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. In this set of experiments, the discovery of connected components through the use of breadth-first search (BFS) is explored both on Erdos-Renyi graphs[11, p. 5] and real world datasets from the SNAP database[10]. A novel but incomplete method of Erdos-Renyi graph generation is proposed for improvements in generation efficiency.

2 Analysis

2.1 `text2mpig`

The graph representations available from the SNAP database[10] are in a textual format over which is difficult to efficiently parallelize data access. Whereas one could serially access the data from a predetermined root node and thereafter distribute it to other participating nodes, the loading time might be reduced if all the nodes could participate in the loading by each focusing on a subset of the data to be loaded. At rest, the vertex-centric model has a disadvantage against the edge-centric model since the variability of node composition in the absence of a guarantee of isomorphism or regularity necessitates additional header information on top of the unevenness of distribution. The optimal distribution of data cannot be determined *a priori* by an unbiased strategy.

Conversely, the read load of an edge-centric model can be evenly distributed when encoded in an efficient packed binary format since all edges are symmetric with each other provided that the vertex labels can be expressed in a fixed-width format. It is likely for this reason that the SNAP database provides models as a set of edges. A little preprocessing work up front could yield dividends, even when the algorithm model is vertex-centric. To this end, the `txt2mpig` utility was created with the additional benefit of a decrease in storage costs at rest since each node label only requires 4 bytes (each edge 8 bytes).

The resulting edge-centric binary files exhibited expected compression by virtue of being more efficient on average in terms of label storage.

Real-World Graph	**Text File Size**	**Binary File Size**
facebook_combined	835K	690K
ca-AstroPh	5.1M	3.1M
roadNet-TX	57M	30M

2.2 `genmpig`

The same binary file format could easily be generated in a stream-processing fashion for an Erdos-Renyi probabilistic graph. Two variants of the algo-

Algorithm 1 TEXT-TO-MPI-GRAPH, Algorithm for Converting Edge-Centric Text Model to Compact Edge-Centric Form

Require: e_B, the number of edges per output block per machine
Require: k, the number of participating nodes
Require: r, the participating process rank
  Open the output file for writing each edge (u, v) as $<u|v>$.
  while ¬EOF **do**
    if $r = 0$ **then**
      Read a line of text.
      if the line is of the form "u v" **then**
        Add $<u|v>$ to the binary read buffer.
      else
        Discard the line.
      Check for EOF.
    Broadcast the updated read offset to all processes.
    Broadcast the EOF status to all processes.
    if read buffer is at capacity ($16e_Bk$ bytes) or EOF **then**
      Wait for the previous non-blocking write to finish.
      Scatter the read buffer evenly among all the processes.
      Start a non-blocking write at an offset linearly scaled by r to avoid contention.
      Increment the absolute write offset for the next pass.
      Tare the read buffer offset.
  Wait for any outstanding disk writes.
  Close the output file.

rithm were created, largely out of a necessity to find a more time-efficient graph generator. Unfortunately, as the names suggest, the latter algorithm sacrificed some correctness for the sake of speed.

Algorithm 2 GENERATE-MPI-GRAPH, Distributed Algorithm for Generating an Erdos-Renyi Graph Stored in Compact Edge-Centric Form

Require: e_B, edges per output block per machine
Require: p, the probability of any given edge
Require: n, the graph population size (nodes)
Require: r, the process rank
  Seed a PRNG for a Bernoulli distribution with decision probability p.
  Open the output file for writing each edge (u, v) as $<u|v>$.
  for $i \in [0, n-1]$ **do**
    $u \leftarrow i + r$
    if $u < n$ **then**
      Add an "edge" (u, u) to ensure inclusion of u even as a singleton.
      if write buffer has reached capacity **then**
        Dump the buffer via a shared file pointer.
        Tare the write buffer offset.
      Generates incident edges from u according to Algorithm 3 or Algorithm 4
  Dump remainder of write buffer if partially full.
  Wait for final write.
  Close the output file.

Algorithm 3 CHOOSE-EDGES-LAS-VEGAS, Naive Edge Selection Algorithm with Guaranteed Correctness

Require: u, the source node label
Require: P_E, a Bernoulli distribution with decision probability p
  for $v \in [u+1, n-1]$ **do**
    Take sample E_{uv} from space with distribution P_E.
    if $E_{uv} = 1$ **then**
      Add $<u|v>$ to the binary write buffer.
      if write buffer has reached capacity **then**
        Dump the buffer via a shared file pointer.
        Tare the write buffer offset.

