figure 3: Empirical Complexity of Algorithm 1 on LVER Graphs, $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$

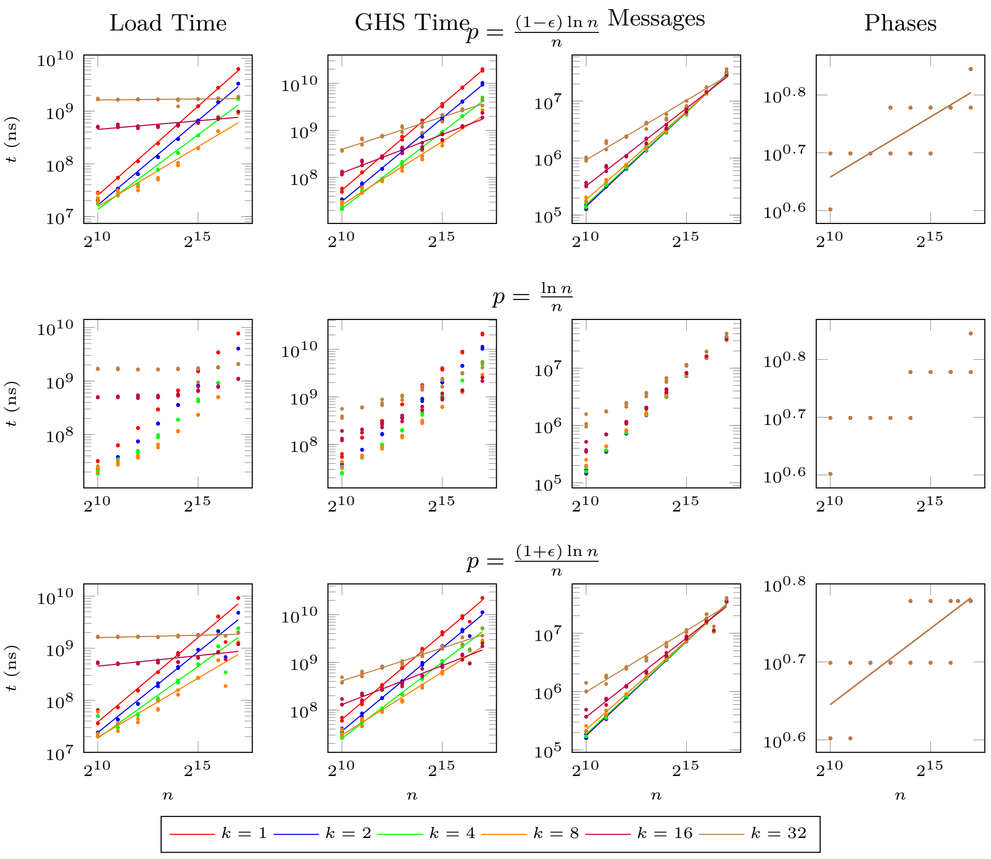

Figure 4: Empirical Complexity of Algorithm 1 on LVER Graphs, $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$

The LVER graphs depicted in Figure 3 exhibit similar behavior to that shown by the MCER graphs depicted in Figure 1. This again lends support to the rationale behind the MCER edge selection scheme as a foundation for a legitimate optimization in graph generation.

The LVER graphs depicted in Figure 4 also follow the MCER graphs depicted in Figure 2. The affinity of this implementation of the GHS algorithm for sparse graphs is likewise underscored.