Algorithm 4 CHOOSE-EDGES-MONTE-CARLO, Time-Efficient Edge Selection Algorithm with Suboptimal Correctness

Require: u, the source node label
Require: P_v, a uniform distribution over $[u+1, n-1]$
Require: B_E, a $Bin(n-1-u, p)$ distribution
  Sample B_E to get a number of successful edge creations E_u.
  for $e \in [0, E_u - 1]$ **do**
    $v \leftarrow$ a sample from P_v
    Add $<u|v>$ to the binary write buffer.
    if write buffer has reached capacity **then**
      Dump the buffer via a shared file pointer.
      Tare the write buffer offset.

The asymptotic time benefits of Algorithm 4 over Algorithm 3 are fairly apparent, particularly when one considers the overhead necessary to generate a combinatorial set of coin flips. However, the use of the uniform distribution to choose an $O(np)$ sequence of numbers runs the risk of sampling duplicate edges. Even when $p = 1$, the algorithm did not always generate a complete graph. A simple shuffle algorithm would be a better choice if not for the onerous *space* requirements at the data scales under consideration.

2.3 `bfs-coco`

The BFS algorithm for finding connected components is given below in Algorithm 5. 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 6 is actually abstracted out into a separate function. All other sub-algorithms run within the main context to avoid unnecessary function call overhead, particularly Algorithm 7.

Algorithm 5 BFS-Connected-Components, Distributed BFS Algorithm for Determining 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.
  $(V_{in}, E_{in}) \leftarrow$ Load-Edge-To-Vertex-Centric(F, k)
  Initialize exchange info X_B for the broadcast phase.
  Initialize exchange info X_U for the upcast phase.
  $T \leftarrow \emptyset$ {component set}
  $B \leftarrow \emptyset$ {broadcast vertex set}
  $U \leftarrow \emptyset$ {upcast vertex set}
  $V_{out} \leftarrow \emptyset$ {finished vertex set}
  $\omega_r \leftarrow V_{in} = \emptyset$ {local termination condition}
  $\Omega \leftarrow \wedge_{r=1}^{k} \omega_r$ {global termination condition}
  while $\neg\Omega$ **do**
    $i_{g,r} \leftarrow \min_{v \in V_{in}} i(v)$
    $i_g \leftarrow \min_{r=0}^{k-1} i_{g,r}$
    $m_g \leftarrow$ Machine-Hash(h_a, h_b, k, i_g)
    $|g|_r \leftarrow 0$
    $|g| \leftarrow 0$
    if $r = m_g$ **then**
      $s(V_{in}[i_g]) \leftarrow$ broadcast
      $B \leftarrow B \cup \{V_{in}[i_g]\}$
    while $|g| = 0$ **do**
      Execute BFS-Connected-Components-Broadcast phase.
      Execute BFS-Connected-Components-Upcast phase.
      $|g| \leftarrow \max_{r=0}^{k-1} |g|_r$
    $\omega_r \leftarrow V_{in} = \emptyset$
    $\Omega \leftarrow \wedge_{r=1}^{k} \omega_r$

Algorithm 6 Exchange, 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**
    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 7 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 $((h_a i_v + h_b) \mod M_{61}) \mod k$

Algorithm 8 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**
    $e_{uv} \leftarrow (E_r[2i], E_r[2i+1])$
    $e_{vu} \leftarrow (E_r[2i+1], E_r[2i])$
    $machine_u \leftarrow$ MACHINE-HASH$(h_a, h_b, k, e_{uv}[0])$
    $machine_v \leftarrow$ MACHINE-HASH$(h_a, h_b, k, e_{vu}[0])$
    Add e_{uv} to the exchange buffer for $machine_u$.
    Add e_{vu} to the exchange buffer for $machine_v$.
  EXCHANGE(X_e)
  $E_{in} \leftarrow emptyset$
  for $(i_u, i_v) \in E_{rcvd}$ **do**
    $u \leftarrow V_{in}[i_u]$
    if $u = \perp$ **then**
      $u \leftarrow \{i = u, i_p = u, i_g = u, |g| = 1, s = \text{UNGROUPED}, w = 0, N = \emptyset, C = \emptyset\ \}$
      $V_{in}[i_u] \leftarrow u$
    if $i_u \neq i_v$ **then**
      $N(u) \leftarrow N(u) \cup \{i_v\}$
      if $i_v \notin E_{in}$ **then**
        $E_{in}[i_v] \leftarrow \emptyset$
      $E_{in}[i_v] \leftarrow E_{in}[i_v] \cup \{i_u\}$
  for $u \in V_{in}$ **do**
    $w(u) \leftarrow |N(u)|$ {Avoid double-counting neighbors if duplicates in edge storage.}
  return (V_{in}, E_{in})

Algorithm 9 BFS-Connected-Components-Upcast, Broadcast Phase of Algorithm 5

for all $u \in B$ **do**
  $s(u) \leftarrow$ PENDING
  for all $i_v \in N(u)$ **do**
    $m_v \leftarrow$ Machine-Hash(h_a, h_b, k, i_v)
    $send(X_B)[m_v] \leftarrow send(X_B)[m_v] \cup \{i(u)\}$
  if $w(u) = 0$ **then**
    $U \leftarrow U \cup \{u\}$
Exchange(X_B)
Rewind(X_B)
$B \leftarrow \emptyset$
for all $i_u \in recv(X_B)$ **do**
  $m_u \leftarrow$ Machine-Hash(h_a, h_b, k, i_u)
  for all $i_v \in E_{in}[i_u]$ **do**
    $v \leftarrow V_{in}[i_v]$
    if $s(v) =$ UNGROUPED **then**
      $i_p(v) \leftarrow i_u$
      $s(v) \leftarrow$ BROADCAST
      $i_g(v) \leftarrow i_g$
      $B \leftarrow B \cup \{v\}$
    else
      $send(X_U)[m_u] \leftarrow send(X_U)[m_u] + (i_u, i(v), 0)$

Algorithm 10 BFS-Connected-Components-Upcast, Upcast Phase of Algorithm 5

for all $u \in U$ **do**
  $s(u) \leftarrow$ FINISHED
  if $i(u) = i_g$ **then**
    $|g|_r \leftarrow |g|(u)$
  else
    $m_p \leftarrow$ Machine-Hash$(h_a, h_b, k, i_p(u))$
    $send(X_U)[m_p] \leftarrow send(X_U)[m_p] + (i_p(u), i(u), |g|(u))$
    $V_{out} \leftarrow V_{out} \cup \{u\}$
$U \leftarrow \emptyset$
Exchange(X_U)
Rewind(X_U)
for all $(i_v, i_u, |g|_u) \in recv(X_U)$ **do**
  $v \leftarrow V_{in}[i_v]$
  $|g|(v) \leftarrow |g|(v) + |g|_u$
  if $|g|_u > 0$ **then**
    $C(v) \leftarrow C(v) \cup \{i_u\}$
  $w(v) \leftarrow w(v) - 1$
  if $w(v) = 0$ **then**
    $U \leftarrow U \cup \{v\}$
$V_{in} \leftarrow V_{in} \setminus V_{out}$

Theorem 1. *Algorithm 5 determines the connected components in an arbitrary graph of n vertices distributed over k machines with communication complexity $O(nk + m)$).*

Proof. Each node must serve as either a BFS tree root or subroot or leaf. Each top-level root found incurs by necessity $O(k)$ meta-messages of coordination since no optimizations are made in the singleton case. In the event that the graph is completely disconnected, this becomes $O(nk)$ total meta-messages.

Suppose that the graph is connected to some degree. Each edge must be traversed by the BFS: once during the broadcast phase, and once during the upcast phase. Thus, the message count in this respect is $O(m)$.

The communication complexity follows by simple addition. Whichever term is greater will necessarily dominate. In highly connected graphs, m will dominate. In sparsely connected grpahs, nk will dominate. □

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

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

- 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.

`bfs-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 5 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`[5] of vertex structures keyed by vertex ID. In each vertex structure, the collections of neighbors and BFS tree children were of type `std::unordered_set<uint32_t>`[8]. The incoming edge map used to facilitate faster lookup on incoming messages was accomplished with a `std::map<std::unordered_set<uint32_t>>`. The vertex output map was a <std::vector>[4] of vertex structures, but could just as easily have been a `std::map` (and was in earlier development).