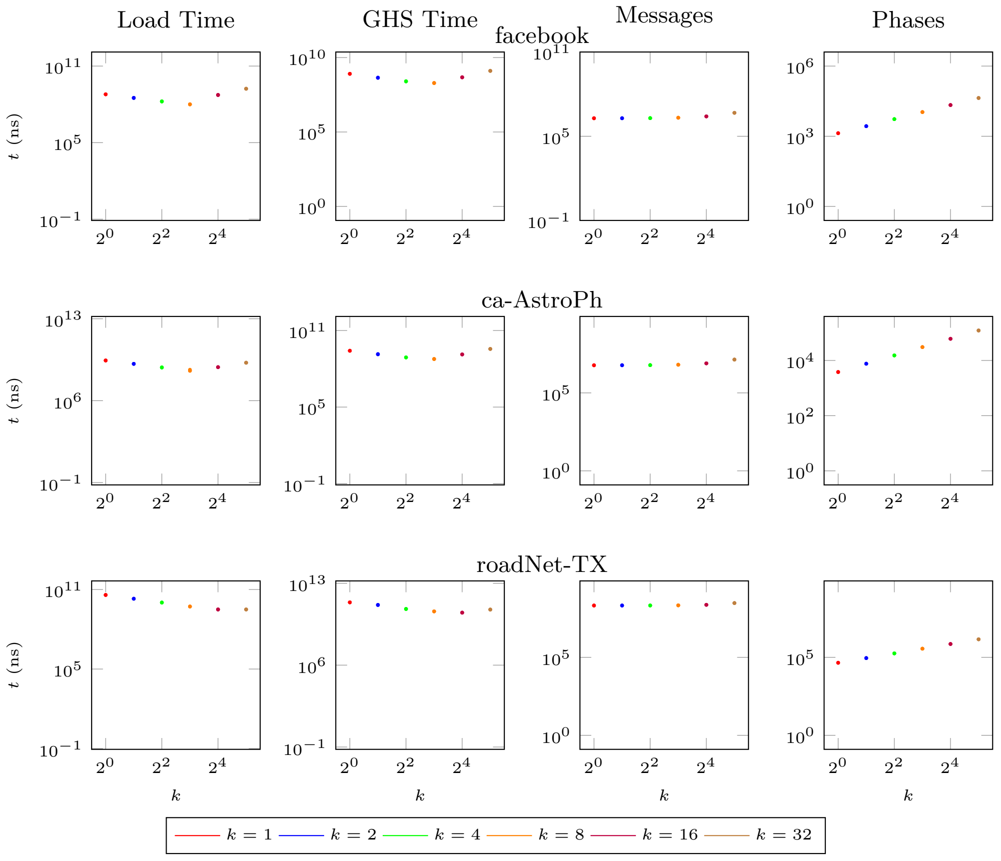

Figure 5: Empirical Complexity of Algorithm 1 on Real-World Graphs

3.3 Real-World Graphs

The real world graphs depicted in Figure 5 exhibit similar characteristics as seen with the BFS algorithm. The machine boundary effect is once again evident with the uptick in execution time for higher values of k. The fact that the graphs are the same size lends further credence to this since there is no increase in population size that can be charged for the loss of speedup.

It should be noted that the allocation strategy in these experiments differed from the BFS experiments. At the time of experimentation, the cluster was completely idle, and so the batch scripts simply requested 8

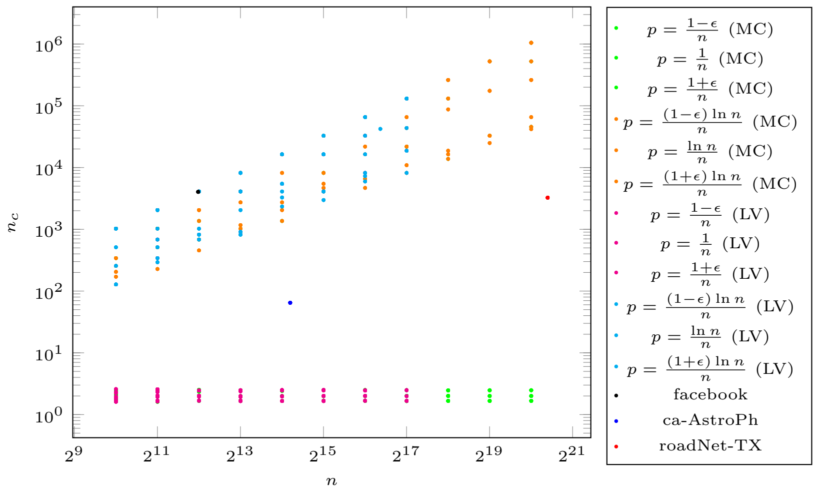

Figure 6: Component Mean Size for Various Regimes

CPUs each from four specific nodes (`crill-101`, `crill-102`, `crill-201`, and `crill-202`) regardless of actual need to avoid script proliferation and simplify the experimentation process.