Randomized elements[1] made use of the Mersenne twister PRNG `std::mt19937`[6] 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 $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$, 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, but the BFS search times actually increase quite drastically in proportion to k. Since the graphs in Figure 1 are largely disconnected, it may be that the message overhead of voting on singleton roots may dominate the search time. Moreover, the single round of no-op gather operations per singleton node probably induces overhead, as well.

The BFS times and load times are more in line with expectations in the regimes depicted by Figure 2. The total message complexity is two orders of magnitude lower than the experiments depicted by Figure 1. This further confirms the suspicion that the handling of singleton components is suboptimal under the current implementation. The round counts (i.e., counts of paired broadcast and upcast phases) are also significantly smaller, which stands to reason since more “rounds” would generate more messages.

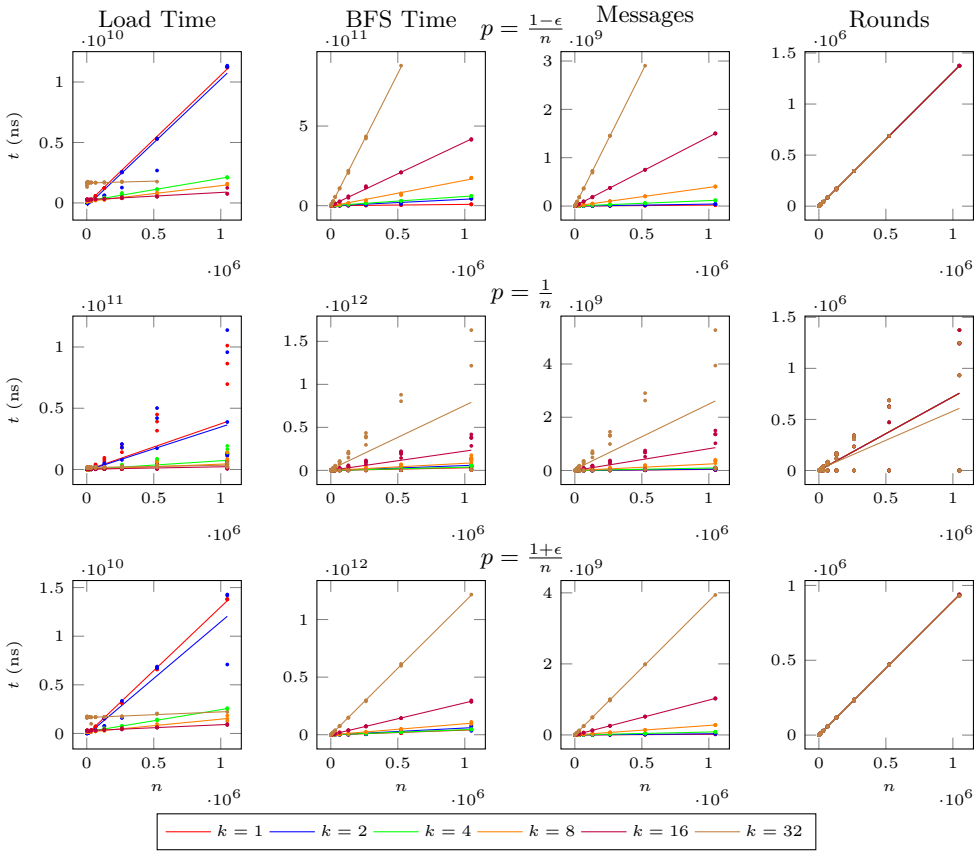

Figure 1: Empirical Complexity of Algorithm 5 on MCER Graphs, $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$

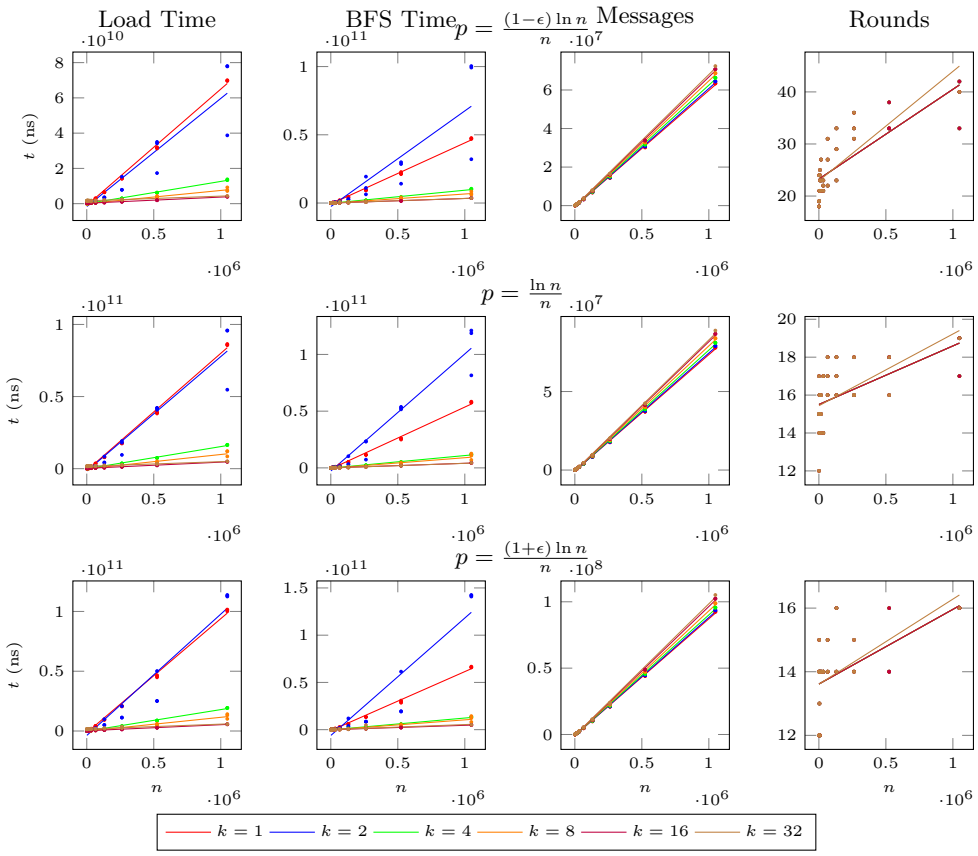

Figure 2: Empirical Complexity of Algorithm 5 on MCER Graphs, $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$