Under this regime, it becomes increasingly clear that the machine boundary effect dominated and produced nearly identical results compared with the BFS experiments that had higher local utilization throughout. Although the resource allocation within the machine will surely have some effect, it was likely dwarfed by the network communication. It is possible that the GHS experiments hit the same intra-machine contention and even likely that they hit the same intra-job contention since again no explicit constraints were placed on the individual processes and the CPUs they occupied. However, the fact that there was no *inter*-job contention more strongly supports the machine boundary hypothesis.

3.4 Component Statistics

The following figures depict component size statistics for all the different regimes. Plots are made on logarithmic axes for clarity.

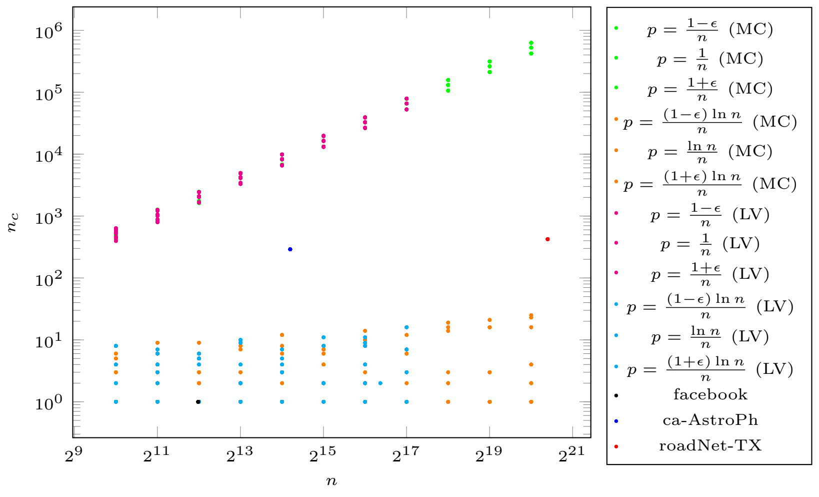

Figure 7: Component Counts for Various Regimes

Figure 6 reflects exactly the component mean sizes observed in the BFS experiments. There is not much more to say beyond what was covered in the previous report[13, pp. 17-19] except that this provides some empirical corroboration of the correctness or at least consistency of the GHS implementation with respect to the BFS implementation.

Figure 7 again confirms the statistics seen in the previous report.

4 Conclusions

It was somewhat surprising to see that the BFS algorithm slightly outperformed the GHS algorithm in the experiments on dense graphs. This was largely due to a weakness in the implementation, namely the pure reliance on a unicast messaging strategy. While the strategy was employed due to time constraints and over concerns of ensuring correctness, there are a number of opportunities for improvement.

In the first case, broadcast strategies could easily be employed during the MWOE search subphase for several of the message types, notably MWOE and FIND. Several bytes per message could have been saved. Moreover, the shell of the `bfs-coco` source served as the genesis for the `ghs-coco` source, and the map of incoming edges was already constructed during execution but never used. The incoming edges map would likely need to be supplemented by an incoming parents map to achieve the best optimization.

Another improvement would be to explore the use of subcommunicators. It may happen (and certainly did happen according to debugging runs executed during local development) that a machine exhausts its set of local components with active edges before other machines finish theirs. It would be better for those machines to become in essence "dark silicon" at the mere cost of a single message forfeiting participation to the savings of innumerable useless participations in the gather operations better left to the subset of active machines.

Still, the aggressive pruning of known singletons from the start and of joined edges during the merge phase paid off dividends in reducing overall phase count and, in the case of sparse graphs, empirical message and time complexity. More research into the areas suggested for improvement as well as more explicit and considered job allocation strategies would likely reap a lucrative benefit, especially when one considers the often overlooked cost of energy.

References

[1] Anonymous. *Pseudo-random number generation.* cppreference.com. Mar. 4, 2019. URL: `https://en.cppreference.com/w/cpp/numeric/random` (visited on 04/18/2019).

[2] Anonymous. *std::bernoulli_distribution.* cppreference.com. June 15, 2018. URL: `https://en.cppreference.com/w/cpp/numeric/random/bernoulli_distribution` (visited on 04/18/2019).

[3] Anonymous. *std::binomial_distribution.* cppreference.com. June 15, 2018. URL: `https://en.cppreference.com/w/cpp/numeric/random/binomial_distribution` (visited on 04/18/2019).