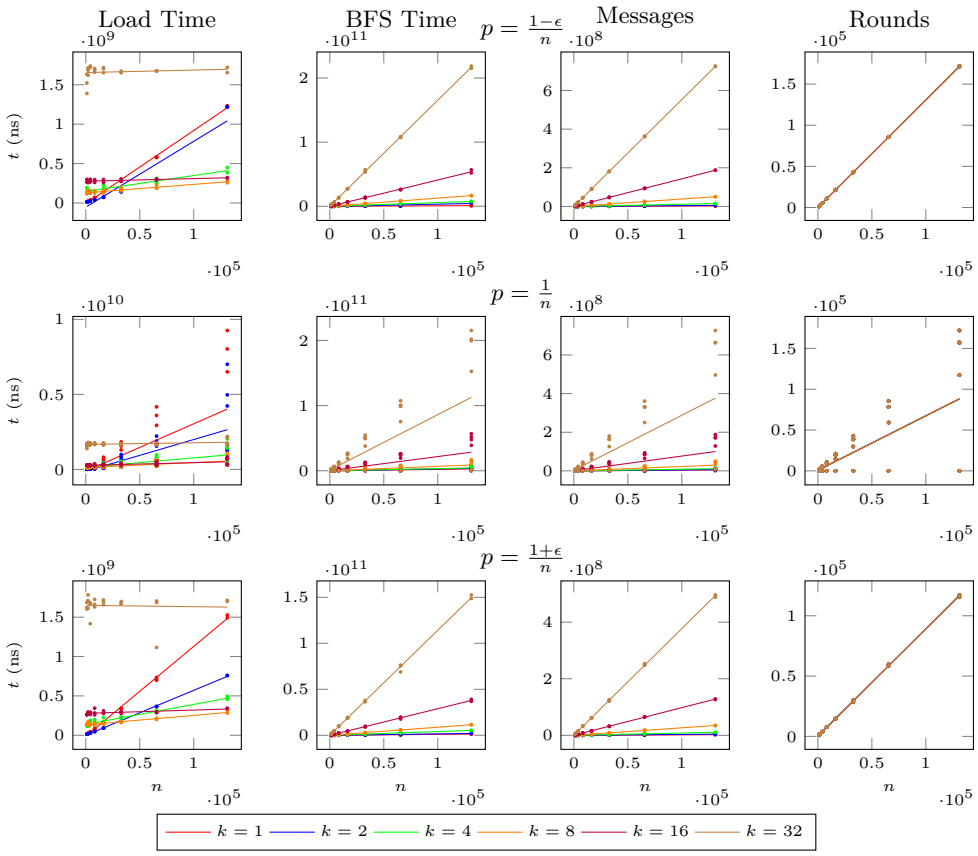

Figure 3: Empirical Complexity of Algorithm 5 on LVER Graphs, $p \in [\frac{1-\epsilon}{n}, \frac{1+\epsilon}{n}]$

The LVER graphs depicted in Figure 3 exhibit similar behavior to that shown by the MCER graphs depicted in Figure 1. This 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 discrepancy in wall-clock time between the connected and disconnected regimes is likewise reflected.

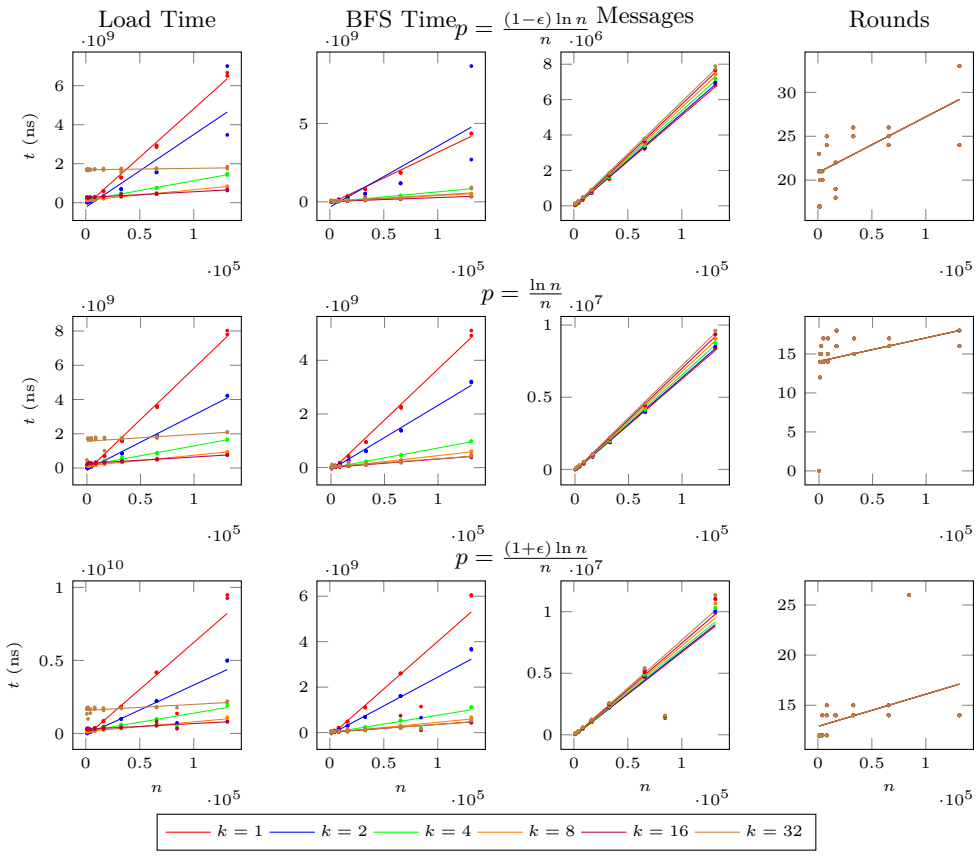

Figure 4: Empirical Complexity of Algorithm 5 on LVER Graphs, $p \in [\frac{(1-\epsilon)\ln n}{n}, \frac{(1+\epsilon)\ln n}{n}]$

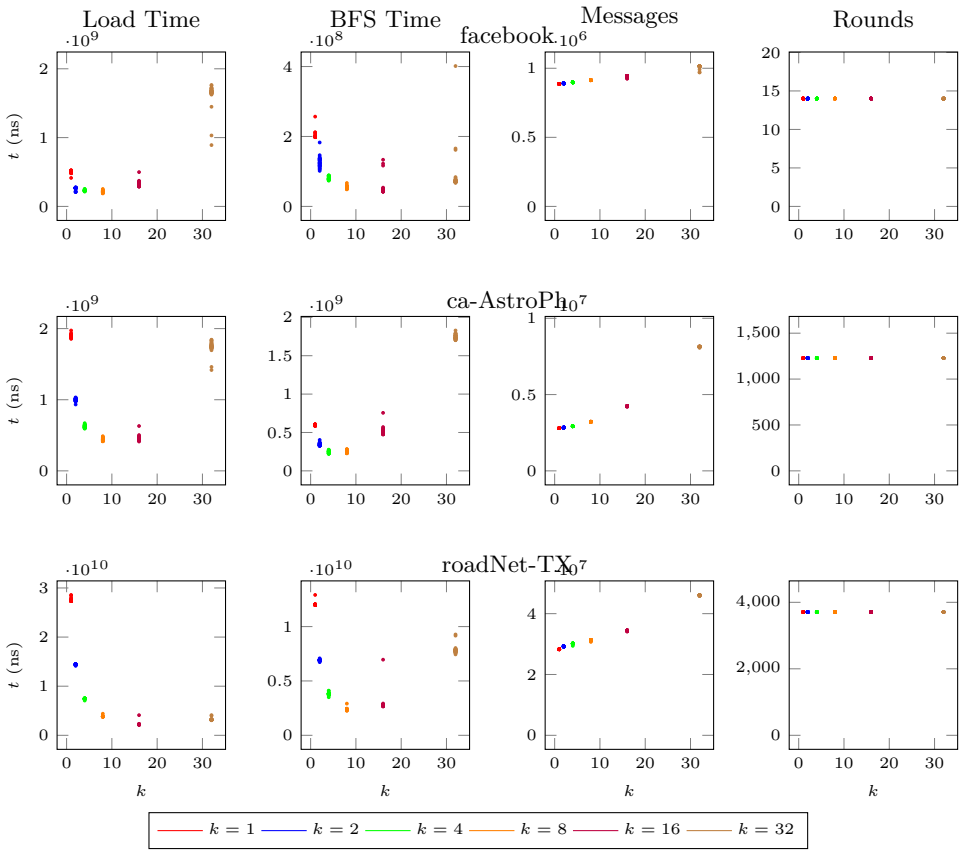

Figure 5: Empirical Complexity of Algorithm 5 on Real-World Graphs

3.3 Real-World Graphs

The real world graphs depicted in Figure 5 show the expected decrease in both BFS and load time for smaller values of k, but there appears to be an inflection point either at $k = 8$ (`ca-AstroPh`, `roadNet-Tx`) or $k = 16$ (`facebook`). The increase in execution time is particularly notable in the ca-AstroPh graph. This probably owes to the nature of the division of processes within the cluster for the experiments.

k	**Machines**	**Cores**
1	1	1

2	1	2
4	2	2
8	2	4
16	2	8
32	4	8

At the higher values of k, there was likely more contention for resources within the given machine since more processors were occupied per machine.

Because the number of available machines were limited by outages and concurrent jobs from other users, the experiments were done with a ceiling of 8 processes per machine. It is difficult to say whether the 8 processes were mapped to independent cores or to core hyperthreads since no constraints of that granularity were employed in the batch scripts. Even if the processes were always mapped to independent cores, the contention for memory and higher levels of cache could have created bottlenecks, especially in cases of large n.

3.4 Component Statistics

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

In Figure 6, the component mean size climbs with total vertex population size in the case where p is close to $\frac{\ln n}{n}$. This suggests a significantly higher degree of connectivity than the case where p is close to $\frac{1}{n}$, where the components appear to be singletons or very nearly singletons on average. It would seem that setting $\epsilon = 0.2$ may be too tight to observe the transition threshold alluded to in the assignment specification[11, p. 5]. Widening ϵ to 0.5 or greater may have offered a more stark comparison. It is interesting to note that the component size for `facebook` suggests complete connection whereas that for `ca-AstroPh` and `roadNet-TX` exhibit slightly lower connectivity, though not as low as the Erdos-Renyi graphs where p is close to $\frac{1}{n}$.