[4] Anonymous. *std::map.* cppreference.com. Feb. 10, 2019. URL: `https://en.cppreference.com/w/cpp/container/map` (visited on 04/16/2019).

[5] Anonymous. *std::mersenne_twister_engine.* cppreference.com. Feb. 25, 2019. URL: `https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine` (visited on 04/17/2019).

[6] Anonymous. *std::set*. cppreference.com. Nov. 20, 2018. URL: `https://en.cppreference.com/w/cpp/container/set` (visited on 05/04/2019).

[7] Anonymous. *std::uniform_int_distribution*. cppreference.com. June 15, 2015. URL: `https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution` (visited on 04/17/2019).

[8] Anonymous. *std::unordered_set*. cppreference.com. Nov. 19, 2018. URL: `https://en.cppreference.com/w/cpp/container/unordered_set` (visited on 04/16/2019).

[9] Richard Graham et al., eds. *MPI: A Message-Passing Interface Standard*. Version 2.1. Message Passing Interface Forum. June 3, 2008. URL: `https://www.mpi-forum.org/docs/mpi-2.1/mpi21-report.pdf` (visited on 04/16/2019).

[10] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. Stanford University. June 2014. URL: `https://snap.stanford.edu/data/` (visited on 04/17/2019).

[11] Gopal Pandurangan. *COSC 6326 Programming Assignment 2 and Final Project*. Apr. 5, 2019. URL: `https://elearning.uh.edu/bbcswebdav/pid-5628740-dt-content-rid-39995246_1/courses/H_20191_COSC_6326_15630/prog-assign2.pdf` (visited on 04/16/2019).

[12] Gopal Pandurangan. *Distributed Network Algorithms*. Feb. 20, 2019. URL: `https://sites.google.com/site/gopalpandurangan/dna` (visited on 04/16/2019).

[13] Michael Yantosca. “COSC6326 Programming Assignment 2”. Apr. 26, 2019.

5 Appendix A

Result graphs from the `bfs-coco` experiments on Erdos-Renyi graphs are provided here for convenience of comparison with the `ghs-coco` experiments. The plots have been redrawn with logarithmic x and y axes for legibility.

Component statistics are not included on account of redundancy. If one compares the graphs from the report on the `bfs-coco` experiments with the component statistics from this report[13, p. 18], one sees the exact same graph. This reproducibility implies an empirical corroboration of the correctness of both implementations or, at the very least, consistent wrongness.

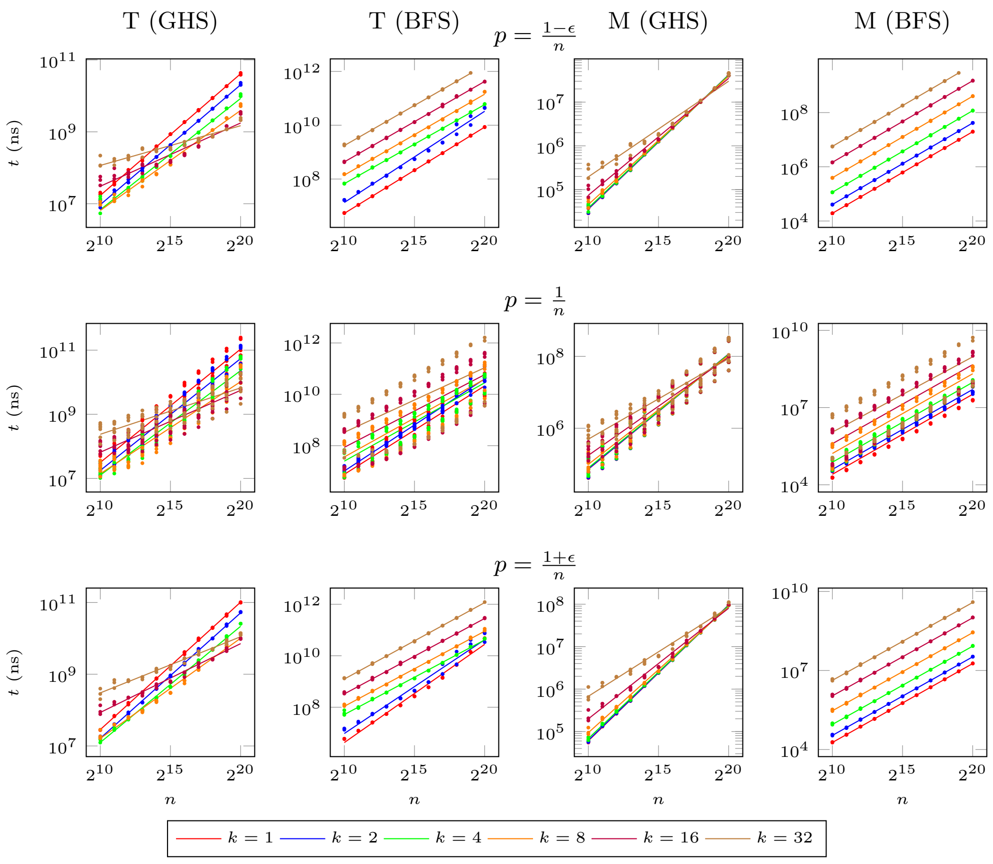

Figure 8: Empirical Complexities on MCER Graphs, $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$

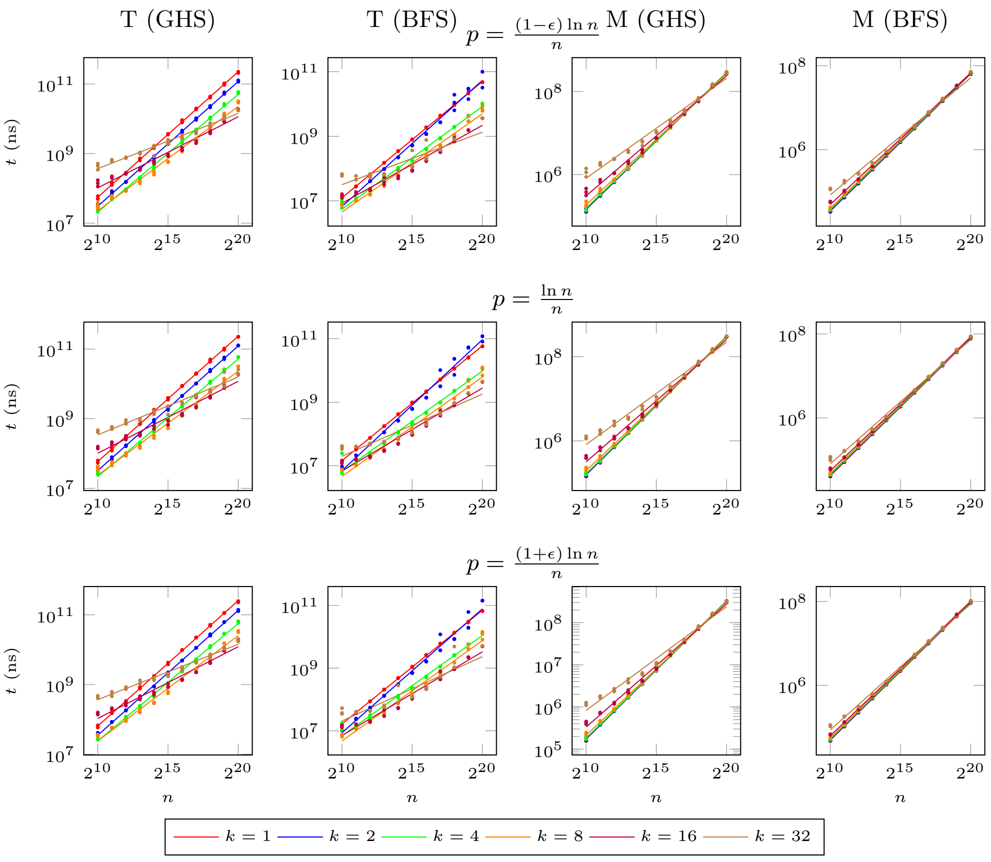

Figure 9: Empirical Complexities on MCER Graphs, $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$