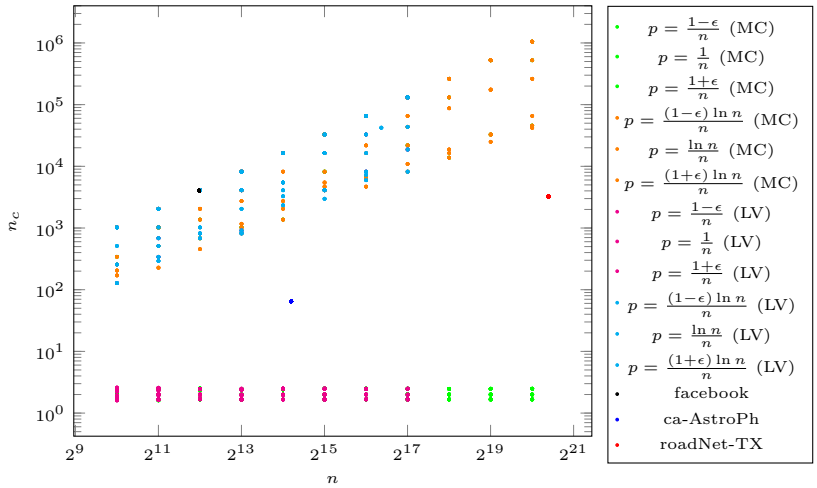

Figure 6: Component Mean Size for Various Regimes

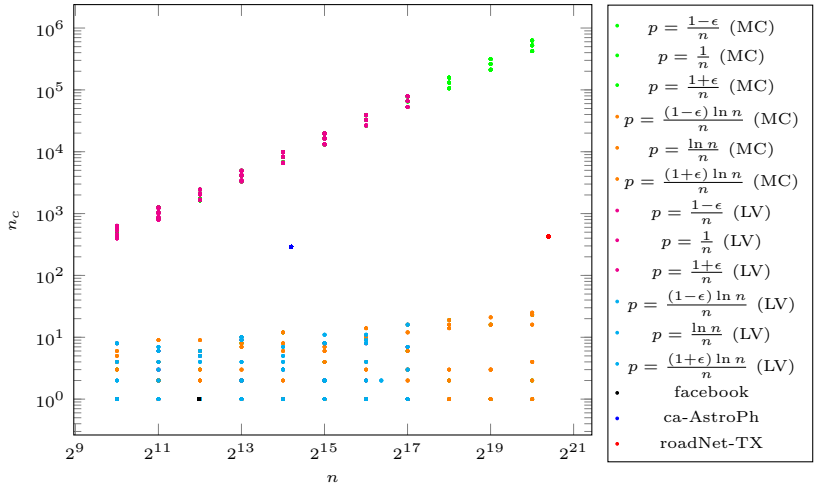

Figure 7: Component Counts for Various Regimes

Figure 7 confirms the suggestions arising from Figure 6 in a more concrete fashion. The Facebook graph is completely connected. The network of co-publishing astrophysicists seems to have between 100 and 1000 clusters of scientists. This probably could be considered as a marker for institution or subfield, though without more explicit data it is impossible to say for certain. The Texas road network graph is a little more surprising since one might have expected something closer to complete connectivity. The components could be a marker for the urban/rural divide since all the major cities in Texas would have been connected by major freeways. The isolates would likely be smaller towns or possibly farmland and ranches with relatively self-contained road subnets.

4 Conclusions

Some optimizations around the handling of singleton components are sorely wanted here. One optimization would be to preprocess singletons locally in a single pass over a local population and perform an all-gather operation to ensure that the forest was up to date among all participating machines. There would be no need for broadcast or upcast phases, which would avoid superfluous messages in terms of voting, exchange, and termination checking.

The Monte Carlo edge selection scheme appears to have enjoyed mild success as a simulacrum for the exceedingly more time-consuming Las Vegas edge selection scheme. With some minor tweaks, the former coould be improved to more correctly simulate the intensive Bernoulli trials of the latter.

The primary deficiency of the implementation is in its poor performance for high values of k. Some careful thought will need to be given to overcome these issues. While using utilities like `callgrind` uncovered major inefficiencies in the implementation, it was difficult to test longer runs because of the inherent overhead that profiling utilities would add to an already long execution time. A focus on moving as much computation as possible into the local sphere would likely yield the best dividends.

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. Nov. 20, 2018. URL: https://en.cppreference.com/w/cpp/container/map (visited on 04/16/2019).

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

[6] 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).

[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).