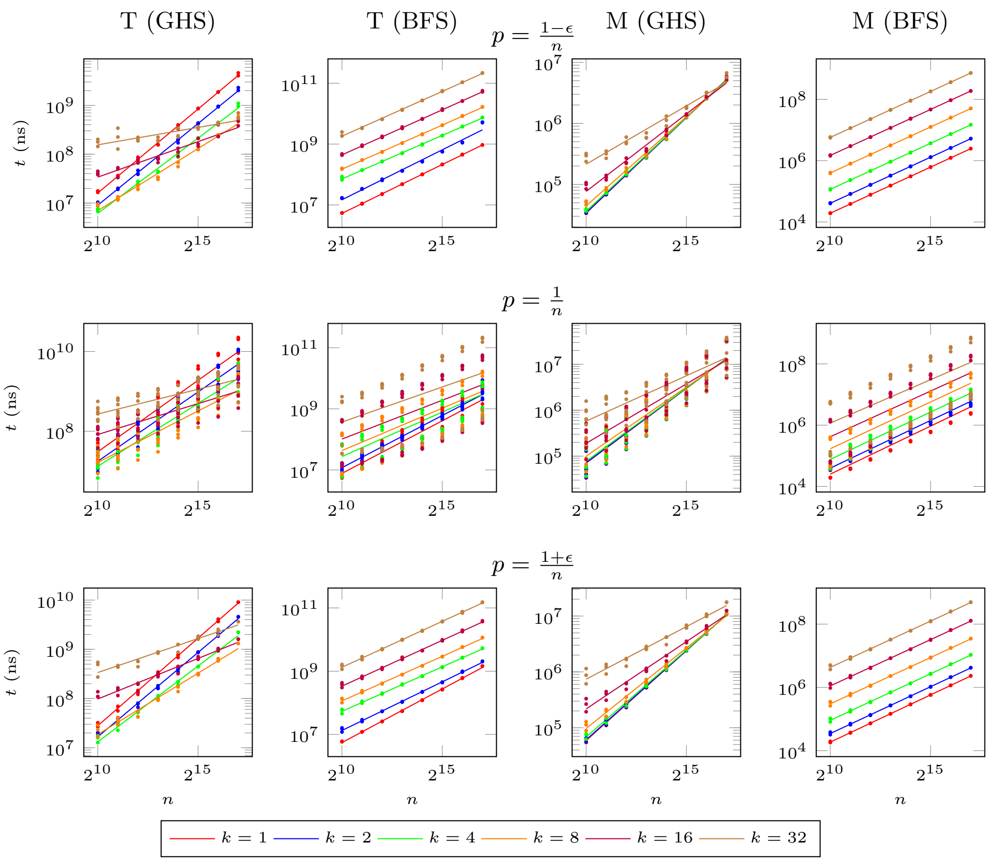

Figure 10: Empirical Complexities on LVER Graphs, $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$

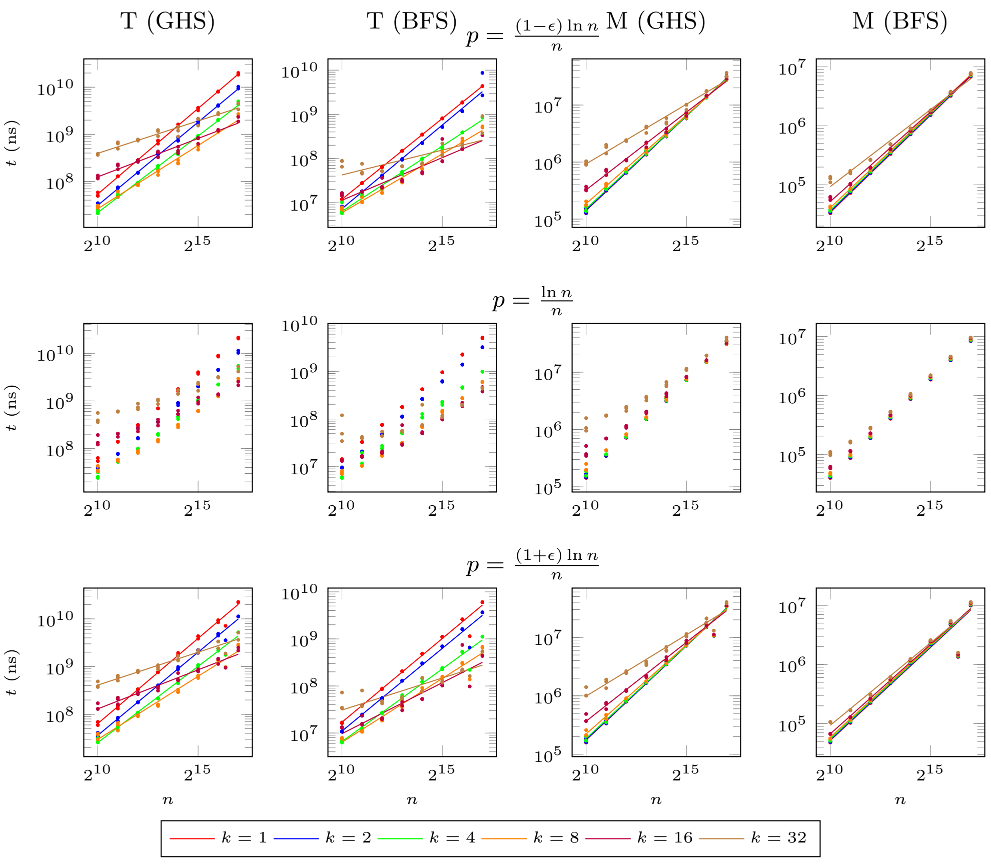

Figure 11: Empirical Complexities on LVER Graphs, $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$

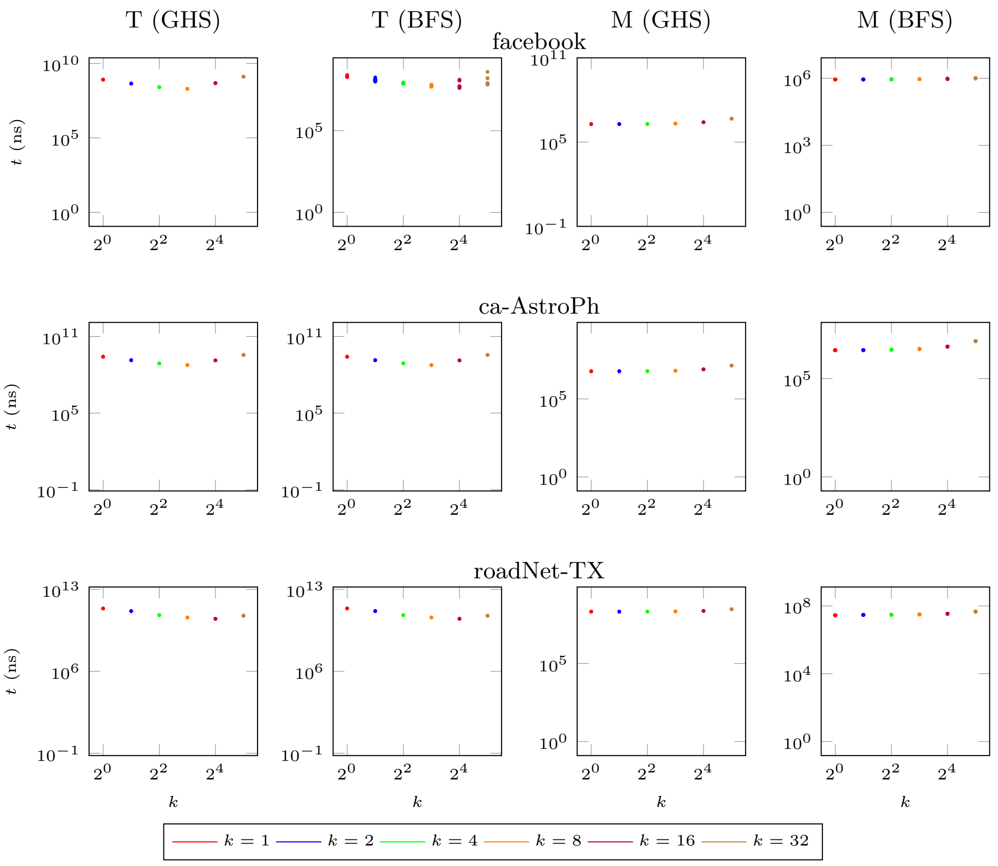

Figure 12: Empirical Complexities on Real-World Graphs